Automated Configuration of Algorithms for Solving Hard Computational Problems

by

Frank Hutter

Dipl. Inf., Darmstadt University of Technology, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

October 2009

Abstract

The best-performing algorithms for many hard problems are highly parameterized. Selecting the best heuristics and tuning their parameters for optimal overall performance is often a difficult, tedious, and unsatisfying task. This thesis studies the automation of this important part of algorithm design: the configuration of discrete algorithm components and their continuous parameters to construct an algorithm with desirable empirical performance characteristics.

Automated configuration procedures can facilitate algorithm development and be applied on the end user side to optimize performance for new instance types and optimization objectives. The use of such procedures separates *high-level cognitive tasks* carried out by humans from *tedious low-level tasks* that can be left to machines.

We introduce two alternative algorithm configuration frameworks: iterated local search in parameter configuration space and sequential optimization based on response surface models. To the best of our knowledge, our local search approach is the first that goes beyond local optima. Our model-based search techniques significantly outperform existing techniques and extend them in ways crucial for general algorithm configuration: they can handle categorical parameters, optimization objectives defined across multiple instances, and tens of thousands of data points.

We study how many runs to perform for evaluating a parameter configuration and how to set the *cutoff time*, after which algorithm runs are terminated unsuccessfully. We introduce data-driven approaches for making these choices adaptively, most notably the first general method for adaptively setting the cutoff time.

Using our procedures—to the best of our knowledge still the only ones applicable to these complex configuration tasks—we configured state-of-the-art tree search and local search algorithms for SAT, as well as CPLEX, the most widely-used commercial optimization tool for solving mixed integer programs (MIP). In many cases, we achieved improvements of orders of magnitude over the algorithm default, thereby substantially improving the state of the art in solving a broad range of problems, including industrially relevant instances of SAT and MIP.

Based on these results, we believe that automated algorithm configuration procedures, such as ours, will play an increasingly crucial role in the design of high-performance algorithms and will be widely used in academia and industry.

Table of Contents

List of Tables

List of Figures

List of Algorithms and Procedures

List of Symbols

Symbol	Meaning
$\boldsymbol{\Theta}$	Parameter configuration space, space of allowable parameter configurations
$\boldsymbol{\theta}$	Parameter configuration, element of $\boldsymbol{\Theta}$
$\boldsymbol{\theta}_{i:j}$	Vector of parameter configurations, $[\boldsymbol{\theta}_i, \boldsymbol{\theta}_{i+1}, \ldots, \boldsymbol{\theta}_j]$
$\theta_{i,j}$	value of j-th parameter of parameter configuration $\boldsymbol{\theta}_i$
d	dimensionality of $\boldsymbol{\Theta}$ (number of parameters to be tuned)
Π	Set of problem instances
π	Problem instance, element of Π
s	Random number seed
o	Outcome of a single target algorithm run (*e.g.*, runtime, solution cost)
$\mathbf{R}$	Sequence of runs, with elements $(\boldsymbol{\theta}_i, \pi_i, s_i, \kappa_i, o_i)$ as defined in Section 1.
$\mathbf{R}[i]$	ith element of $\mathbf{R}$
$\mathbf{R}_{\boldsymbol{\theta}}$	Subsequence of runs using configuration $\boldsymbol{\theta}$ (*i.e.*, those runs with $\boldsymbol{\theta}_i = \boldsymbol{\theta}$)
$N(\boldsymbol{\theta})$	Length of $\mathbf{R}_{\boldsymbol{\theta}}$
$c_{valid}(\boldsymbol{\theta})$	Empirical cost estimate of $\boldsymbol{\theta}$ using all $\langle$instance, seed$\rangle$ combinations in the training set
$\hat{c}(\boldsymbol{\theta})$	Empirical cost statistic over the $N(\boldsymbol{\theta})$ runs with $\boldsymbol{\theta}$: $\hat{\tau}(\{\mathbf{R}[i].o \vert \boldsymbol{\theta}_i = \boldsymbol{\theta}\})$
$\hat{c}_N(\boldsymbol{\theta})$	Identical to $\hat{c}(\boldsymbol{\theta})$, emphasizing that $N(\boldsymbol{\theta}) = N$
$\mathcal{M}$	Response surface model
$p(x\vert y)$	The conditional probability distribution of x given y
$\mathcal{N}(\mu, \Sigma)$	Normal distribution with mean μ and covariance matrix Σ
$p(x\vert y) = \mathcal{N}(x\vert\mu, \Sigma)$	Conditional probability distribution of x given y is a Normal distribution with mean μ and covariance matrix Σ

Glossary

Algorithm Configuration The formally-defined algorithm configuration problem; see Definition 1 on page 10

Algorithm Configuration Procedure See *Configuration Procedure*

AC ACTIVECONFIGURATOR, a model-based framework for the configuration of algorithms with discretized configuration spaces; see Section 12.3.2

Blackbox Optimization Problem The problem of minimizing a "blackbox" function, that is, a function the only information for which is its value at query points we can sequentially select; see Section 1.2.1

Captime A time bound, usually denoted by κ, after which unsuccessful target algorithm runs are terminated

Cutoff time See *Captime*

Conditional Parameter Target algorithm parameter that is only defined if certain "higher-level" parameters take on specific values

Configuration See *Parameter Configuration*

Configuration Procedure An executable procedure, *i.e.*, an algorithm, for optimizing the parameters of target algorithms

Cost Distribution $\mathbb{P}_{\{\boldsymbol{\theta}\}}$, a distribution of costs of single runs of target algorithm $\mathcal{A}$ with configuration $\boldsymbol{\theta}$, on instances π drawn from a distribution $\mathcal{S}$ and pseudo-random number seeds s drawn uniformly at random from a set of allowable seeds

Cost Measure $c(\boldsymbol{\theta})$, the criterion being optimized, a statistical population parameter, τ, of the cost distribution, $\mathbb{P}_{\{\boldsymbol{\theta}\}}$. For example, expected runtime or median solution cost

Cost Statistic See *Empirical Cost Statistic*

Configurator See *Configuration Procedure*

DACE model A noise-free GP model introduced by Sacks et al. (1989); DACE is an acronym for "Design and Analysis of Computer Experiments". In SPO and its variants, the DACE model is fitted on empirical cost statistics, and we use the term in this meaning

Deterministic Blackbox Optimization Problem See *Blackbox Optimization Problem*

EIC Expected Improvement Criterion, a function that quantifies how promising a parameter configuration is by using predictions of a response surface model

EGO Efficient Global Optimization, a model-based approach for deterministic continuous blackbox optimization described by Jones et al. (1998)

Empirical Cost Statistic $\hat{c}_N(\boldsymbol{\theta})$, empirical statistic of the cost distribution $c(\boldsymbol{\theta})$ based on N samples

GP Model Regression model based on a Gaussian stochastic process; see Section 9.2

Incumbent Parameter Configuration $\boldsymbol{\theta}_{inc}$, the parameter configuration a configurator judges to be best and would return if it had to terminate. The incumbent changes over time, and we denote the incumbent at time step t as $\boldsymbol{\theta}_{inc}(t)$

MIP The mixed integer programming problem; see Section 3.1.2

PAR Penalized average runtime of a set of runs with a captime κ, where runs timed out after the captime are counted as $a \cdot \kappa$; we set penalization constant a to 10 throughout this thesis, see Section 3.4

Parameter Configuration $\boldsymbol{\theta}$, a vector $\boldsymbol{\theta}$ of parameter values $(\theta_1, \ldots, \theta_d)^\intercal$ that uniquely identify a single executable variant of a target algorithm

Parameter Configuration Space $\boldsymbol{\Theta}$, set of all parameter configurations $\boldsymbol{\theta}$ a target algorithm allows; this is a subset of the Cartesian product of the domains of each parameter, $\boldsymbol{\Theta}_1 \times \cdots \times \boldsymbol{\Theta}_d$

RF model Regression model based on a random forest; see Section 11.1

Response Surface Model Regression model that aims to predict the cost measure, $c(\boldsymbol{\theta})$, of parameter configurations, $\boldsymbol{\theta}$. In the context of multiple instances, these models can also be used to predict the cost distribution for a combination of a parameter configuration and an instance

RMSE Root mean squared error, a measure of model quality $\in [0, \infty)$ (the lower the better); RMSE measures the root of the squared differences between model predictions and validation costs, see Definition 14 on page 148

EIC quality A measure of model quality $\in [-1, 1]$ (the higher the better); it measures the Spearman correlation coefficient between expected improvement based on model predictions, and validation costs, see Definition 13 on page 148

Quality of predictive ranks A measure of model quality $\in [-1, 1]$ (the higher the better); it measures the Spearman correlation coefficient between model predictions and validation costs, see Definition 12 on page 148

SAT The propositional satisfiability problem; see Section 3.1.1

SPO Sequential Parameter Optimization, a model-based framework for stochastic continuous blackbox optimization first described by Bartz-Beielstein et al. (2005)

SKO Sequential Kriging Optimization, a model-based approach for stochastic continuous blackbox optimization described by Huang et al. (2006)

Stochastic Blackbox Optimization Problem Similar to a deterministic *Blackbox Optimization Problem*, but the observations we make at a query point, $\boldsymbol{\theta}$, are samples of a cost distribution, $\mathbb{P}_{\{\boldsymbol{\theta}\}}$, rather than deterministic values of a function, $f(\boldsymbol{\theta})$; see Section 1.2.1

Target Algorithm A parameterized algorithm whose parameters are to be optimized

Training Performance of a Configurator at Time t Empirical cost estimate of the configurator's incumbent at time t, $\boldsymbol{\theta}_{inc}(t)$, across the runs the configurator performed with $\boldsymbol{\theta}_{inc}(t)$

Test Performance of a Configurator at Time t Empirical cost estimate of the configurator's incumbent at time t, $\boldsymbol{\theta}_{inc}(t)$, for runs with $\boldsymbol{\theta}_{inc}(t)$ on test set

Training Set Set of $\langle$problem instance, pseudo-random number seed$\rangle$ combinations used during configuration

Test Set Set of $\langle$problem instance, pseudo-random number seed$\rangle$ combinations used for offline test purposes; disjoint from training set

Validation Cost $c_{valid}(\boldsymbol{\theta})$, empirical cost estimate of parameter configuration $\boldsymbol{\theta}$ using all $\langle$instance, seed$\rangle$ combinations in the training set

Acknowledgements

I am deeply grateful to my three co-supervisors, Holger Hoos (my principal supervisor), Kevin Leyton-Brown, and Kevin Murphy. With their energy, creativity, and rigor, they shaped my work in many important ways, while at the same time giving me the freedom to explore my own ideas. Together, throughout my Ph.D., they made sure that I develop principled methods that walk the fine line between elegance and good performance in practice. Without their encouragement to always go a step further than what I thought is possible, I would have probably shied away from problems as hard as configuring CPLEX (which eventually became one of the two most exciting application domains in this thesis). Many thanks also to Alan Mackworth, the fourth member of my supervisory committee, for making sure I work on interesting but feasible problems and stay on track. I also thank my external examiner Henry Kautz, as well as my university examiners Michael Friedlander and Lutz Lampe, for their valuable feedback.

Holger and Kevin L.-B. have also helped me greatly during the writing process of this thesis. With their detailed and insightful feedback, they have improved the end result in many ways, including content, structure, narrative, grammar, and style.

I gratefully acknowledge funding from various sources. Firstly, I thank the German National Academic Foundation ("Studienstiftung des deutschen Volkes") for continuing to fund me after my M.Sc. thesis, through a one-year foreign exchange scholarship and a three-year doctoral fellowship. In this context, my sincere gratitude to Wolfgang Bibel, who nominated me for a fellowship in the first place and who supported me as a mentor for many years. I also received a university graduate fellowship and a R. Howard Webster Foundation fellowship from the University of British Columbia (UBC).

Co-authors are acknowledged in a separate section of the frontmatter, but I would like to thank some of them in particular. I would like to thank Thomas Stützle, my M.Sc. thesis supervisor, for the continued fruitful collaboration and many insightful discussions. I would also like to express my gratitude to Youssef Hamadi for hiring me as a research intern at Microsoft Research in Cambridge, UK, in the summer of 2005. The 12 weeks spent under his guidance have been very enjoyable and influenced me in many ways. Most importantly in the context of my Ph.D., they boosted my interest in automated parameter optimization and helped my decision to choose it as a dissertation topic. I am indebted to Lin Xu for the many fruitful discussions we shared, for fixing countless problems with our computer cluster and database, and for being a pleasant colleague that brought joy to many of my days. The productive and enjoyable collaboration with Domagoj Babić has opened my eyes to the full

potential of automated algorithm configuration, removing my final doubts about its eventual usefulness for practitioners.

Many thanks to everyone who has helped me out in one way or the other. In particular, thanks to Dave Tompkins for help with UBCSAT, to Daniel Le Berre for help with SAT4J, to Belarmino Adenso-Díaz for providing the CALIBRA system and benchmark data, and to Theodore Allen for access to the SKO source code. Many thanks to Chris Fawcett for taking over and maintaining the PARAMILS project and fixing some bugs to help external users. Also thanks to Christian Bang, my office mate during my M.Sc. thesis, who helped me automate my experiments based on Ruby scripts; over time, these scripts grew into the first version of PARAMILS.

I would like to thank many other members of the laboratory for computational intelligence (LCI) and the laboratory for bioinformatics, empirical and theoretical algorithmics (β) at UBC computer science, both for their friendship and many interesting chats about our respective work. In LCI, these include Eric Brochu, Mike Chiang, Nando de Freitas, Matt Hoffman, Roman Holenstein, Emtiyaz Khan, Hendrik Kueck, and Mark Schmidt, and in β Mirela Andronescu, Chris Fawcett, Ashiqur KhudaBukhsh, Kevin Smyth, Chris Thachuk, Dave Tompkins, Dan Tulpan, and Lin Xu.

I was lucky enough to spend almost three years in Green College, a residential graduate college at UBC. I thank my many Green College friends from all kinds of academic disciplines for broadening my perspective on both life and research, and for the endless fun we shared.

Finally, I would like to thank my loving partner Tashia for putting up with my nights and weekends spent working and for bringing sunshine into my life when all else failed. I am extremely grateful to her and to my family, especially my parents, my brother, and his wife, for their unconditional love, care, advice, and emotional support. I also would like to thank my family and many friends in Germany for the peaceful and enjoyable environment I can always be certain to find when I return home. I especially thank my parents for supporting my choice to follow the path that led me so far away from home, for so long.

Dedication

To my parents, Ilona and Gerald Hutter

Co-Authorship Statement

Over the past five years I have had the good fortune to work with outstanding co-authors. I am indebted to them for their many contributions to the work described in this thesis.

The development of PARAMILS had a major impact on my PhD thesis. An early version of PARAMILS goes back to an unpublished side project during my MSc. thesis (see Appendix A of Hutter, 2004), which was co-supervised by Thomas Stützle and Holger Hoos. In joint work with Holger and Thomas during my PhD, we improved PARAMILS and first published it at AAAI-07 (Hutter et al., 2007b). This formed the basis for Chapter 5. Subsequently, Kevin Leyton-Brown got involved in the project and, notably, contributed the idea for the adaptive capping mechanism discussed in Chapter 7; thanks also to Kevin Murphy for valuable discussions of that topic. Adaptive capping is discussed in a comprehensive journal article on PARAMILS with Holger Hoos, Kevin Leyton-Brown, and Thomas Stützle, which also introduced the application to CPLEX and has been accepted for publication at JAIR (Hutter et al., 2009c). That article contains much of the material in Chapters 5, 7, and 8.

Chapter 6 is based on joint work with Domagoj Babić, Holger Hoos, and Alan Hu that appeared at FMCAD-07 (Hutter et al., 2007a). Domagoj deserves credit for much of the research presented in that chapter; it is based on a joint case study, in which he played the role of algorithm designer while I performed the algorithm configuration.

Chapter 4 is based on joint work with Holger Hoos and Kevin Leyton-Brown, which is about to be submitted for publication (Hutter et al., 2009d). Thomas Stützle provided valuable feedback on an early draft of that work.

Chapters 9 and 10 are primarily based on a conference publication at GECCO-09 (Hutter et al., 2009e), co-authored by Holger Hoos, Kevin Leyton-Brown, and Kevin Murphy, as well as a book chapter with the same co-authors and Thomas Bartz-Beielstein (Hutter et al., 2009a).

Chapters 11 through 13 are all based on yet-unpublished joint work with Holger Hoos, Kevin Leyton-Brown, and Kevin Murphy.

A number of my publications are only indirectly reflected in this thesis. My first publications on (per-instance) parameter optimization are jointly with Youssef Hamadi (Hutter and Hamadi, 2005), and with Youssef Hamadi, Holger Hoos, and Kevin Leyton-Brown (Hutter et al., 2006). I have not included that work prominently in this thesis since it dealt with a different (albeit very interesting) problem; it is briefly discussed in Section 14.3.5. Joint publications with Lin Xu, Holger Hoos, and Kevin Leyton-Brown on per-instance algorithm selection (Xu et al., 2007b, 2008) fall into the same category. The SAT instance features we use in Section 13.1 originated in this work.

Part I

Algorithm Configuration: The Problem

—in which we introduce and motivate the algorithm configuration problem and discuss related work

Chapter 1

Introduction

Civilization advances by extending the number of important operations which we can perform without thinking of them.
—Alfred North Whitehead, English mathematician and philosopher

Parameterized algorithms are abundant in computer science and its applications. For many computational problems, there exist a wide array of solution approaches, and it is the task of computer scientists to identify the one that best matches the domain-specific user requirements. Typically, once a general solution approach has been chosen, there are a number of subsequent lower-level choices to be made before arriving at a complete algorithm specification. Often, some of those choices are left open; these *free parameters* allow users to adapt the algorithm to their particular scenario. Particularly, this is the case for the heuristic algorithms used for solving computationally hard problems. As an example, consider CPLEX, the most widely used commercial optimization tool for solving mixed integer programming problems.[1] Its latest version, 11.2, has about 80 parameters that affect the solver's search mechanism and can be configured by the user. Other examples of parameterized algorithms can be found in areas as diverse as sorting (Li et al., 2005), linear algebra (Whaley et al., 2001), numerical optimization (Audet and Orban, 2006), compiler optimization (Cavazos and O'Boyle, 2005), parallel computing (Brewer, 1995), computer vision (Muja and Lowe, 2009), machine learning (Maron and Moore, 1994; Kohavi and John, 1995), database query optimization (Stillger and Spiliopoulou, 1996), database server optimization (Diao et al., 2003), protein folding (Thachuk et al., 2007), formal verification (Hutter et al., 2007a), and even in areas far outside of computer science, such as water resource management (Tolson and Shoemaker, 2007).

In this thesis, we adopt a very general notion of what constitutes an *algorithm parameter*. This notion includes *numerical* parameters (*e.g.*, level of a real-valued threshold); *ordinal* parameters (*e.g.*, low, medium, high); and *categorical* parameters (*e.g.*, choice of heuristic), with the frequent special case of binary parameters (*e.g.*, algorithm component active/inactive). Note that categorical parameters can be used to select and combine discrete building blocks of an algorithm (e.g., preprocessing and variable ordering heuristics in a SAT solver). Consequently, our general view of algorithm configuration includes the automated construction of

[1] http://www.ilog.com/products/cplex/

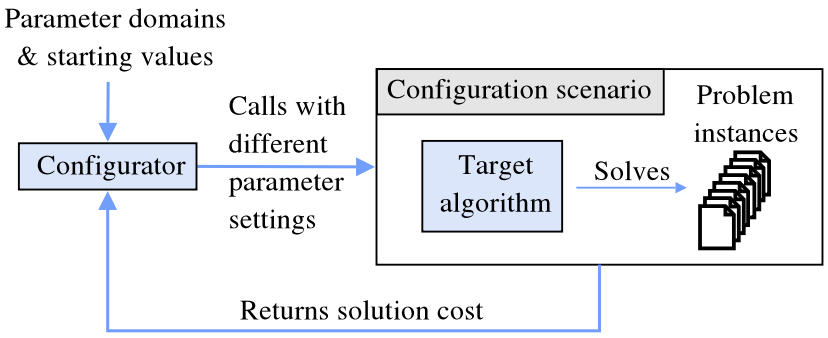


Figure 1.1: Visualization of algorithm configuration. A configuration procedure executes the target algorithm with specified parameter settings on one or more problem instances, receives information about algorithm performance, and uses this information to decide about target algorithm runs to perform subsequently. A configuration scenario includes the target algorithm to be configured and a collection of instances.

a heuristic algorithm from such building blocks. Thus, while other authors have referred to the optimization of an algorithm's performance by setting its (typically few and numerical) parameters as *parameter tuning*, we deliberately use the term *algorithm configuration* to emphasize our focus on configuring algorithms with potentially many, partially categorical algorithm parameters. (We discuss related work on parameter tuning in Chapter 2.)

To avoid potential confusion between algorithms whose performance is optimized and algorithms used for carrying out this optimization task, we refer to the former as *target algorithms* and the latter as *configuration procedures*. We refer to instances of the algorithm configuration problem as *configuration scenarios*. This setup is illustrated in Figure 1.1. Note that we treat algorithm configuration as a black-box optimization problem: our configuration procedures execute the target algorithm on a problem instance and receive feedback about the algorithm's performance; yet, they do not have access to any internal state of the algorithm (unless it is part of the optimization objective and encoded in the performance measure). This leads to a clean interface and considerably eases the application of our methods to the configuration of new algorithms. Different variants of parameter optimization problems have been considered in the literature, including setting parameters on a per-instance basis and modifying the parameters while the algorithm is running; we defer a discussion of these approaches to Chapter 2.

Whether manual or automated, effective configuration procedures are central in the development of heuristic algorithms for many classes of problems, particularly for $\mathcal{NP}$-hard problems. Algorithms with state-of-the-art empirical performance for such problems tend to be highly heuristic, and their behaviour and efficiency depends strongly and in complex ways on their parameters. They often combine a multitude of approaches and feature a correspondingly large and structured parameter space (see, *e.g.*, the aforementioned solver CPLEX or many state-of-the-art solvers for SAT). In this area, algorithm configuration is crucial, as the runtime of weak and strong algorithms regularly differs by several orders of magnitude on the

same problem instances. Existing theoretical techniques are typically not powerful enough to determine whether one parameter configuration is superior to another, and so the algorithm designer typically relies on manual empirical evaluation to determine the best configuration.

Hence, in this thesis, we propose automated methods for algorithm configuration. We hope that such methods will spare researchers and practitioners most of the countless (and often futile) hours spent manually experimenting with different heuristics and "tweaking parameters". As we will demonstrate in Chapters 6 and 8, these automated configuration procedures have already improved the performance of state-of-the-art algorithms for a variety of problems.

The concepts and approaches we introduce in this thesis are not restricted to the configuration of algorithms for solving hard combinatorial problems. Conversely, the issue of setting an algorithm's parameters to achieve good performance is almost omnipresent in algorithmic problem solutions. There already exists a host of domains for which automated configuration procedures have been applied successfully—we review these in Chapter 2. However, most existing work has applied methods geared specifically towards particular applications. We believe that more general freely-available algorithm configuration procedures could benefit research in most areas of computer science.

In the remainder of this chapter, we motivate the algorithm configuration problem further, define the problem formally, summarize the main contributions of our work, and outline the remainder of the thesis.

1.1 Motivation for Automated Algorithm Configuration

Our main motivation for developing generally-applicable automated procedures for algorithm configuration is (1) the potential to replace the most tedious and unrewarding part of traditional algorithm design with an automated process that leads to better results in a fraction of the time. In particular, automated configuration procedures pave the way for a paradigm of automated algorithm design from components. Automated algorithm configuration can also be used to (2) replace its manual counterpart on the end-user side of a complex algorithm, and (3) in order to enable fairer comparisons between algorithms.

In order to motivate the automated configuration of such algorithms, we first describe the traditional, manual approach to algorithm design (see also Hoos, 2008). We then show how this process can be improved by means of automated algorithm configuration.

1.1.1 Traditional, Manual Algorithm Design

State-of-the-art algorithms for solving computationally hard problems are traditionally constructed in an iterative, manual process in which the designer gradually introduces or modifies components or mechanisms. The performance resulting from these modifications is then evaluated empirically, typically based on rather small-scale experiments (small and fairly easy sets of benchmark problems, and comparably small cutoff times are typically used to limit the time required). Based on these experiments, some of the degrees of freedom—typically those the algorithm designer is most uncertain about—are exposed as parameters, while most others are hard-coded and usually never re-considered for modification.

When the iterative design process continues for an extended period of time the considered space of potential algorithm designs can rapidly grow unwieldy. This is particularly the case when the number of exposed parameters is not pruned to remain small throughout the process. An extreme example for this is the commercial optimization tool CPLEX, which—during its 20-year history—has exposed more and more parameters, leading to over 80 user-controllable parameters in version 10.1. Identifying the best design choices in such an immense design space requires substantial knowledge about the heuristic algorithm components and their interactions, which is far beyond what can be expected from a typical user. In fact, even the algorithm designers themselves often cannot solve this complex problem very well (Hutter et al., 2007a).

During the development of a typical high-performance algorithm there is only time to consider a few different algorithm designs (*i.e.*, combinations of parameter settings). This is because even small-scale empirical evaluations require a considerable amount of time, and experiments generally have to be launched manually. Instead of rigidly following an experimental design strategy (*e.g.*, outlined in Crary and Spera, 1996), the designs to be evaluated are typically chosen in an *ad hoc* fashion based on the intuition of the algorithm designer whenever a new feature is implemented. In particular, single design choices are often evaluated in isolation, with all other choices fixed, implicitly—and falsely—assuming that design choices do not interact.

Such experiments often lead algorithm designers to make unjustified generalizations, such as "algorithms using component A_1 perform better than algorithms using component A_2". If these components have only been empirically compared based on fixed instantiations of other design choices, (*e.g.*, B_2, C_1, D_3), then all that can be claimed is that, for the particular problem set used, A_1 performs better in combination with the remaining design choices (B_2, C_1, D_3) than does A_2. Since design choices often interact in complex and nonintuitive ways that are hard to grasp by human designers, the manual process is likely to miss the best combination of design choices (*e.g.*, A_2, B_2, C_3, D_2).

In contrast to humans, computer-aided mechanisms tend to perform well at such complex high-dimensional optimization tasks with many interdependent decisions. This very reason has brought success to the fields of operations research and constraint programming; the ability to efficiently handle complex high-dimensional data has also led to some of the most impressive success stories in artificial intelligence. As an analogy, consider Sudoku puzzles. While humans can in principle solve these, computer algorithms can do so much more efficiently. While Sudoku is a fun exercise for the brain, countless days of experimenting with different heuristics and parameter settings is *not*; quite the contrary, it holds researchers (often graduate students) back from spending their time with more meaningful problems (or at the beach, for that matter). More importantly, this manual experimentation is unlikely to fully realize the potential of a given algorithm. For this reason, we advocate the use of automated algorithm configuration tools whenever possible; in the next section, we outline a number of potential uses.

1.1.2 Automated Algorithm Design from Components

Efficient approaches for automated algorithm configuration pave the way for a paradigm of automated algorithm design from components. In this paradigm, the algorithm designer implements a high-level parameterized framework with interchangeable algorithm components, as well as a set of high-performance (parameterized) algorithm components that can instantiate the various parts of the framework. Then, he chooses a type of problem instances and a performance metric to be minimized (*e.g.*, cost, operational risk, or undesirable impacts on the environment) and applies an automated algorithm configuration procedure to search for the framework instantiation that empirically optimizes the metric on the chosen type of instances.

This semi-automated process has the advantage of a clean separation between *high-level cognitive tasks* performed by human designers and end users on the one hand and *tedious low-level tasks* that can be left to machines[2] on the other hand. High-level cognitive tasks include the conceptual design of the algorithm framework and the specification of which algorithm components should be considered (and which lower-level parameters should be exposed for tuning). Together, these high-level design choices comprise the design space that is later searched by automated procedures. Further high-level choices concern the selection of a problem distribution and a performance metric of interest. These can in fact be left to the *end user*: she simply calls an automated configuration procedure to search the design space (defined by the algorithm designer) in order to generate an algorithm optimizing the objective she specified. If her objective or instance distribution ever changes, she can simply repeat the automated procedure with a modified objective to generate an algorithm customized for the new problem.

Note that this process of automated algorithm design from components has in fact already been demonstrated to yield new state-of-the-art algorithms. In particular, we automatically constructed different instantiations of the SPEAR algorithm and thereby substantially improved the state of the art for two sets of SAT-encoded industrial verification problems (Hutter et al., 2007a). The SATENSTEIN framework of local search algorithms for SAT by KhudaBukhsh et al. (2009) took this process to the extreme, combining a multitude of components from various existing local search algorithms. Both SPEAR and SATENSTEIN were configured using ParamILS, one of the automated configuration procedures we introduce in this thesis (see Chapter 5). We provide full details on the configuration of SPEAR in Chapter 6 and review the SATENSTEIN application in Section 8.3.1. We hope that automated algorithm configuration procedures will become a mainstream technique in the design of algorithms; they have the potential to significantly speed up and improve this process.

1.1.3 Practical End Use of Algorithms

The end use of existing state-of-the-art algorithms is often complicated by the fact that these algorithms are highly parameterized. This is aggravated by the fact that the ability of complex heuristic algorithms to solve large and hard problem instances often depends critically on the use of parameter settings suitable for the particular type of problem instances. End users have

[2]For the time being, these tasks are addressed by human-designed automated approaches carried out on machines.

two options: to use the default parameter configuration and hope for the best or to invest time in exploring the space of possible parameter configurations.

End users often opt to use the default algorithm configuration, since they typically have little or no knowledge about the algorithm's parameter configuration space and little time to explore alternative parameter configurations. However, even if it has been carefully optimized on a standard benchmark set, a default configuration typically does not perform as well on the particular problem instances encountered by a user as a parameter configuration developed specifically with that type of instances in mind.

In situations where the default configuration does not yield satisfactory performance end users are forced to experiment with different parameters. For example, the ILOG CPLEX manual states this explicitly:

> "Integer programming problems are more sensitive to specific parameter settings, so you may need to experiment with them." (ILOG CPLEX 10.0 user manual, page 130)

Such experimentation requires expertise in the heuristic methods the algorithm is based on and experience in their empirical evaluation, which is not necessarily available on the end user's side. Most importantly, it is simply very time-consuming. As a consequence, shortcuts are often taken and suboptimal parameter settings are used, which can result in solutions far worse than possible. This leads to an opportunity cost in the metric being minimized.

1.1.4 Scientific Studies of Algorithm Performance

A central question in empirical comparisons of algorithms is whether one algorithm outperforms another one because it is fundamentally superior, or because its developers more successfully optimized its parameters (Johnson, 2002). In principle, similar efforts should be exerted for configuring (or tuning) all algorithms participating in a "horse-race" comparison. In practice, however, algorithm designers are most familiar with their own algorithm, say A, and often invest considerable manual effort to tune it. Even if a competitor algorithm, B, exposes a set of parameters and even if the designers of A invest the same amount of effort for tuning B as they did for A (both of which are desirable but not always true in practice), they can still be expected to do better at tuning A. This is because they are more familiar with A and have a better intuition about the importance of each of its parameters and about good ranges for continuous parameters.

In contrast, using a generally-applicable configuration procedure can mitigate this problem of unfair tuning and thus facilitate more meaningful comparative studies. One would simply run an automated configurator for a prespecified amount of time for each algorithm of interest and then compare the found instantiations of these algorithms against each other. In contrast to the manual configuration of each competitor algorithm, the use of an automated configuration method also is a well-defined and repeatable process.

However, we note that absolute fairness might be hard to achieve. One challenge is that the time allowed for configuration can have a strong influence on the outcome of comparisons. For example, think of a comparison between a robust (basically) parameter-less algorithm A and a heavily parameterized algorithm B that strongly depends on a configurator to instantiate it for

a given domain. It can be expected that for short configuration times A outperforms B whereas a specialized configuration of B found given a long enough configuration time outperforms A. Of course, different configuration procedures might also do better at configuring different algorithms.

A further potential use of algorithm configuration procedures is for identifying which types of parameter configurations do well for which types of instances. The model-based approaches we investigate in Part IV of this thesis show particular promise for linking the performance of algorithm components to characteristics of the instance at hand. Such insights may lead to new scientific discoveries and further improve algorithm performance for important subclasses of problems. They may also guide the (human part of the) algorithm design process.

On the theoretical side, the study of algorithm behaviour with different parameter configurations can yield important insights that may help characterize which core components of an algorithm are effective for solving various kinds of problem instances. Similar to lab experiments in the natural sciences, empirical studies can help build intuition and inspire new theoretical results, such as probabilistic algorithm guarantees (Hoos, 1999b). Furthermore, note that it is entirely possible to automatically configure algorithms with certain theoretical performance guarantees. This can be achieved by defining the parameter configuration space such that all allowed configurations have these guarantees (Hoos, 2008). Finally, if an automated method could be devised to check whether a certain theoretical property holds for a given configuration (or which out of a set of properties hold), the same methods we discuss in this thesis could be used to search for a configuration with the desirable properties.

1.2 Problem Definitions and Notation

The central topic of this thesis is the *algorithm configuration problem*. This problem can be informally stated as follows: given an algorithm, a set of parameters for the algorithm, and a set of input data, find parameter values under which the algorithm achieves the best possible performance on the input data.

Before we define this problem more formally, we first introduce some notation and relate the problem to standard blackbox function optimization problems. Let $\mathcal{A}$ denote an algorithm, and let $\theta_1, \ldots, \theta_k$ be parameters of $\mathcal{A}$. We denote the domain of possible values for each parameter θ_i as Θ_i; these domains can be infinite (as for continuous and unbounded integer parameters), finite and ordered (ordinal), or finite and unordered (categorical). We use $\boldsymbol{\Theta} \subseteq \Theta_1 \times \ldots \times \Theta_k$ to denote the space of all feasible parameter configurations, and $\mathcal{A}(\boldsymbol{\theta})$ to denote the instantiation of algorithm $\mathcal{A}$ with parameter configuration $\boldsymbol{\theta} \in \boldsymbol{\Theta}$. Let $\mathcal{D}$ denote a probability distribution over a space Π of problem instances, and denote an element of Π, *i.e.*, an individual problem instance, as π. $\mathcal{D}$ may be given implicitly, as through a random instance generator or a distribution over such generators. It is also possible (and indeed common) for Π to consist of a finite sample of instances. In this case, we define $\mathcal{D}$ as the uniform distribution over Π.

1.2.1 Blackbox Optimization

Algorithm configuration can be seen as a special type of *BlackBox Optimization (BBO)* problem. *Deterministic* BBO problems are problems of the form

$$\min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \quad f(\boldsymbol{\theta}),$$

where $f : \boldsymbol{\Theta} \to \mathbb{R}$ is a "blackbox function": we can query f at arbitrary inputs $\boldsymbol{\theta} \in \boldsymbol{\Theta}$, and the only information available about f are its function values at these queried points. In typical BBO problems, the optimization domain is continuous, that is, $\boldsymbol{\Theta} = \mathbb{R}^d$.

In *stochastic* BBO problems, the (deterministic) function f is replaced with a stochastic process $\{F_{\boldsymbol{\theta}} | \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$, a collection of random variables indexed by $\boldsymbol{\theta} \in \boldsymbol{\Theta}$. The goal in stochastic BBO is to find the configuration $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ that optimizes a given statistical parameter, τ (*e.g.*, the expected value or median), of $F_{\boldsymbol{\theta}}$'s distribution. Denoting the distribution of $F_{\boldsymbol{\theta}}$ as $\mathbb{P}_{\{\boldsymbol{\theta}\}}$, stochastic BBO problems are thus of the form

$$\min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \quad \tau(\mathbb{P}_{\{\boldsymbol{\theta}\}}).$$

For example, in algorithm configuration, $\mathbb{P}_{\{\boldsymbol{\theta}\}}$ might be the runtime distribution of a randomized algorithm with parameter configuration $\boldsymbol{\theta}$, and τ the expected value. We refer to $\mathbb{P}_{\{\boldsymbol{\theta}\}}$ as the configuration's *cost distribution*, to observed samples from that distribution as single *observed costs*, o_i, and to the statistical parameter, τ, of that cost distribution as *overall cost*, $c(\boldsymbol{\theta})$:

$$c(\boldsymbol{\theta}) := \tau(\mathbb{P}_{\{\boldsymbol{\theta}\}}).$$

Since objectives differ between applications, we leave the definition of the overall cost, $c(\boldsymbol{\theta})$, up to the user. For example, we might aim to minimize expected runtime or median solution cost. In this thesis, we predominantly minimize mean runtime (penalizing timed-out runs as discussed in Section 3.4).

1.2.2 Algorithm Configuration

Algorithm configuration for deterministic algorithms and single instances can be seen as a deterministic BBO problem, whereas the configuration of randomized algorithms or the configuration on distributions with infinite support Π would be a stochastic BBO problem. Since algorithm parameters are often discrete, in algorithm configuration we usually have $\boldsymbol{\Theta} \neq \mathbb{R}^d$.

Another difference between algorithm configuration and (standard) BBO problems lies in the fact that the cost distribution, $\mathbb{P}_{\{\boldsymbol{\theta}\}}$, of a parameter configuration, $\boldsymbol{\theta}$, contains structure. For a fixed combination of a parameter configuration, $\boldsymbol{\theta}$, an instance, π, and a pseudorandom number seed, s, the observed cost, $o(\boldsymbol{\theta}, \pi, s)$, is deterministic.[3] Distribution $\mathbb{P}_{\{\boldsymbol{\theta}\}}$ is induced

[3]This discussion relies on the fact that randomized algorithms typically take a seed for a pseudorandom number generator as an input. Note that this is an artifact of the way random decisions are implemented in

by sampling instances π from distribution $\mathcal{D}$ and seeds from a uniform distribution, $\mathcal{S}$, over allowable seeds. The structure in this distribution can, indeed, be exploited. For example, knowing the runtime of configuration $\boldsymbol{\theta}_1$ on an instance can yield important information on the runtime of configuration $\boldsymbol{\theta}_2$ on the same instance. This is even the case for random seeds. Think of two configurations $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ that only differ in a parameter the algorithm never uses; given $o(\boldsymbol{\theta}_1, \pi, s_1)$, we perfectly know $o(\boldsymbol{\theta}_2, \pi, s_1)$, but not $o(\boldsymbol{\theta}_2, \pi, s_2)$. We can, for example, exploit such dependence by performing blocked comparisons (see Section 3.6.1).

Finally, we have an additional degree of freedom in algorithm configuration: at which *cutoff time*, κ, to terminate a run that is unsuccessful within time κ. In standard BBO problems, each function evaluation is assumed to take the same amount of time. In contrast, in algorithm configuration, runtimes with different configurations can vary substantially. For example, when minimizing algorithm runtime to solve a given problem, runtimes can differ by many orders of magnitude across parameter configurations. In order to save time spent in long runs for poor parameter configurations, we can limit the time for each run by a cutoff time, κ.

With this intuition in mind, we now define the algorithm configuration problem formally.

Definition 1 (Algorithm Configuration Problem)**.** *An instance of the* algorithm configuration *problem is a 6-tuple $\langle \mathcal{A}, \boldsymbol{\Theta}, \mathcal{D}, \kappa_{max}, o, \tau \rangle$, where:*

- *$\mathcal{A}$ is a parameterized algorithm;*
- *$\boldsymbol{\Theta}$ is the parameter configuration space of $\mathcal{A}$;*
- *$\mathcal{D}$ is a distribution over problem instances with some domain Π;*
- *κ_{max} is a cutoff time (or* captime*), after which each run of $\mathcal{A}$ will be terminated if still running;*
- *o is a function that measures the observed cost of running $\mathcal{A}(\boldsymbol{\theta})$ on instance $\pi \in \Pi$ with captime $\kappa \in \mathbb{R}$; and*
- *τ is a statistical population parameter to be optimized.*

Examples for o are runtime for solving the instance, or quality of the solution found within the captime; in the former case, o must also define a cost for runs that do not complete within the captime. Throughout most of this thesis, we aim to minimize the penalized average runtime of an algorithm on a distribution of instances, where timeouts after κ_{max} are counted as $a \cdot \kappa_{max}$ with $a \geq 1$, and τ is the expected value (see Section 3.4 for a discussion of that choice of optimization objective).

Our problem formulation also allows us to express conditional parameter dependencies. For example, one algorithm parameter, θ_1, might be used to select among several search components, with component i's behaviour controlled by further parameters.The values of those further parameters are irrelevant unless $\theta_1 = i$; We thus call such parameters *conditional* on the higher-level parameter θ_1. The configuration framework we introduce in Part III of this thesis exploits this and effectively searches the space of equivalence classes in parameter

current algorithms. We can exploit this artifact by making informed decisions about which seeds to employ (as for example in blocking), but our approaches also work for target algorithms that do not rely on seeds to implement randomness.

configuration space. In addition, our formulation supports the specification of infeasible combinations of parameter values, which are excluded from $\boldsymbol{\Theta}$.

Any parameter configuration $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ is a candidate solution of the algorithm configuration problem. An *algorithm configuration procedure* (short: *configuration procedure* or *configurator*), is a procedure for solving the algorithm configuration problem. For each configuration, $\boldsymbol{\theta}$, $\mathbb{P}_{\{\boldsymbol{\theta}\}}$ denotes the *cost distribution* induced by function o, applied to instances π drawn from distribution $\mathcal{D}$ and multiple independent runs for randomized algorithms, using captime κ_{max}. As in the case of stochastic BBO problems, the *cost* of a candidate solution $\boldsymbol{\theta}$ is defined as

$$c(\boldsymbol{\theta}) := \tau(\mathbb{P}_{\{\boldsymbol{\theta}\}}), \tag{1.1}$$

the statistical population parameter, τ, of the cost distribution. An optimal solution, $\boldsymbol{\theta}^*$, minimizes $c(\boldsymbol{\theta})$, *i.e.*,

$$\boldsymbol{\theta}^* \in \operatorname*{argmin}_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} c(\boldsymbol{\theta}). \tag{1.2}$$

We cannot optimize c directly, since we cannot write this function analytically. (In some special cases, writing the function may actually be possible; here, we use a blackbox formalization for generality.) Instead, we must execute a sequence of runs $\mathbf{R}$ of the target algorithm $\mathcal{A}$ with different parameter configurations, derive empirical estimates of c's values at particular points in configuration space, and use them to identify a configuration with low cost.

We denote the sequence of runs executed by a configurator as $\mathbf{R} = ((\boldsymbol{\theta}_1, \pi_1, s_1, \kappa_1, o_1), \ldots, (\boldsymbol{\theta}_n, \pi_n, s_n, \kappa_n, o_n))$. The ith run is described by five values:

- $\boldsymbol{\theta}_i \in \boldsymbol{\Theta}$ denotes the parameter configuration being evaluated;
- $\pi_i \in \Pi$ denotes the instance on which the algorithm is run;
- s_i denotes the random number seed used in the run (a constant for algorithms that do not accept seeds);
- $\kappa_i \in \mathbb{R}$ denotes the run's captime; and
- o_i denotes the observed cost of the run.

Note that each of $\boldsymbol{\theta}$, π, s, κ, and o can vary from one element of $\mathbf{R}$ to the next, regardless of whether or not other elements are held constant. This is in particular the case for κ: we are free to terminate algorithm runs after any captime $\kappa \leq \kappa_{max}$, but the eventual cost of a configuration is always computed using the "full" captime, κ_{max}. Also note that $\mathbf{R}$ is typically constructed sequentially. We denote the ith run of $\mathbf{R}$ as $\mathbf{R}[i]$, and the subsequence of runs using parameter configuration $\boldsymbol{\theta}$ (*i.e.*, those runs with $\boldsymbol{\theta}_i = \boldsymbol{\theta}$) as $\mathbf{R}_{\boldsymbol{\theta}}$.

We use such sequences of runs, $\mathbf{R}$, in order to estimate the cost, $c(\boldsymbol{\theta})$, of a parameter configuration $\boldsymbol{\theta}$, both online, during runtime of a configurator, as well as offline, for evaluation purposes. We now introduce the notion of an empirical cost estimate.

Definition 2 (Empirical Cost Estimate)**.** *Given an algorithm configuration problem instance* $\langle \mathcal{A}, \boldsymbol{\Theta}, \mathcal{D}, \kappa_{max}, o, \tau \rangle$*, an* empirical cost estimate *of cost* $c(\boldsymbol{\theta})$ *based on a sequence of runs* $\mathbf{R} = ((\boldsymbol{\theta}_1, \pi_1, s_1, \kappa_1, o_1), \ldots, (\boldsymbol{\theta}_n, \pi_n, s_n, \kappa_n, o_n))$ *is defined as* $\hat{c}(\boldsymbol{\theta}, \mathbf{R}) := \hat{\tau}(\{o_i \mid \boldsymbol{\theta}_i = \boldsymbol{\theta}\})$*, where* $\hat{\tau}$ *is the sample statistic analogue to the statistical population parameter* τ*.*

For example, when $c(\boldsymbol{\theta})$ is mean runtime over a distribution of instances and random number seeds, $\hat{c}(\boldsymbol{\theta}, \mathbf{R})$ is the sample mean runtime of runs $\mathbf{R}_{\boldsymbol{\theta}}$. We often omit $\mathbf{R}$ for brevity of notation and write $\hat{c}_N(\boldsymbol{\theta})$ to emphasize that estimates are based on N runs, that is, $|\mathbf{R}_{\boldsymbol{\theta}}| = N$.

All configuration procedures studied in this thesis are anytime algorithms in the sense that at all times they keep track of the configuration currently believed to have the lowest cost. We refer to this configuration as the *incumbent configuration*, or, short, the *incumbent*, $\boldsymbol{\theta}_{inc}$. We evaluate a configurator's performance at time t by means of its incumbent's training and test performance, defined as follows.

Definition 3 (Training performance)**.** *When at some time t a configurator has performed a sequence of runs $\mathbf{R} = ((\boldsymbol{\theta}_1, \pi_1, s_1, \kappa_1, o_1), \ldots, (\boldsymbol{\theta}_t, \pi_t, s_t, \kappa_t, o_t))$ to solve an algorithm configuration problem instance $\langle \mathcal{A}, \boldsymbol{\Theta}, \mathcal{D}, \kappa_{max}, o, \tau \rangle$, and has thereby found an incumbent configuration $\boldsymbol{\theta}_{inc}(t)$, then its* training performance *at time t is defined as the empirical cost estimate $\hat{c}(\boldsymbol{\theta}_{inc}(t), \mathbf{R})$. We denote training performance at time t as $p_{t,train}$.*

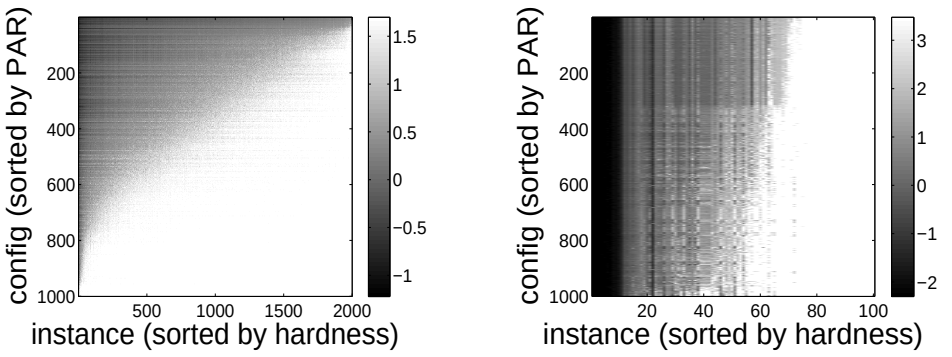
The set of instances $\{\pi_1, \ldots, \pi_t\}$ discussed above is called the *training set*. While the true cost of a parameter configuration cannot be computed exactly, it can be estimated using training performance. However, the training performance of a configurator is a biased estimator of its incumbent's true cost, because the same instances are used for selecting the incumbent as for evaluating it (see, *e.g.*, Birattari, 2005). In order to achieve unbiased estimates during offline evaluation, we set aside a fixed set of instances $\{\pi'_1, \ldots, \pi'_T\}$ (called the *test set*) and random seeds $\{s'_1, \ldots, s'_T\}$, both unknown to the configurator, and use these for evaluation. Note that test set and training set are disjoint.

Definition 4 (Test performance)**.** *At some time t, let a configurator's incumbent for an algorithm configuration problem instance $\langle \mathcal{A}, \boldsymbol{\Theta}, \mathcal{D}, \kappa_{max}, o, \tau \rangle$ be $\boldsymbol{\theta}_{inc}(t)$ (this is found by means of executing a sequence of runs on the training set). Furthermore, let $\mathbf{R}_{test} = ((\boldsymbol{\theta}_{inc}(t), \pi'_1, s'_1, \kappa_{max}, o_1), \ldots, (\boldsymbol{\theta}_{inc}(t), \pi'_T, s'_T, \kappa_{max}, o_T))$ be a sequence of runs on the T instances and random number seeds in the test set (which is performed offline for evaluation purposes), then the configurator's* test performance *at time t is defined as the empirical cost estimate $\hat{c}(\boldsymbol{\theta}_{inc}(t), \mathbf{R}_{test})$. We denote test performance at time t as $p_{t,test}$, or simply as p_t.*

Finally, we evaluate predictions, $\hat{\mu(\boldsymbol{\theta})}$, of a parameter configuration's cost measure, $c(\boldsymbol{\theta})$, by comparing them to $\boldsymbol{\theta}$'s empirical cost estimate on the entire training set. Since this estimate is computed offline for validation purposes, we refer to it as *validation cost*, $c_{valid}(\boldsymbol{\theta})$.

Definition 5 (Validation cost)**.** *For an algorithm configuration problem instance $\langle \mathcal{A}, \boldsymbol{\Theta}, \mathcal{D}, \kappa_{max}, o, \tau \rangle$ and a parameter configuration, $\boldsymbol{\theta} \in \boldsymbol{\Theta}$, let $\mathbf{R}_{valid} = ((\boldsymbol{\theta}, \pi'_1, s'_1, \kappa_{max}, o_1), \ldots, (\boldsymbol{\theta}, \pi'_T, s'_T, \kappa_{max}, o_T))$ be a sequence of runs on the T instances and random number seeds in the training set (which is performed offline for evaluation purposes). Then, the* validation cost, *$c_{valid}(\boldsymbol{\theta})$, of configuration $\boldsymbol{\theta}$ is defined as the empirical cost estimate $\hat{c}(\boldsymbol{\theta}, \mathbf{R}_{valid})$.*

To provide more intuition for the algorithm configuration problem, we visualize the joint space of configurations and instances faced by configuration procedures. For finite configuration spaces, $\boldsymbol{\Theta}$, and training sets, Π, we can imagine the algorithm configuration problem as a matrix of size $|\boldsymbol{\Theta}| \times |\Pi|$, where entry (i, j) contains the (deterministic) cost



(a) CPLEX-REGIONS100 (b) SPEAR-IBM

Figure 1.2: Visualization of algorithm configuration: matrix of runtimes for $M = 1\,000$ configurations and all training instances, Π. Each dot in the matrix represents the runtime of a configuration on a single instance: darker dots represent shorter runtimes; note that the scale is logarithmic with base 10. Configurations are sorted by penalized average runtime (PAR, see Section 3.4) on the training set; instances are sorted by hardness (mean runtime of the $|\boldsymbol{\Theta}|$ configurations, analogously to PAR counting runs that timed out at captime κ_{max} as $10 \cdot \kappa$).

(in this thesis typically runtime) of $\mathcal{A}(\boldsymbol{\theta}_i)$ using seed s_j on instance π_j. For visualization purposes, we randomly sample a subset of parameter configurations $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ and sort them by their empirical cost estimate $\hat{c}_{|\Pi|}$ across the training set. Likewise, we sort instances by their hardness across configurations. Figure 1.2 shows two configuration scenarios; the target algorithms and instance sets used in these scenarios are described in detail in Chapter 3. Briefly, in CPLEX-REGIONS100 (see Figure 1.2(a)) we optimize CPLEX for a homogeneous set of MIP-encoded winner determination problems in combinatorial auctions, and in SPEAR-IBM (see Figure 1.2(b)), we optimize the tree search SAT solver SPEAR for a set of industrial bounded model-checking instances.

This visualization provides some intuition about how different actual instances of the algorithm configuration problem can be. For example, in scenario CPLEX-REGIONS100, the difference between the worst and the best configurations was much higher than for SPEAR-IBM: the worst sampled configuration did not solve any instance within $\kappa_{max} = 5$ seconds whereas the best configuration solved all. In contrast, in scenario SPEAR-IBM every sampled configuration solved the easiest 50% of the instances and no configuration solved the hardest 30%.

Intuitively, in order to find a configuration that minimizes $c(\boldsymbol{\theta})$, algorithm configuration procedures sequentially execute runs of the target algorithm, thereby filling in entries of the runtime matrix (only filling in lower bounds in cases where κ is chosen $< \kappa_{max}$ and the run times out). It is not obvious how an automatic algorithm configurator should choose which runs to perform. Filling in the whole matrix is computationally prohibitive: gathering the data for the visualization in Figures 1.2(a) and 1.2(b) took 1.5 and 2.5 CPU months, respectively. This is despite the fact that we only used a small fraction of $1\,000$ randomly-sampled configuration

from a very large configuration space, $\mathbf{\Theta}$ ($|\mathbf{\Theta}| = 1.38 \cdot 10^{37}$ for CPLEX and $|\mathbf{\Theta}| = 8.34 \cdot 10^{17}$ for SPEAR). In order to best minimize $c(\boldsymbol{\theta})$ within a given time budget, we thus have to make the following choices:

1. Which sequential search strategy should be used in order to select a sequence of parameter configurations $\vec{\mathbf{\Theta}}$ to be evaluated?
2. Which problem instances $\Pi_{\boldsymbol{\theta}'} \subseteq \Pi$ should be used for evaluating each element of $\vec{\mathbf{\Theta}}$, and how many runs should be performed for it?
3. Which cutoff time κ should be used for each run?

In combination, these choices define the design space for algorithm configuration procedures. We therefore refer to them as the *dimensions of algorithm configuration*. It is not straight-forward to make the right choices for these dimensions; in essence, the topic of this thesis is to determine choices that result in effective algorithm configuration procedures. As we will show throughout, using our configuration procedures we found much better configurations than the simple random sampling approach, in a fraction of the runtime.

1.3 Summary of Contributions

The main contribution of this thesis is a comprehensive study of the algorithm configuration problem. This includes an empirical analysis approach to study the characteristics of configuration scenarios, two fundamentally different search frameworks to instantiate the first dimension of algorithm configuration, various adaptive mechanisms for the second and third dimensions, and a demonstration of algorithm configuration's practical relevance. Here, we describe these in somewhat more detail.

- We introduce and experimentally compare two fundamentally different frameworks (model-free and model-based search) for the first dimension of algorithm configuration, the search strategy (Parts III and IV). To the best of our knowledge, our model-free search mechanism is the first that goes beyond local optima. Our model-based search techniques significantly outperform existing techniques and are substantially more widely applicable.

- We introduce the first algorithm configuration procedures capable of configuring algorithms with many categorical parameters, such as CPLEX (63 parameters), SPEAR (26 parameters) and SATENSTEIN (41 parameters) (Chapters 5 and 12). Our first method, PARAMILS (introduced in Chapter 5), is already in use by other research groups to configure their algorithms for solving hard combinatorial problems.

- We demonstrate the practical relevance of automated algorithm configuration by using it to substantially advance the state of the art for solving important types of problem instances. In particular, we configure the tree search SAT algorithm SPEAR, yielding 4.5-fold and over 500-fold speedups for SAT-encoded industrial hardware and software verification, respectively (Chapter 6). This amounts to a substantial improvement of the

state of the art for solving these types of instances. Based on the automated configuration of the commercial optimization tool CPLEX, we also achieve up to 23-fold speedups of the state of the art for MIP-encoded problem distributions (Section 8.1). Finally, by configuring GLS$^+$, we improve the state of the art for solving the MPE problem in certain types of Bayesian networks by more than a factor of 360 (Section 5.4).

- We empirically analyze the tradeoff between the second and third dimensions of algorithm configuration—how many runs to perform for evaluating each configuration and how to set the per-run captime (Chapter 4). We demonstrate that there exists no optimal fixed choice. Thus, we introduce data-driven approaches for making these choices adaptively. We present three new methods for adaptively selecting the number of runs to perform for each configuration (Sections 5.3, 10.3.1, and 13.6.1). We also introduce a general method for adaptively setting the captime—to the best of our knowledge, the first of its kind (Chapter 7).

- Overall, we offer a thorough and systematic study of algorithm configuration and provide reliable and scalable configuration procedures that can strongly improve algorithm performance with minimal manual user effort.

A further contribution stems from the improvements and extensions of model-based sequential optimization procedures.

- We substantially extend the applicability of model-based search techniques to handle categorical parameters (Chapter 12) and optimization objectives defined across multiple problem instances (Chapter 13). Furthermore—in contrast to previous methods—our techniques scale to configuration scenarios that require tens of thousands of target algorithm runs (Sections 11.3 and 11.4) and smoothly handle non-Gaussian, non-stationary observation noise (Section 11.2.1).

1.4 How to Read this Thesis

This thesis provides a comprehensive account of algorithm configuration, organized in a hierarchical fashion (see next section for the outline). The interested but time-constrained reader can find chapter summaries at the end of all chapters (except the last) that provide a high-level overview of the material and are designed to be self-contained. Chapters 2 (Related Work) and 3 (Configuration Scenarios) can largely serve as reference chapters, while Chapter 4 builds intuition that is likely of more immediate benefit. Parts III (Chapters 5-8) and IV (Chapters 9-13) are largely decoupled. Part III is likely of more immediate interest to practitioners looking to use algorithm configuration: it covers a simple algorithm configuration framework and its various existing practical applications. In contrast, Part IV is likely of more interest to researchers with a background in blackbox optimization or in statistics/machine learning. It builds on sequential experimental design methods from the statistics literature, improving and extending them to create a new framework for general algorithm configuration based on response surface models. Chapter 14 (Conclusion) puts these two different frameworks into

perspective; it is somewhat self-contained but points to specific pieces of the thesis to make its points.

1.5 Thesis Outline

This thesis is organized in five parts. Part I introduces and motivates the algorithm configuration problem (this current chapter) and discusses related work (Chapter 2).

In Part II, we introduce our sets of algorithm configuration scenarios, which provide the testing grounds for the configuration procedures we discuss in this thesis. In Chapter 3, we describe the target algorithms, benchmark instances, and optimization objectives used. In Chapter 4, we present the empirical analysis techniques that we used to gain insight into the characteristics of a given algorithm configuration scenario. In this analysis, we focus on the intricate tradeoffs between the number of parameter configurations we are able to evaluate, the (fixed) number of instances used for each evaluation and the (fixed) captime after which to terminate unsuccessful algorithm runs. We show that there exists no optimal fixed tradeoff: the best number of instances and captime to use differs across algorithm configuration scenarios.

In Part III, we discuss our first set of algorithm configuration procedures, as well as practical applications of it. This part focuses on *model-free* search for algorithm configuration, that is, approaches that directly search the space of parameter configurations without relying on a predictive model of performance to guide the search.

In Chapter 5, we present a framework for iterated local search (ILS) for algorithm configuration called PARAMILS. ILS is one possible choice for the first dimension of algorithm configuration (which parameter configurations to evaluate). Our FOCUSEDILS algorithm, one instantiation of the PARAMILS framework, also implements an adaptive mechanism for making the second fundamental choice of algorithm configuration (how many algorithm runs to execute for each parameter configuration).

In Chapter 6, we report a "real-life" application study of PARAMILS. In this case study, we automatically configured the 26 parameters of the tree search SAT solver SPEAR for large industrial bounded model checking and software verification instances, achieving 4.5-fold and 500-fold speedups in mean runtime over the manually-engineered default, respectively.

In Chapter 7, we introduce a novel *adaptive capping* scheme that determines automatically when poor algorithm runs can be terminated early, thereby providing the first adaptive mechanism to instantiate the third and last dimension of algorithm configuration (which captime should be used for each run).

Finally, in Chapter 8, we report further successful "real-life" applications of PARAMILS, most notably our application to automatically setting the 63 parameters of CPLEX, the most complex and heavily parameterized algorithm of which we are aware. Even without using any explicit domain knowledge, we were able to achieve substantial speedups over the well-engineered CPLEX default parameter configuration, in some domains exceeding a factor of ten. We also review other applications of PARAMILS, including an application to automated algorithm design from parameterized components.

In Part IV, we explore a fundamentally different choice for the first dimension of algorithm configuration, the search strategy. In particular, we study *model-based* approaches, which utilize predictive models of algorithm performance (so-called response surface models) in order to guide the search for optimal configurations. This approach was much less developed as a tool for general algorithm configuration, and thus our contribution in this part is mostly methodologically; nevertheless, after five chapters of improvements and extensions of existing procedures, we end up with a state-of-the-art procedure for general algorithm configuration.

In Chapter 9, we start this part with an introduction to sequential model-based optimization (SMBO) for general blackbox function optimization. We introduce a general framework for SMBO and describe two existing state-of-the-art approaches, SPO and SKO, as instantiations of this framework. We then experimentally compare the two, showing that for some configuration scenarios SPO is more robust "out-of-the-box".

We substantially improve upon these approaches in Chapters 10 and 11. In Chapter 10, we experimentally investigate the components of SPO, and identify and resolve a number of weaknesses. Most notably, we introduce a novel intensification mechanism, and show that log transformations of cost statistics can substantially improve the quality of the response surface model. This mechanism leads to substantially improved robustness of our resulting configuration procedure, dubbed SPO$^+$, as compared to SPO. To reduce computational overheads due to the model construction, we also introduce a modification of SPO$^+$, dubbed SPO*.

In Chapter 11, we consider the use of different response surface models in SMBO, which are based on random forests and approximations of Gaussian processes. We demonstrate qualitatively that these models can improve predictive quality under various noise models as well as the performance of SMBO. Quantitatively, we demonstrate that our new models lead to substantially improved predictions. Furthermore, the computational complexity for their construction is substantially lower than that of traditionally-used models. Our computational experiments demonstrate that this leads to speedups of orders of magnitude. Based on these models, we introduce a new family of configuration procedures—dubbed ACTIVECONFIGURATOR—which we demonstrate to show significantly better performance than SPO*.

Next, we substantially expand the scope of SMBO in ways crucial for its use in algorithm configuration. While, prior to our work, existing SMBO methods have been limited to the optimization of numerical parameters, in Chapter 12 we extend its scope to include the optimization of categorical parameters. In particular, we extend our response surface models to handle categorical inputs and introduce a simple local search mechanism to search for the most promising parameter configuration to try next. We also compare instantiations of our ACTIVECONFIGURATOR and PARAMILS frameworks for configuration scenarios with single instances and show that ACTIVECONFIGURATOR yields state-of-the-art performance.

In Chapter 13, we extend SMBO to handle general algorithm configuration problems defined across distributions (or sets) of problem instances. We integrate problem instance features into our response surface model, enabling them to predict the runtime for combinations of instances and parameter configurations. We discuss different sets of instance features and experimentally compare models based on them. Overall, we find that in some cases predictive

performance is excellent, yielding rank correlations between predicted and actual runtimes of above 0.95. We then compare ACTIVECONFIGURATOR variants based on the different sets of features, and demonstrate that they achieve state-of-the-art performance in a variety of configuration scenarios.

Finally, in Part V, we offer conclusions about algorithm configuration in general and about our methods in particular, and discuss various promising avenues for future work.

1.6 Chapter Summary

In this chapter, we motivated and formally defined the *algorithm configuration problem*, the problem central to this thesis. The algorithm configuration problem can be informally stated as follows: given an algorithm, a set of parameters for the algorithm, and a set of input data, find parameter values under which the algorithm achieves the best possible performance on the input data.

An automated method for solving this problem (a so-called *algorithm configuration procedure*) can be used to replace the most tedious and unrewarding part of traditional algorithm design. The resulting automated process may only require a fraction of the time human experts need to establish good parameter settings while yielding better results. Automated configuration procedures (short: *configuration procedures* or *configurators*) not only facilitate algorithm development, they can also be applied on the end user side to optimize performance for new instance types and optimization objectives. The use of such procedures separates *high-level cognitive tasks* carried out by humans from *tedious low-level tasks* that can be left to machines.

Toward the end of the chapter, we also summarized the contributions of this thesis, provided a guideline on how to read the thesis, and gave an outline of the remainder of the thesis.

Chapter 2

Related Work

If I have seen further than others, it is by standing upon the shoulders of giants.
—Sir Isaac Newton, English physicist and astronomer

Many researchers before us have been dissatisfied with the manual optimization of algorithm parameters, and various fields have developed their own approaches automating this task. In particular, research communities that have contributed techniques for algorithm configuration or parameter tuning include planning (Gratch and Dejong, 1992), evolutionary computation (Bartz-Beielstein, 2006), meta-heuristics (Birattari, 2005), genetic algorithms (Fukunaga, 2008), parallel computing (Brewer, 1995), and numerical optimization (Audet and Orban, 2006). In this review of previous work, we focus on the proposed *techniques*, only mentioning application areas in passing. However, in the end of the chapter (in Section 2.4), we discuss some additional promising application areas.

We broadly categorize approaches related to our work into three categories, discussed in the first three sections of this Chapter: (1) approaches that optimize an algorithm's fixed set of parameters; (2) approaches for the automated construction of algorithms; and (3) approaches that deal with related problems (such as per-instance approaches, algorithm portfolios, and online approaches).

2.1 Algorithm Configuration and Parameter Optimization

In this section, we present previous algorithm configuration and parameter optimization methods based on direct search, model-based search and racing.

2.1.1 Direct Search Methods

Approaches for automated algorithm configuration go back at least to the early 1990s, when a number of systems were developed for what Gratch and Chien (1996) called *adaptive problem solving*. Gratch and Dejong (1992)'s *Composer* system performs a hill-climbing search in configuration space, taking moves if enough evidence has been gathered to render a neighbouring configuration statistically significantly better than the current configuration. Gratch and Chien (1996) successfully applied Composer to improving the five parameters of

an algorithm for scheduling communication between a collection of ground-based antennas and spacecraft in deep space.

Around the same time, Minton (1993, 1996) developed the *MULTI-TAC* system. MULTI-TAC takes as input a number of generic heuristics as well as a specific problem domain and a distribution over problem instances. It adapts the generic heuristics to the problem domain and automatically generates domain-specific LISP programs implementing them. In order to choose the best program, it uses a beam search, evaluating each program by running it on a number of problem instances sampled from the given distribution.

Terashima-Marín et al. (1999) introduced a genetic algorithm for configuring a constraint satisfaction algorithm for large-scale university exam scheduling. They constructed and configured an algorithm that works in two stages and has seven configurable categorical parameters. They optimized these choices with a genetic algorithm for each of 12 problem instances, and for each of them found a configuration that improved performance over a modified Brelaz algorithm. However, note that they performed this optimization separately for each instance. Their paper did not quantify how long these optimizations took, but stated that "Issues about the time for delivering solutions with this method are still a matter of research".

Muja and Lowe (2009) applied a custom algorithm configuration procedure to the problem of identifying the best-performing approximate nearest neighbour algorithm for a specific type of data set. They select between multiple randomized kd-trees and a new version of hierarchical k-means trees, and also select the best parameter setting for the algorithm. Their automated configuration procedure first evaluates multiple parameter configurations; it then employs the Nelder-Mead downhill simplex procedure initialized at the best of these to locally optimize numerical parameters. Evaluations are based on cross-validation using a fixed-size data set, typically only one tenth of the available dataset to reduce the computational burden. The reported experiments show a roughly ten-fold speedup compared to the best previously-available software for approximate nearest neighbour matching.

A multitude of direct search algorithms exist for the optimization of numerical parameters. Here, we discuss only the ones that have been used to optimize algorithm performance.

Coy et al. (2001) presented another search-based approach that uses a fixed training set. Their approach works in two stages. First, it finds a good parameter configuration $\boldsymbol{\theta}_i$ for each instance I_i in the training set by a combination of experimental design (full factorial or fractional factorial) and gradient descent. Next, it combines the parameter configurations $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_N$ thus determined by setting each parameter to the average of the values taken in all of them. This averaging step restricts the applicability of the method to algorithms with exclusively numerical parameters.

A similar approach, also based on a combination of experimental design and gradient descent, using a fixed training set for evaluation, is implemented in the CALIBRA system of Adenso-Diaz and Laguna (2006). CALIBRA starts by evaluating each parameter configuration in a full factorial design with two values per parameter. It then iteratively homes in to good regions of parameter configuration space by employing fractional experimental designs that evaluate nine configurations around the best performing configuration found so far. The grid for the experimental design is refined in each iteration. Once a local optimum is found, the search is restarted (with a coarser grid). Experiments showed CALIBRA's ability to

find parameter settings for six target algorithms that matched or outperformed the respective originally-proposed parameter configurations. Its main drawback is the limitation to tuning numerical and ordinal parameters, and to a maximum of five parameters.

Tolson and Shoemaker (2007) presented a simple global optimization algorithm.

Their algorithm, called *dynamically dimensioned search (DDS)*, can be interpreted as a variable-neighbourhood local search mechanism in continuous space. Limited to tuning continuous parameters only, DDS starts from an initial solution and then iterates the following steps: (1) randomly select a subset of parameters; (2) perturb the values of the selected parameters by sampling a new value for each selected parameter θ_i from a Gaussian distribution $\mathcal{N}(\theta_i, 0.2 \cdot r_i)$, where θ_i is the parameter's current value and r_i is the parameter's range; and (3) accept the perturbed solution if it yields a performance improvement. Related to simulated annealing, the probability for selecting parameters to be perturbed shrinks over time. Experiments for the automatic calibration of watershed simulation models with 6 to 30 dimensions showed that some parameterizations of DDS could outperform a competitor, the shuffled complex evolution algorithm.

Work on automated parameter tuning can also be found in the numerical optimization literature. In particular, Audet and Orban (2006) proposed the *mesh adaptive direct search (MADS)* algorithm. Designed for purely continuous parameter configuration spaces, MADS is guaranteed to converge to a *local* optimum of the cost function. The optimization objective in their work was the runtime and number of function evaluations required by interior point methods for solving a set of large unconstrained regular problems from the CUTEr collection (Gould et al., 2004). To reduce training cost, MADS used a set of smaller problems from the same collection to evaluate training performance. Although this bears the risk of improving performance for small problems but worsening performance for large problems, the experiments reported by Audet and Orban (2006) demonstrated performance improvements of around 25% over the classical configuration of four continuous parameters of interior point methods. Audet and Orban (2006) mentioned very small amounts of noise (around 1% variation between evaluations of the same parameter configuration), but did not specifically address them by stochastic optimization techniques.

As discussed in Section 1.2.2, algorithm configuration can be seen as a stochastic optimization problem, and there exists a large body of algorithms designed for such problems (see, *e.g.*, Spall, 1999, 2003). However, many of the algorithms in the stochastic optimization literature require explicit gradient information and are thus inapplicable to algorithm configuration. Often, one can simply approximate gradients by finite differences, but this typically increases the number of required function evaluations substantially. Since algorithms with purely continuous parameter configuration spaces can be expected to have very few local minima, this approach is promising for tuning such algorithms. For objective functions with many local minima, simulated annealing is a frequently-used technique in stochastic optimization. However, in the context of optimizing functions with many categorical parameters, simulated annealing often yields suboptimal results (Hoos and Stützle, 2005). For details on stochastic optimization, we refer the reader to the book by Spall (2003).

2.1.2 Model-Based Optimization

Model-based optimization techniques are typically developed for blackbox optimization, where the only information available about a function to be optimized are values of the function at some queried data points. Given these values, these techniques construct a response surface model, that is, a model that predicts the true value of the function at unseen data points. They use this response surface in order to identify optimal parameter settings. *Experimental design* is concerned with the study of which design points to choose in order to gather data for learning the model. Several measures of a design's optimality exist—for a review on the extensive literature on experimental design see, *e.g.*, the article by Chaloner and Verdinelli (1995).

Sequential model-based optimization methods combine a response surface model with a criterion for selecting the next design point. A very prominent response surface model in statistics, used for example, by Sacks et al. (1989), Jones et al. (1998), and Santner et al. (2003), is a combination of a linear regression model and a noise-free stochastic Gaussian process model (also referred to as a "kriging" model in the statistics literature) that is fitted on the residual errors of the linear model. This model has come to be known as the "DACE" model—an acronym for the "Design and Analysis of Computer Experiments", the title of the paper by Sacks et al. (1989) which popularized the method. A popular criterion for selecting the next design point $\boldsymbol{\theta}$ is the expectation of positive improvement over the incumbent solution at $\boldsymbol{\theta}$ (where the expectation is taken over the response at $\boldsymbol{\theta}$ with respect to the current model). This *expected improvement* criterion goes back to the work of Mockus et al. (1978) and continues to be the most widely-used criterion today. The combination of the DACE model, the expected improvement criterion and a branch and bound method for optimizing the expected improvement criterion makes up the *efficient global optimization (EGO)* algorithm by Jones et al. (1998), a popular framework in statistics for deterministic blackbox function optimization.

The EGO algorithm has been extended by three different lines of work. Two of these extensions deal with an extension to noisy functions: the sequential kriging optimization (SKO) algorithm by Huang et al. (2006), and the sequential parameter optimization (SPO) procedure by Bartz-Beielstein et al. (2005); Bartz-Beielstein (2006). SKO extends EGO by adding noise to the DACE model and augmenting the expected improvement criterion. SPO computes an empirical summary statistic for each design point and fits a noise-free Gaussian process to the values of this statistic. Over time, it increases the number of runs this statistic is based on. For a more detailed description and a comparison of SKO and SPO, see Chapter 10. The third extension, presented by Williams et al. (2000), constructs a Gaussian process model that predicts the response value integrated over a set of "environmental conditions". We discuss this approach in detail in Section 13.2.1. Briefly, it uses its model to optimize marginal performance across the environmental conditions. In the context of algorithm configuration, this could be used to optimize mean performance across a set of problem instances. This extension is computationally expensive, and only applicable for optimizing mean performance. As we will demonstrate in Chapter 9.4, it is not compatible with transformations of the response variable, which are often crucial for strong model performance (Jones et al., 1998; Huang et al., 2006).

From the point of view of algorithm configuration, the main drawbacks of the EGO line of work are

- its limitation to continuous parameters;
- its limitation to noise-free functions or Gaussian-distributed noise;
- the cubic scaling of the time complexity of Gaussian process regression models with respect to the number of data points; and
- its limitation to single problem instances (or mean performance across instances with the limitation that no response transformation be applied.)

In Chapters 10 through 13, we will address these issues in a model-based optimization framework that follows the EGO line of work in its broad strokes.

Not all work on model-based optimization of algorithm parameters builds on Gaussian processes. Ridge and Kudenko (2007) employed classical experimental design methods to tune the parameters of the Max-Min Ant System (MMAS) for solving the travelling salesperson problem. First, they selected values for ten search parameters that represent "low" and "high" levels of that parameter (in statistical terms, factor). Then, they employed a fractional factorial design with a number of replicates for each design point, leading to 1452 design points. Out of these design points, they discarded 3% outliers and then fitted a quadratic model. They used this model for two purposes: to predict the relative importance of each parameter and to predict which combination of parameter settings can be expected to be optimal. It also relies on Gaussian noise assumptions that are often not met by cost distributions in an algorithm configuration setting (Hoos and Stützle, 2005).

Srivastava and Mediratta (2005) used a decision tree classifier to partition parameter configuration space into good and bad configurations for a given domain. All their parameters were discrete or discretized, and the experiments covered scenarios with less than 200 possible parameter configurations. Unfortunately, their evaluation is inconclusive, since the only comparison made is to the *worst* parameter configuration, as opposed to the default or the best parameter configuration.

Bartz-Beielstein and Markon (2004) compared tree-based regression techniques to the DACE model and to a classical regression analysis. Their application domain was to optimize the parameters of an evolution strategy (with four binary and five continuous parameters) and a simulated annealing algorithm (with two continuous parameters) for a single elevator control problem instance. Their experiments are based on 16 data points from a fractional factorial design. Based on the results on this very small data set, they suggested tree-based methods be used in a first stage of algorithm configuration for screening out important factors since they can handle categorical parameters.

2.1.3 Racing Algorithms

A rather different approach for algorithm configuration is based on adaptations of racing algorithms in machine learning (such as Hoeffding races, introduced by Maron and Moore (1994)). Birattari et al. [2002; 2004] developed a procedure dubbed *F-Race* and used it to configure various stochastic local search algorithms. F-Race takes as input an algorithm $\mathcal{A}$,

a finite set of parameter configurations Θ, and an instance distribution $\mathcal{D}$. It iteratively runs the target algorithm with all "surviving" parameter configurations on a number of instances sampled from $\mathcal{D}$ (in the simplest case, each iteration runs all surviving configurations on one instance). After each iteration, F-Race performs a non-parametric Friedman test to check whether there are significant differences among the configurations. If this is the case, it eliminates the inferior configurations using a series of pairwise tests. This process is iterated until only one configuration survives or a given cutoff time is reached. Various applications of F-Race have demonstrated very good performance (for an overview, see the PhD thesis by Birattari (2004)). However, since at the start of the procedure all candidate configurations are evaluated, this approach is limited to situations in which the space of candidate configurations can practically be enumerated. In fact, published experiments with F-Race have been limited to applications with at most 1 200 configurations. A recent extension presented by Balaprakash et al. (2007) iteratively performs F-Race on subsets of parameter configurations. This approach scales better to large configuration spaces, but the version described in that paper handles only algorithms with numerical parameters.

In related work, Chen et al. (2000) introduced an algorithm dubbed *optimal computing budget allocation (OCBA)*. Their application domain is identifying the best out of a finite number of competing designs for a complex system modelled by simulation experiments. This problem is in principle equivalent to the problem of identifying the best out of a finite number of parameter configurations for solving a single problem instance. OCBA is a sequential algorithm for this problem that asymptotically maximizes the probability of identifying the best design given a fixed computational budget (*i.e.*, a fixed number of allowed simulation runs). Their experiments are promising but limited to small configuration spaces (they considered their space of 210 designs "huge").

2.2 Related Work on the Construction of Algorithms

Some approaches aim to automatically construct algorithms to solve instances of a given type. Here, we review methods from the fields of algorithm synthesis and genetic programming.

2.2.1 Algorithm Synthesis

Cognizant of the time-intensive and error-prone nature of manual implementations of complex algorithms, researchers have introduced systems that transform formal problem specifications into synthesized software that is correct by construction. We note that the approaches for algorithm synthesis discussed here complement work on automated algorithm configuration and parameter optimization. They focus on reducing *implementation* effort, while the particular choice of heuristics still rests with the algorithm designer.

Westfold and Smith (2001) described techniques for the automatic reformulation of problems into an effective representation that allows efficient constraint propagation and pruning. They also showed how to automatically derive efficient code for these operations. The authors reported the synthesized programs to be very efficient (sometimes orders of magnitudes faster than manually written programs for the same problem), and attributed this to the specialized representation of constraints and their optimized propagation.

Van Hentenryck and Michel (2007) described the automated synthesis of constraint-based local search methods. In their approach, the algorithm designer only needs to provide a model of the problem in a high-level constraint-based language specifying objectives as well as hard and soft constraints. Their code synthesizer analyzes this high-level model, deduces facts about type and tightness of constraints and uses this information to automatically select local search operators guaranteed to preserve feasibility. The actual code synthesis is based on Comet (Hentenryck and Michel, 2005), an object-oriented programming language that includes modelling and control abstractions to support constraint-based local search. Experiments for a range of constraint programming problems demonstrated that the synthesized code performed almost as good as hand-coded solutions for a large variety of constraint programming domains. We note that in this approach the user still has to specify the meta-heuristic to be used, as well as its parameters. Thus, this approach is perfectly orthogonal to the topic of this thesis. Indeed, we see a lot of potential in combining the two approaches in future work: simply search in the space of possible parameterized meta-heuristics and evaluate a configuration by synthesizing the code for it and running it.

In follow-up work, Monette et al. (2009) introduced the *AEON* system for the domain of scheduling. AEON analyzes a given high-level input model to identify problem characteristics, uses these to classify the problem (using a lookup table) and selects the appropriate solving strategy based on the classification output. (AEON is closely related to the per-instance algorithm configuration approach discussed later in this Chapter, except that here it is applied at the level of *problems* not *problem instances*.) It then uses the corresponding implementation for this solving strategy. Note that AEON supports both complete search and local search, as well as hybrid algorithms generated by composition (*e.g.*, first use a Tabu Search to find a bound B on solution quality, and then apply branch and bound using bound B). As with the work discussed above, we believe that algorithm configuration could be used to explore the space of possible algorithms defined by all basic search procedures and their parameters, as well as their compositions.

Finally, Di Gaspero and Schaerf (2007) presented *EasySyn++*, a different tool for the synthesis of source code for stochastic local search (SLS) methods. Based on the EasyLocal++ framework for developing SLS algorithms, EasySyn++ is a software environment for the fast prototyping of SLS algorithms. While the above-mentioned Comet system follows a top-down approach that—given a high-level problem specification—generates a complete program not to be touched by the user, EasySyn++ follows a bottom-up approach. In particular, it synthesizes most of the code needed on top of the EasyLocal++ framework, leaving only a small—but crucial—portion of the source code to be written by the algorithm designer. While the necessity of having to write low-level code is a drawback, the advantage of EasySyn++ is that algorithm designers have more control over the details of their algorithm.

2.2.2 Genetic Programming

Genetic programming (see, *e.g.*, Poli et al., 2008) evolves a population of programs for solving a given problem. The fitness of an individual can, for example, be assessed by executing the program and measuring its performance on a training set. Thus, it is in principle possible to directly generate effective programs for solving instances of a given class of problems.

Fukunaga (2008) used genetic programming to construct complex variable selection mechanisms for a generic local search algorithm for SAT. These mechanisms are constructed as LISP programs and generated from terminals and functions by means of composition. There is no clear separation between the space of possible mechanisms and the method used to search it: the space of mechanisms is unbounded, but the genetic programming procedure limits itself to a bounded subspace. Fukunaga's generic algorithm only supports variable selection methods based on the prominent SAT local search algorithms GSAT and WalkSAT. His experiments demonstrated that for Random-3-SAT instances the automatically-generated variable selection mechanisms achieved performance comparable to the best local search SAT algorithms in the year 2002.

Li et al. (2005) applied genetic programming to generate optimized sorting algorithms. Their paper introduced two applications of genetic programming to selecting the best combination of various sorting primitives for a given sorting instance. Both approaches take into account instance features, such as number of elements and entropy of values to be sorted. Similar to Fukunaga's work discussed above, the first approach is based on the composition of various primitives in a tree; this approach also shares the problem of an unbounded search space. The second approach is used to generate a classifier system that partitions the space of possible instance features and selects the most appropriate sorting algorithm for each partition. In the experiments Li et al. reported, both new genetic programming approaches yielded substantial speedups over state-of-the-art algorithm libraries on multiple architectures.

Oltean (2005) employed genetic programming to evolve genetic algorithms themselves, searching for the best combination of genetic programming operators, such as mutation, crossover, and selection of individuals. In his experiments the automatically-evolved genetic algorithms outperformed standard implementations of genetic algorithms on a variety of tasks, such as function optimization, the travelling salesperson and the quadratic assignment problem (Oltean, 2005). However, this work did not use a separate test set to evaluate the performance of the final configurations found. Thus, it is unclear to what extent the reported results would generalize to other problem instances.

In a similar vein, Bölte and Thonemann (1996) applied genetic programming to optimize simulated annealing. In particular, they optimized the annealing schedule for solving the quadratic assignment problem. In their experiments, an automatically-found oscillating schedule outperformed the best previously-known simulated annealing algorithms.

Finally, Stillger and Spiliopoulou (1996) applied genetic programming to the database query optimization problem. They evolved a population of query execution plans (QEPs), using QEPs directly instead of using a string representation as had been done in previous work. One interesting feature of the application to database query optimization is that the fitness of a particular individual does *not* require an execution of the QEPs. In contrast, the performance of any QEP can be evaluated before its execution. It is thus possible to search for the best QEP to a given query, and then execute that QEP. Stillger and Spiliopoulou (1996) reported experiments in which the QEPs their method found yielded performance similar to the performance of QEPs constructed with a widely-used iterative improvement method.

2.3 Approaches for Related Problems

So far we have focused on the problem of finding the best parameter configuration for an entire set (or distribution) of problem instances. Related problems are (1) to find the best configuration or algorithm on a per-instance basis; (2) to run multiple algorithms or copies of a single algorithm at once; (3) to adapt algorithm components or parameters during the execution of an algorithm, and (4) to combine algorithms to find the best solution for a single problem. We now survey approaches for these problems.

2.3.1 Per-Instance Approaches

While some algorithms and parameter configurations dominate others in the sense that they perform better on all instances of a problem class, often there is not a single best approach for all problem instances.[1] This observation led Rice (1976) to define the algorithm selection problem: selecting the best algorithm for each problem instance based on computable properties of the instance. In what we call *per-instance algorithm configuration (PIAC)*, analogously, the task is to select the best *parameter configuration* for each problem instance. Note that per-instance algorithm configuration is a generalization of algorithm selection: the choice between different algorithms can be encoded as a categorical top-level parameter of a portfolio-based algorithm that chooses between subalgorithms. PIAC is more complex, however, since it also deals with ordinal and numerical parameters, and since the parameter configuration space is structured. Most importantly, the number of possible configurations to choose from is typically quite small in algorithm selection and typically very large in PIAC. This fact often renders it impossible to perform experiments with every single configuration, as is often done in algorithm selection (see, *e.g.*, Xu et al., 2008).

Methods for solving PIAC are relevant for many of the application scenarios given in Section 1.1, namely the design and development of complex algorithms; empirical studies, evaluations, and comparisons of algorithms; and the actual end use of algorithms in order to solve problems. A robust solution to PIAC would enable algorithm developers to focus on the development of algorithm components and use the PIAC solver to pick the right combination of components for every problem instance. PIAC is in principle more powerful than the "per-distribution" algorithm configuration we consider throughout this thesis. However, it requires the existence of cheaply-computable features that characterize problem instances and are informative about which kind of approaches would work well for a given instance. The models this approach is based on can also require a large amount of training data.

Knuth (1975) introduced a Monte-Carlo approach to estimate the size of a search tree, which can be used to judge the hardness of an instance. Lobjois and Lemaître (1998) used a similar approach to choose the algorithm which can traverse the entire search tree quickest. A related approach was presented by Kilby et al. (2006).

Leyton-Brown et al. (2003b,a, 2009) introduced portfolio-based algorithm selection using predictive models of algorithm runtime, dubbed *empirical hardness models*. These models

[1] One citation typically used in support of this argument is the so-called No Free Lunch (NFL) Theorem (Wolpert and Macready, 1997). However, note that this theorem only applies to optimization problems for which the objective function is specified as a blackbox lookup table—which is far from typical for hard combinatorial problems.

predict the hardness of an unseen test instance based on a set of polytime-computable instance features. In most of their work, Leyton-Brown et al. use linear basis function regression to predict log-transformed algorithm performance. Nudelman et al. (2004) demonstrated that empirical hardness models work on (uniform-random) SAT. Hutter et al. (2006) showed how to apply them to randomized, incomplete algorithms. Xu et al. (2007a) introduced hierarchical hardness models, which first apply a classifier to predict the type or solution status of an instance, and then combine the prediction of lower-level models for the separate classes. Empirical hardness models can be used in a straightforward manner for portfolio-based algorithm selection: simply predict the performance of each candidate algorithm and select the one with best predicted performance. This idea was first used by Leyton-Brown et al. (2003b,a) and was later extended by Xu et al. (2007b, 2008). These portfolio approaches have been repeatedly demonstrated to yield state of the art performance for various classes of SAT problems. Most notably, in each of the 2007 and the 2009 SAT competitions, SATzilla won three gold and two other medals.

Brewer (1994, 1995) proposed a very similar approach in the context of parallel computing. That work used linear regression models to predict the runtime of different implementations of portable, high-level libraries for multiprocessors, with the goal of automatically selecting the implementation and parameter setting with the best predicted runtime on a new architecture. While most other work we discuss here is concerned with solving $\mathcal{NP}$-hard problems, Brewer focused on problems of low polynomial complexity (sorting and stencil computations, with respective asymptotic complexities of $O(n \log n)$ and $O(n^{1.5})$).

While some approaches exist for algorithm *selection*, there is not much work for algorithm *configuration* on a per instance basis, that is, approaches that pick a parameter configuration depending on instance features (but then keep it fixed). Patterson and Kautz (2001) introduced the Auto-WalkSAT algorithm. This approach is based on an easily computable characteristic (the "invariant ratio", see the work by McAllester et al., 1997), which was empirically found to typically be 10% less than the optimal value for WalkSAT's noise parameter. Auto-WalkSAT simply computes the invariant ratio and then sets the noise parameter to its value plus 10%. The experiments Patterson and Kautz (2001) reported show that this simple approach found almost optimal settings of the noise parameter for such heterogeneous problem classes as unstructured, graph colouring, and blocksworld instances. However, it failed for logistics instances for which the above mentioned relationship between invariant ratio and optimal noise parameter does not hold. Auto-WalkSAT is inherently limited to SAT and in particular to the WalkSAT framework and its single noise parameter.

Gebruers et al. (2005) used case-based reasoning—a classification technique—in order to determine the best configuration of a constraint programming algorithm on a per-instance basis. In particular, the parameters being optimized included problem modelling, propagation, the variable selection heuristic, and the value selection heuristic. However, they used a flat representation of all possible parameter configurations that did not exploit the structured parameter configuration space and led to a simple algorithm selection problem instead of a structured per-instance algorithm configuration problem. They did not specify how many parameter configurations are possible nor which kind of instance features were used (except that they are extracted offline, before the algorithm starts). Experiments demonstrated that the

case-based reasoning approach performed better than C4.5 with default parameters for solving instances of the social golfer problem.

Hutter and Hamadi (2005) and Hutter et al. (2006) used an approach based on empirical hardness models in order to select the best parameter settings of two local search algorithms for SAT. They used linear basis function regression in order to fit a joint model of instance characteristics and (continuous) algorithm parameter settings. They then used this model to determine the appropriate parameter settings to be used on a per-instance basis. Their experiments showed that for a mixed benchmark set parameter settings that were automatically selected on a per-instance basis outperformed the best fixed parameter setting by a factor of almost 10. However, the methods introduced in these papers were limited to purely continuous algorithm parameters.

Finally, there are applications of per-instance approaches in compiler optimization. For example, for each program a different combination of compiler optimizations yields the best performance; this is even true on a per-method basis. Both of these problems can be formalized in the PIAC framework, with program features in the first case and method features in the second one. There are a number of possible compiler optimizations, and optimally, one would like to determine the optimal *sequence*. Cavazos and O'Boyle (2006) simplified this problem to only picking the optimizations to be performed, with their order defined externally. For each compiler optimization flag, they performed a binary logistic regression to determine whether or not the optimization would lead to a speedup in a given method. They generated training data offline and for all logistic regressions at once by compiling a large number of training methods under all possible compiler optimizations and recording the best optimization sequence for each one. Whether or not optimization i is used in the best optimization sequence for a training method determines the method's training label for the ith logistic regression. This approach does not take interactions between the optimizations into account. However, the experimental results reported by Cavazos and O'Boyle (2006) showed strong results, suggesting that the independent treatment of optimizations was not too detrimental.

In another application in compiler optimization, Agakov et al. (2006) predicted which program optimizations were promising on a per-program basis. In an offline training phase, they evaluated 1,000 optimization sequences for a set of training programs and kept all good sequences for each training program (within 5% of optimal). For each new program P to be compiled, their method determines the closest training program, P', in Euclidean PCA-reduced feature space, and creates a probability distribution $\mathcal{D}$ based on the optimization sequences with good performance on P'. Then, their method uses distribution $\mathcal{D}$ in a rather simple way to bias a search in parameter space towards good compiler optimizations for P: randomly sample from $\mathcal{D}$ or initialize the population of a genetic algorithm based on a random sample from $\mathcal{D}$. Their experiments showed that this approach sped up the search for good compiler optimizations by up to an order of magnitude compared to a random search or initialization of the GA with a random sample.

2.3.2 Dynamic Algorithm Portfolios

Work on dynamic restarts of algorithms involves multiple independent runs of an algorithm. The approach by Horvitz et al. (2001) and Kautz et al. (2002) executes one algorithm run

at a time, and after an initial observation horizon predicts whether the algorithm run will be good or bad. For this prediction, it uses a binary classification into long and short runs by means of a decision tree. Due to the sometimes extraordinarily long runtimes of bad runs, such an approach can lead to significant speedups of algorithms with heavy-tailed runtime distributions. However, this approach does not scale to drawing multiple decisions during an algorithm run: for each decision, it requires a separate classifier. In order to scale to a more general online case, one would like to learn a single classifier which can handle features that are computed at arbitrary times during the search.

Gagliolo and Schmidhuber (2006) introduced dynamic algorithm portfolios, which run multiple algorithms with different shares in parallel. This approach fits a runtime distribution across instances. (When instance features are available, such a model could also take them into account.) Gagliolo and Schmidhuber stress that the learning in this case does not happen offline but rather in a life-long learning scenario, by which the learned models are updated after each solved problem instance. Arguably, the more important novelty in this work is that predictions are no longer just used to make a single irreversible decision about which algorithm to use. Rather, the decision about how to prioritize algorithms is constantly revisited in the light of new evidence, namely the fact that algorithms have not terminated after having been alloted a certain time. Later work on learning restart strategies by Gagliolo and Schmidhuber (2007) is another example of dynamic algorithm portfolios, applying more principled methodology based on a bandit problem solver.

Finally, low-knowledge algorithm control is an approach by Carchrae and Beck (2004, 2005) to build reactive algorithm portfolios for combinatorial optimization problems. They assume that all algorithms in the portfolio are anytime algorithms that continuously improve a lower bound on solution quality and assign priorities to each algorithm based on its respective improvements of solution quality over time. Algorithms only communicate by sharing the best solutions found so far.

2.3.3 Online Methods

Finally, there exist a variety of approaches that combine different solution strategies during a single algorithm run. The most appropriate strategy to use typically varies over the course of a search trajectory, and online approaches aim to select the most appropriate one in a given situation.

It is hard to attribute the improvements an algorithm incrementally achieves during a search trajectory to single decisions or series of decisions made by the algorithm. This blame-attribution problem invites the use of reinforcement learning. Examples for reinforcement approaches include the STAGE algorithm by Boyan and Moore (2000), as well as work by Lagoudakis and Littman (2000, 2001) on algorithm selection and selecting branching rules in the DPLL procedure for SAT solving. The original results for STAGE were very encouraging, but unfortunately we are not aware of any follow-up work that showed STAGE to outperform state-of-the-art meta-heuristics, in particular iterated local search with simple random perturbations. Lagoudakis and Littman (2000, 2001) showed that in simple domains it is possible to perform better than the best single approach for a particular instance. However, their reinforcement learning approach was limited to very simple characteristics of the current

search state, such as problem size for algorithm selection and number of unassigned variables for SAT tree search algorithms. How to make reinforcement learning work with a more expressive set of search state features is an open and interesting research problem.

Some approaches adaptively change the heuristics to be used in tree search algorithms. Samulowitz and Memisevic (2007) applied an online approach to the problem of solving quantified Boolean formulae (QBF). Their tree search method solves an algorithm selection problem at various levels of the search tree in order to determine the best solution approach for the particular subtree rooted at the current decision point. Even though this approach does not taken future decisions into account (as a reinforcement learner would), their experiments showed that it sped up the search substantially. A more heuristic approach was taken by Borrett et al. (1995). Their *quickest first principle (QFP)* employs a number of algorithms of increasing power (but also increasing complexity), and switches to the next approach when search stagnation is detected. The Adaptive Constraint Engine (ACE) by Eppstein and Freuder (2001) and Epstein et al. (2002) uses a set of so-called advisors, heuristics that vote for possible actions during the search. These votes are taken at every decision point during the search and effectively compose the variable- and value-selection heuristics of ACE.

Work in *hyper-heuristics* originates in the meta-heuristic[2] community and is concerned with the development of heuristics that search a space of heuristic algorithms to identify one of them to be used in a given situation. Burke et al. (2003) surveyed the area and stated as the main motivation behind hyper-heuristics that many competitive meta-heuristics are too problem-specific or too knowledge-intensive to be implemented cheaply in practice. Hyper-heuristics are hoped to raise the level of generality and ease of use of meta-heuristics. Burke et al. gave a framework for hyper-heuristic algorithms, which iterates the following central step: given a problem state S_i, find a heuristic ingredient that is most suitable for transforming that state, and apply it to reach a new state S_{i+1}.

Cowling et al. (2002) discussed the application of such a hyper-heuristic to a university personnel scheduling problem. They introduced several possible move types and defined low-level heuristics that iteratively use one move type until some termination criterion is met. One of their proposed hyper-heuristics adaptively ranks the low-level heuristics and selects effective ones more often. The results presented show that this hyper-heuristic produced very good results, dramatically better than those found manually or by a constructive heuristic.

The local search community has developed a great variety of approaches for adapting search parameters to the algorithm trajectory. The *reactive search* framework by Battiti et al. (2008) uses a history-based approach to decide whether the search is trapped in a small region of the search space, and makes a diversification move more likely when trapped. For example, reactive tabu search makes the tradeoff between intensification (more intensely searching a promising small part of the search space) and diversification (exploring other regions of the search space) via its tabu tenure—the number of steps for which a modified variable cannot be changed after a modification. When search stagnation is detected, reactive tabu search increases the tabu tenure exponentially, and otherwise slowly decreases it.

Hoos (2002a) used a very similar mechanism in an adaptive noise mechanism for WalkSAT.

[2]Note that the term "meta-heuristic" is commonly used to refer to a set of heuristics that are generally applicable to more than one problem. Unlike hyper-heuristics, it does *not* refer to heuristics that act on heuristics.

Instead of the tabu tenure, the resulting *Adaptive Novelty*$^+$ algorithm controls its noise parameter. Adaptive Novelty$^+$ increases its noise if its does not observe any improvement in the objective function for too long and decreases it otherwise. Introduced in 2002, Adaptive Novelty$^+$ is still a competitive local search algorithm for SAT. Hutter et al. (2002) introduced a similar reactive variant for the SAPS algorithm. Reactive SAPS, or RSAPS, adapts the probability of performing a smoothing step, where smoothing corresponds to an intensification of the search. Since the optimal parameter setting may change during the course of the search, in principle, this strategy has the potential to achieve better performance than any fixed parameter setting. In practice this is true for some instances, but overall, SAPS still shows more robust performance with its default parameter settings than RSAPS.

One could argue that online methods are more powerful than approaches that commit to a given parameter configuration before running the algorithm. The flexibility of modifying algorithm components during runtime leads to a much larger configuration space that may indeed contain much better algorithms. However, it is not clear that existing online methods make the best use of this flexibility. Since many decisions have to be made during the course of a search, the computational efficiency of the learning component also becomes an important issue. Thus, research focuses on simple heuristics or hard-coded learning rules (where this use of the term "learning" does not have much in common anymore with traditional machine learning approaches). One exception is the more principled work on reinforcement learning for search. However, so far, this line of work has not resulted in state-of-the algorithms for SAT or related problems. We hope that the type of response surface models we discuss in Part IV of this thesis can be generalized to help tackling the problem of reactively tuning algorithm parameters and choosing algorithms while solving a problem instance.

Finally, work on online methods and on algorithm configuration is in part orthogonal. Online methods often retain many parameters whose settings are kept fixed throughout the search. Thus, it is perfectly possible to configure online methods using automated algorithm configuration procedures. In particular, dynamic methods are often more complex than their static counterparts and automated configuration procedures can facilitate the consideration of such complex mechanisms. In fact, such methods have already have been used to optimize the parameters of a dynamic multi-armed bandit approach for adaptive operator selection (Maturana et al., 2009).

2.3.4 Finding a Single Good Solution for Optimization Problems

Cases where the objective is simply to find a good solution for a given optimization problem cannot be formalized as algorithm configuration problems, except in the case of deterministic algorithms. In algorithm configuration, we try to identify a configuration with good performance across repeated runs, such as, for example, mean solution quality across test runs. In contrast, in some optimization problems, the objective is simply to achieve good performance once; the objective function is thus the *maximal* performance across a number of training runs. For deterministic algorithms, this can be formalized as an algorithm configuration problem where training and test set coincide. For randomized algorithms, one could still apply the algorithm configuration approaches discussed here, but they can be expected to yield suboptimal results since they optimize a different objective function.

Cicirello and Smith (2004) introduced the max k-armed bandit problem in order to model scenarios, in which the goal is to maximize the *maximal* performance achieved during training. Their framework can be used to prioritize runs of various randomized algorithms for optimization problems (or runs of a single algorithm with several parameter configurations). While earlier work (Cicirello and Smith, 2004, 2005; Streeter and Smith, 2006a) assumed generalized extreme value distributions of performance, Streeter and Smith (2006b) applied a distribution-free approach. This approach was used to assign priorities to five different priority dispatching rules for the resource-constrained project scheduling problem with maximal time lags (RCPSP/max). For this problem, round-robin sampling achieved better overall performance than any single rule, and the max k-armed bandit approach further reduced the regret of round-robin sampling by half.

2.4 Further Promising Application Areas for Algorithm Configuration

While the discussion above concentrated on methods, here we discuss two promising application areas for automated algorithm configuration; these applications are rather different from the ones we focus on in this thesis.

2.4.1 Machine Learning

For potential applications of algorithm configuration in supervised machine learning, we distinguish between model parameters and algorithm parameters. In order to fit a given data set, supervised learners typically select the best-fitting model from a given hypothesis space, using either closed-form solutions or some sort of optimization to set a number of *model parameters*. This is *not* what we would refer to as algorithm configuration or parameter tuning. In contrast, there exist *algorithm parameters* that determine the hypothesis space or the method for searching this space; it is those parameters which would lend themselves to the use of automated configuration techniques. Further parameters that could be determined by algorithm configuration procedures are *model hyper-parameters* or *model complexity parameters*; for simplicity, we also refer to these as algorithm parameters. Model parameters are typically set automatically *within* the learning algorithm to minimize some loss function, whereas algorithm parameters are typically determined manually or optimized based on cross-validation. Typical examples include the number of hidden layers in a neural network, the number of neighbours in a nearest neighbour classifier, the choice of which numerical optimizer to use (and how to initialize it), and the manual setting of hyper-parameters. Similar to what is the case for algorithms outside machine learning, in some situations certain algorithm parameters can be transformed into model parameters or avoided altogether: for example, the depth to which to build a decision tree does not need to be specified if one instead uses pruning techniques. The typical optimization objective in machine learning is predictive performance on a previously-unseen test set of instances.

As one of many examples, consider the work of Kohavi and John (1995). They demonstrated that it is possible to automatically find good parameter configurations $\boldsymbol{\theta}$ for the popular

decision tree learning algorithm C4.5. In that study, a parameter configuration consisted of the instantiation of one integer, one continuous, and two binary parameters; the parameter configuration space $\boldsymbol{\Theta}$ was discretized to a total of 1156 choices. For each of 33 data sets in their study, they performed a best-first search in $\boldsymbol{\Theta}$, where each $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ was evaluated by cross validation on the training set. Their approach found different well-performing configurations for each of their data sets which significantly outperformed the default parameter configuration of C4.5 in nine of 33 cases. Note that in our terminology, "machine learning data set" translates to "problem instance $\pi \in \Pi$". A k-fold cross-validation on the training set can thus be seen as assessing average performance of a parameter configuration across k instances. Alternative problem formulations could have been to find the configuration $\boldsymbol{\theta}$ with best performance *across* the 33 data sets, or—given some cheaply computable features for data sets—the best configuration on a per-data-set basis.

Some machine learning algorithms have a rather large number of parameters. One example can be found in the training of restricted Boltzmann machines and deep belief networks, which relies on various implementation "tricks" (Sversky and Murphy, 2009). The right combination of the parameters associated with these "tricks" can be found with automated configuration procedures, an approach that is far closer to the philosophy of *machine* learning than the manual optimization of such parameters.

2.4.2 Optimizing Algorithms for Frequently-Used Polytime Operations

In this thesis, we focus on the optimization of algorithms for solving *hard* computational problems, such as, *e.g.*, $\mathcal{NP}$-hard problems. Very similar approaches would also be applicable to the empirical optimization of polytime algorithms for many prominent problems, such as, *e.g.*, sorting, finding shortest paths in graphs, and linear algebra operations. Here, algorithm configuration interfaces with *algorithm engineering*, an area that combines theoretical algorithm design with the empirical analysis of practical algorithm performance. While algorithm engineering manually combines algorithm components to yield algorithms that are often provably efficient (but can also include heuristics), some or all of this task could be automated. Even though the theoretical performance guarantees may not change (and even if they degrade, as, *e.g.*, in Quicksort compared to MergeSort), empirical performance can improve by orders of magnitude when using careful implementations.

As an example, consider the construction of fast sorting algorithms. One example for the engineering effort necessary in this task is given by Bentley and McIlroy (1993). That paper specifically concludes "We have added a few new tricks to Quicksort's bag [...] We mixed these with a few old tricks, ignored many others, and arrived at the new champion Program." We note that these "tricks" often involve numerical parameters and combinations of categorical algorithm components. For example, in their work there are three integer parameters which determine the size of subarray for which to use InsertionSort and select between different adaptive partitioning schemes. Further, the swapping macro is highly optimized and contains categorical decisions about whether to inline algorithm code or not. These explicit optimizations sped up swaps by a factor of 12 for the common special case of sorting integer arrays.

Another example can be found in the problem of finding the shortest path between

two vertices of a graph (V, E). Even though this problem can be solved by Dijkstra's classical algorithm (Dijkstra, 1959) in time $O(|E| + |V| \log(|V|))$, until recently practitioners used heuristic solutions without correctness guarantees. Only the last decade has seen the development of (parameterized) algorithms that are both exact and empirically outperform Dijkstra's algorithm by up to a factor of one million (Sanders and Schultes, 2007).

A final but no less important problem is the optimization of high-performance libraries, such as those in a linear algebra package. Research in automatically optimizing such libraries has demonstrated that frequently-used routines can be optimized to run orders of magnitude faster than naïve implementations (Whaley, 2004). However, the optimizations are platform-specific and an optimization on one platform may cause a performance decrease on another. Thus, the approach taken in the ATLAS project (Whaley et al., 2001) is to optimize algorithm performance on the end user side. Custom search heuristics are used to automatically determine the most appropriate implementation for the user's architecture, which—next to setting various numerical parameters—also requires making some categorical choices. General algorithm configuration procedures could be used for cases where custom heuristics either do not perform well or simply have not been developed yet.

In all of these potential applications of automated algorithm configuration procedures, single algorithm runs would only require a small fraction of the time they take in the applications typical for this thesis. Nevertheless, the configuration procedures we introduce in this thesis would remain valid. However, for the optimization of extremely fast operations it appears advisable to pay special attention to keeping the overhead of configuration procedures small. Another characteristic shared by algorithms for all these applications is their widespread use. Improvements would thus have an immediate impact on a large user base.

2.5 Chapter Summary

In this chapter, we discussed related work from various fields of research. We started with the most closely-related work, methods that apply relatively directly to (special cases) of the algorithm configuration problem we defined in Section 1.2.2. Most such existing methods are based on direct search in parameter configuration space, some are based on response surface models, and some on statistical racing techniques.

We then discussed methods for automatically constructing algorithms. This included methods for the construction of correct implementations based on high-level specifications and no or little manual implementation effort in low-level programming languages. We also discussed related methods from genetic programming that evolve programs for solving problem instances of a given type. In such methods, the specification of possible programs is often tightly coupled to the methods used to search the space of programs.

Next, we discussed a variety of approaches for problems related to algorithm configuration, such as per-instance algorithm selection and parameter tuning, algorithm portfolios, and methods that adapt algorithm parameters while solving a problem instance.

Finally, we discussed two areas of research that promise to hold much potential for applying automated algorithm configuration procedures: machine learning and the optimization of algorithms for frequently-used polytime operations.

Part II

Algorithm Configuration: Scenarios and Empirical Analysis

—in which we introduce our algorithm configuration scenarios and empirically analyze their characteristics

Chapter 3

Algorithm Configuration Scenarios

We have to learn again that science without contact with experiments is an enterprise which is likely to go completely astray into imaginary conjecture.
—Hannes Alfvén, Swedish plasma physicist

In this chapter, we describe the algorithm configuration scenarios we use throughout this thesis to experimentally evaluate configuration procedures. We constructed these configuration scenarios—instances of the algorithm configuration problem—to study various aspects of algorithm configuration.

Algorithm configuration aims to improve the performance of existing target algorithms for solving sets of instances of a problem. First, in Section 3.1, we discuss the two problems we focus on: propositional satisfiability (SAT) and mixed integer programming (MIP). Then, we introduce target algorithms (Section 3.2) for these problems, sets of benchmark instances (Section 3.3), and optimization objectives (Section 3.4). Next, we introduce seven sets of configuration scenarios that we will use throughout the thesis (Section 3.5). Finally, at the end of this chapter, we also discuss experimental preliminaries that apply throughout the thesis.

This chapter serves as a reference and can be skimmed or skipped by readers eager to move on.

3.1 Problems

We studied algorithm configuration for two problems in detail. In particular, we focused on SAT, the most-widely studied $\mathcal{NP}$-complete problem, and mixed integer programming (MIP), which is important due to the wide spread of problem formulations as MIP in industry and academia.

Beyond these two, we considered additional problems to demonstrate the generality of our methods. In particular, we studied global continuous blackbox optimization and the most probable explanation problem (MPE).

3.1.1 Propostitional Satisfiability (SAT)

The propositional satisfiability (SAT) problem is the prototypical $\mathcal{NP}$-hard problem (Garey and Johnson, 1979). All SAT instances we used were propositional formulae in conjunctive normal form (CNF), that is, each instance is a conjunction of disjunctions of literals (negated or non-negated variables). The goal is to determine whether or not there exists a variable assignment under which the formula evaluates to TRUE. We restrict ourself to CNF-encoded formulae since this is the widely-accepted standard input format for SAT solvers. Any propositional formula can be encoded into CNF with linear overhead by adding additional variables.

3.1.2 Mixed Integer Programming (MIP)

Mathematical programs are general optimization problems of the form

$$\begin{aligned} \min \quad & f(\boldsymbol{x}) \\ \text{s.t.} \quad & c(\boldsymbol{x}), \end{aligned}$$

where $f(\cdot)$ is the objective function and $c(\cdot)$ expresses feasibility constraints. In integer programs, all variables x_i are constrained to be integer. In *mixed integer programs* only a subset of variables is constrained in that way. In the frequent case of mixed integer linear programs (MILP), $f(\cdot)$ is a linear function. Even with binary domains for each x_i this conceptually simple problem is $\mathcal{NP}$-hard.

All MIP instances we used can be expressed in the rather general representation used by CPLEX 10.1.1, which we used to solve these instances:

$$\begin{aligned} \text{minimize} \quad & 1/2 \cdot \boldsymbol{x}^\mathsf{T} \cdot \boldsymbol{Q} \cdot \boldsymbol{x} + \boldsymbol{c}^\mathsf{T} \cdot \boldsymbol{x} \\ \text{subject to} \quad & \boldsymbol{A} \cdot \boldsymbol{x} \bowtie \boldsymbol{b} \\ & \boldsymbol{a}_i^\mathsf{T} \cdot \boldsymbol{x} + \boldsymbol{x}^\mathsf{T} \cdot \boldsymbol{Q}_i' \cdot \boldsymbol{x} \leq r_i \text{ for } i = 1, \ldots, q \\ & l_i \leq x_i \leq u_i \text{ for } i = 1, \ldots, n \\ & x_i \text{ is integer for } i \text{ in a subset of } \{1, \ldots, n\}, \end{aligned}$$

where $\bowtie$ may be $\geq$, $\leq$, or $=$, $\boldsymbol{x}$ denotes the n optimization variables, $\boldsymbol{Q}$ is an $n \times n$ matrix of quadratic coefficients of the objective function, $\boldsymbol{c}$ is a column vector of linear coefficients of the objective function, the $m \times n$ matrix $\boldsymbol{A}$ and the column vector $\boldsymbol{b}$ specify linear constraints, the $n \times n$ matrices $\boldsymbol{Q}_i'$ and scalars r_i specify quadratic constraints, and the scalars l_i and u_i specify lower and upper bounds on x_i. These lower and upper bounds can be positive or negative infinity or any real number.

3.2 Target Algorithms

In this section, we first describe the SAT solvers we used, then the commercial optimization tool CPLEX for MIP, and finally the algorithms for the other two problems. Table 3.1 gives an overview of those target algorithms for which we considered a discretized parameter

Algorithm	Parameter type	# params of type	# values considered	Total # configs, $\lvert\Theta\rvert$
CPLEX	Categorical	50	2–7	$1.38 \cdot 10^{37}$
	Integer	8	5–7	
	Continuous	5	3–5	
SPEAR	Categorical	10	2–20	$8.34 \cdot 10^{17}$
	Integer	4	5–8	
	Continuous	12	3–6	
SATENSTEIN	Categorical	16	2–13	$4.82 \cdot 10^{12}$
	Integer	5	4–9	
	Continuous	20	3–10	
SAT4J	Categorical	4	3–6	567 000
	Integer	4	5–7	
	Continuous	3	5	
SAPS	Continuous	4	7	2 401
GLS$^+$	Categorical	1	2	1 680
	Integer	2	5–7	
	Continuous	2	4–6	

Table 3.1: Target algorithms and their parameters. We list those algorithms for which we discretized all parameters, in order of decreasing complexity (measured in terms of total number of parameter configurations).

Algorithm	Parameter type	# parameters of type
SAPS	Continuous	4
CMA-ES	Integer	1
	Continuous	3

Table 3.2: Target algorithms with only numerical parameters.

configuration space. Table 3.2 lists our target algorithms with only numerical parameters. Note that SAPS is contained in both tables; we performed experiments with both discretized and continuous versions of its parameter configuration space.

Throughout, we paid careful attention to only selecting high-performance solvers as target algorithms to be configured. Improving poor algorithms might be much easier, but good results for such algorithms might not imply improvements of the state of the art.

3.2.1 Target Algorithms for SAT

For SAT, we considered two local search and two tree search algorithms.

Stochastic Local Search Algorithms for SAT

SAPS This stochastic local search (SLS) SAT solver is a type of dynamic local search algorithm, that is, it dynamically changes its internal cost function during the search. SAPS does this by associating weights with the clauses of the input SAT formula. In each encountered

local minimum, m, it increases the weights of the currently unsatisfied clauses, thereby changing the cost function to ensure that m is not a local minimum anymore. At each iteration, with a certain probability clause weights are smoothed towards the uniform distribution. SAPS(short for "Scaling And Probabilistic Smoothing") was introduced by Hutter et al. (2002); we use the UBCSAT implementation by Tompkins and Hoos (2004). When introduced in 2002, SAPS was a state-of-the-art solver, and, with appropriately-chosen parameters, it still offers high performance on many types of "crafted" and unstructured instances. We chose to study this algorithm because it is well known, it has relatively few parameters, we are intimately familiar with it, and we knew from earlier work that SAPS's parameters can have a strong impact on its performance (Hutter et al., 2006). Its four continuous parameters control the scaling and smoothing of clause weights, as well as the percentage of random steps. Their original defaults were set through manual configuration based on experiments with prominent sets of benchmark instances; this manual experimentation kept the percentage of random steps fixed and took up about one week of development time. We gained more experience with SAPS' parameters for more general problem classes in our early work on parameter optimization (Hutter et al., 2006). Informed by that work, we chose promising intervals for each parameter, including but not centered at the original default.

For our experiments with procedures restricted to discretized configuration spaces, we picked seven possible values for each parameter. These were spread uniformly across its respective interval, that is, equally dividing the interval. For the one multiplicative parameter ('α'), we picked these seven values uniformly on a log scale, that is, by an equal division of its log-transformed interval. This resulted in $2\,401$ possible parameter configurations (exactly the values used in Hutter et al., 2007b). As the starting configuration for configuration procedures requiring discretized values, we used the center of each parameter's interval.

We also performed configuration experiments without discretizing SAPS' parameters. In these experiments, we used the same interval as above for each parameter, extending from the minimum to the maximum of the seven values chosen above. Since the original SAPS default is contained in this continuous space, we used it as the starting configuration.

SATENSTEIN This highly parameterized framework for stochastic local search (SLS) SAT solvers was very recently introduced by KhudaBukhsh et al. (2009). SATENSTEIN draws on components from WalkSAT-based algorithms (Selman et al., 1996), dynamic local search algorithms (Hutter et al., 2002) and G^2WSAT variants (Li and Huang, 2005), all combined in a highly parameterized framework solver with a total of 41 parameters. It covers almost all state-of-the-art SLS solvers for SAT and is designed to be used in conjunction with a configuration procedure to automatically construct new SLS solvers for new types of problem instances of interest.

The parameter configuration space of SATENSTEIN is highly structured. There is one top-level parameter that decides whether to construct a dynamic local search (DLS) type algorithm or a non-DLS algorithm. The subspace for DLS algorithms has 21 parameters, 17 out of which are conditional parameters that are only active depending on certain instantiations of other (higher-level) parameters. The non-DLS subspace has 36 parameters, 29 of which are conditional parameters. Thus, an efficient mechanism for handling conditional parameters

might be important for the effective configuration of SATENSTEIN. In total, the configuration space of SATENSTEIN is of size $4.82 \cdot 10^{12}$. When studying $1\,000$ random configurations of SATENSTEIN in Chapter 4, we used 500 configurations from each of those two subspaces.

Tree Search Algorithms for SAT

SPEAR This recent tree search algorithm for SAT solving was developed by Babić (2008). It is a state-of-the-art SAT solver for industrial instances, and with appropriate parameter settings it is the best available solver for certain types of SAT-encoded hardware and software verification instances (Hutter et al., 2007a). Furthermore, configured with one of the algorithm configuration tools we introduce in this thesis (PARAMILS), SPEAR won the quantifier-free bit-vector arithmetic category of the 2007 Satisfiability Modulo Theories Competition.

SPEAR has 26 parameters, including ten categorical, four integer, and twelve continuous parameters. Their default values were manually engineered by its developer, using a benchmark set of relatively small software verification and bounded model checking instances. (Manual tuning required about one week.) The categorical parameters mainly control heuristics for variable and value selection, clause sorting, resolution ordering, and also enable or disable optimizations, such as the pure literal rule. The continuous and integer parameters mainly deal with activity, decay, and elimination of variables and clauses, as well as with the interval of randomized restarts and percentage of random choices. We discretized the integer and continuous parameters by choosing lower and upper bounds at reasonable values and allowing between three and eight discrete values spread relatively uniformly across the resulting interval, including the default. The number of discrete values was chosen according to our intuition about the importance of the parameter. After this discretization, there were $3.7 \cdot 10^{18}$ possible parameter configurations. Exploiting the fact that nine of the parameters are conditional (i.e., only relevant when other parameters take certain values) reduced this to $8.34 \cdot 10^{17}$ configurations. As the starting configuration for our configuration procedures, we used the default.

In Section 6.3.1, we discuss SPEAR's parameter configuration space in more detail in the context of a case study for algorithm configuration.

SAT4J This library[1], developed by Le Berre, provides an implementation of SAT solvers in Java. It is targeted at users who would like to embed a black box SAT solver in their application without worrying about the details. SAT4J is based on an implementation of MiniSAT (Eén and Sörensson, 2004), extended with new heuristics and learning strategies. In our experiments, we used a modified version of SAT4J version 1.5, for which all parameters had been made accessible from the command line (the version number outputted by the code is "OBJECTWEB.1.0.113").[2] These parameters include categorical choices of variable ordering, learning strategy, data structure and clause minimization. One variable ordering heuristic and three learning heuristics are parameterized with one additional continuous parameter

[1] http://www.sat4j.org/index.php

[2] Many thanks to Daniel Le Berre for preparing this version for us.

each. Finally, three other numerical parameters govern variable decay, first restart and the multiplication factor of the restart interval.

We only experimented with SAT4J early on and replaced it with SPEAR later, since SPEAR was typically faster and in particular was the best available algorithm for some important industrial benchmark distributions. However, we do see much promise in integrating automated algorithm configuration techniques with current versions of SAT4J. SAT4J's typical usage scenarios as a blackbox SAT solver for users unfamiliar with SAT technology could be combined nicely with an automated option to improve the solver for the (a priori unknown) types of problem instances it faces.

3.2.2 Target Algorithm for MIP

CPLEX The most-widely used commercial optimization tool for solving mixed integer programs is ILOG CPLEX. As stated on the CPLEX website (http://www.ilog.com/products/cplex/), currently over 1 300 corporations and government agencies use CPLEX, along with researchers at over 1 000 universities. CPLEX is massively parameterized and considerable effort has been expended to engineer a good default parameter configuration:

> "A great deal of algorithmic development effort has been devoted to establishing default ILOG CPLEX parameter settings that achieve good performance on a wide variety of MIP models." (ILOG CPLEX 10.0 user manual, page 247)

Despite these efforts, the end user still sometimes has to experiment with parameters:

> "Integer programming problems are more sensitive to specific parameter settings, so you may need to experiment with them." (ILOG CPLEX 10.0 user manual, page 130)

As such, the automated configuration of CPLEX is very promising and has the potential to directly impact a large user base.

For the experiments in this thesis, we used CPLEX version 10.1.1. Out of its 159 user-specifiable parameters, we identified 81 parameters that affect its search trajectory. We did this without any experience with CPLEX, solely based on two days spent with its manual. Thus, we may have omitted some important parameters or included inconsequential ones. We were careful to omit all parameters that change the problem formulation (*e.g.*, by changing the numerical accuracy of a solution). Many CPLEX parameters deal with MIP strategy heuristics (such as variable and branching heuristics, probing, dive type, and subalgorithms) and amount and type of preprocessing to be performed. There are also nine parameters each governing how frequently a different type of cut should be used (there are four magnitude values and the value "choose automatically"; note that this last value prevents the parameters from being ordinal). A considerable number of other parameters deal with simplex and barrier optimization, and with various other algorithm components. For categorical parameters with an automatic option, we considered all categorical values as well as the automatic one. In contrast, for continuous and integer parameters with an automatic option, we chose that option instead of hypothesizing

values that might work well. We also identified some parameters that primarily deal with numerical accuracy, and fixed those to their default values. For other numerical parameters, we chose up to five possible values that seemed sensible, including the default. For the many categorical parameters with an automatic option, we included the automatic option as a choice for the parameter, but also included all the manual options. Finally, we ended up with 63 configurable parameters, leading to $1.78 \cdot 10^{38}$ possible configurations. Exploiting seven conditional parameters reduced this to $1.38 \cdot 10^{37}$ distinct configurations. As the starting configuration for our configuration procedures, we used the CPLEX default settings.

3.2.3 CMA-ES for Global Continuous Function Optimization

This prominent gradient-free global optimization algorithm for continuous functions (Hansen and Ostermeier, 1996; Hansen and Kern, 2004) is based on an evolutionary strategy that uses a covariance matrix adaptation scheme. CMA-ES has been used as an application domain of parameter optimization algorithms (Bartz-Beielstein et al., 2008). We used the interface they wrote for the SPO toolbox, which used the Matlab implementation of CMA-ES 2.54.[3] As an evolutionary strategy, CMA-ES has two obvious parameters: the number of parents, N, and a factor $\nu \geq 1$ relating the number of parents to the population size. (The population size is defined as $\lfloor N \cdot \nu + 0.5 \rfloor$.) Furthermore, Bartz-Beielstein et al. (2008) modified CMA-ES's interface to expose two additional parameters: the "learning rate for the cumulation for the step size control", c_σ or `cs`, and the damping parameter, d_σ or `damps` (for details, see Hansen, 2006). All these parameters are numerical. We used CMA-ES as a test case for optimizing such parameters without the need for discretization.

3.2.4 GLS$^+$ for the Most Probable Explanation (MPE) Problem

GLS$^+$ is a guided local search algorithm for solving the Most Probable Explanation (MPE) problem in discrete-valued graphical models, that is, the problem of finding the variable instantiation with maximal overall likelihood (Hutter et al., 2005; Hutter, 2004). It has five parameters, one binary and four continuous. The binary parameter decides whether to initialize at random or by using a weight-bounded version of the Mini-Buckets approximation algorithm (Dechter and Rish, 2003). The numerical parameters govern the scaling of clause penalties (amount and interval), the weighting factor for clause weights and the weight bound on preprocessing with Mini-Buckets. We only performed experiments with discretized parameters (between 4 and 7 values per numerical parameter, depending on our intuition as to its importance). This led to a parameter configuration space of size 1 680.

3.3 Benchmark Sets

We selected our benchmark distributions by the following principles. Firstly, we only used standard benchmark distributions for a problem that had previously been used to evaluate algorithms for that problem. This avoided the potential issue of creating a problem distribution

[3] The newest CMA-ES version, 3.0, differs mostly in the interface and in supporting "separable" CMA (see the change log at http://www.lri.fr/~hansen/cmaes_inmatlab.html).

that lends itself to our approach. Secondly, whenever possible we used distributions of practical relevance, to ensure improvements we make by automated configuration would have immediate impact. Prime examples are our two sets of industrial verification instances; some of our CPLEX distributions are also derived from industrial applications. We selected instance distributions of varying hardness. While easy instances were useful for facilitating algorithm development, we needed to verify the performance of our configurators on harder instance distributions. Along the same lines, for the evaluation of configuration procedures that are limited to single instances, we selected a variety of single instances of varying hardness.

Generally, we randomly split each benchmark set 50-50 into training and test sets. All configuration runs only had access to the training set, and we report results on the test set unless explicitly stated. For benchmark "sets" that only contain one instance, we measured test performance on independent test runs with different random seeds but on the same single instance.

3.3.1 SAT Benchmarks

For our experiments on configuring the various SAT solvers discussed above, we selected structured instances from two broad categories: "crafted" instances, based on encodings of randomly-generated instances of other $\mathcal{NP}$-hard problems that are prominent in SAT research, and "industrial" instances based on encodings of industrial verification instances. For the study of configuration procedures that are limited to single problem instances, we selected instances of varying hardness within each of these categories. All instances we study contain structure that can potentially be exploited by SAT solvers. This structure is due to the SAT encoding and to the structure in the original problem.

Encodings of Randomly Generated Instances of Other Hard Problems

QCP This benchmark set contains $23\,000$ instances of the quasi-group completion problem (QCP), which has been widely studied by researchers in artificial intelligence and constraint programming. The objective in this problem is to determine whether the unspecified entries of a partial Latin square can be filled to obtain a complete Latin square. Latin squares play a role in applications such as scheduling, timetabling, experimental design, and error correcting codes. They have been widely used for benchmarking purposes (Gomes and Selman, 1997). Xu generated these QCP instances around the solubility phase transition, using the parameters given by Gomes and Selman (1997). Specifically, the order n was drawn uniformly from the interval $[26, 43]$, and the number of holes H (open entries in the Latin square) was drawn uniformly from $[1.75, 2.3] \cdot n^{1.55}$.

For use with complete SAT solvers, we sampled $2\,000$ of these SAT instances uniformly at random. These had on $1\,497 \pm 1\,094$ variables (mean $\pm$ standard deviation across all instances) and $13\,331 \pm 12\,473$ clauses; $1\,182$ of the instances were satisfiable. For use with incomplete SAT solvers, we randomly sampled $2\,000$ instances from the subset of satisfiable instances (determined using a complete algorithm). Their number of variables and clauses were very similar to those of the set used for complete solvers ($1\,515 \pm 1\,173$ variables, $14\,304 \pm 12\,528$

clauses).

Single Instances QCP$_{med}$, QCP$_{q075}$, and QCP$_{q095}$ These three single instances are the instances at the 50%, 75%, and 95% quantiles of the subset of satisfiable instances of QCP in terms of hardness for SAPS with its default configuration. They contain $1\,009$, $2\,367$, and $2\,946$ variables and $7\,647$, $23\,332$, and $29\,096$ clauses, respectively.

Single Instance QWH This benchmark consists of a single quasigroup completion problem based on a quasigroup with randomly punched holes. (QWH differs from QCP in that QWH instances are always satisfiable by construction.) It is one of a set of $10\,000$ instances that Xu generated using `lsencode` by Gomes and Selman (1997), namely the instance at the 50% quantile of hardness for SAPS with its default configuration. This instance contains $1\,077$ variables and $7\,827$ clauses. In our experiments, it could be solved by SAPS in a median number of $85\,500$ search steps, taking well below a second on our reference machines (see Section 3.6). It thus allowed us to perform many target algorithm runs quickly and so facilitated algorithm development. For this reason, we used this instance extensively in early development phases of both algorithm configuration frameworks discussed in this thesis: model-free search in Part III and model-based search in Part IV.

SWGCP This benchmark set contains $20\,000$ instances of the graph colouring problem (GCP) based on the small world (SW) graphs of Gent et al. (1999). Using their generator, Xu created these instances, with a ring lattice size sampled uniformly at random from $[100, 400]$, each node connected to the 10 nearest neighbours, a rewiring probability of 2^{-7} and chromatic number 6. We sampled $2\,000$ of these instances uniformly at random for use with complete SAT solvers. These had $1\,813 \pm 703$ variables and $13\,902 \pm 5\,393$ clauses; $1\,109$ of the instances were satisfiable. For use with incomplete local search solvers, we randomly sampled $2\,000$ satisfiable instances (again, determined using a complete SAT algorithm). Their number of variables and clauses was very similar to that in the set for complete solvers ($1\,958 \pm 646$ variables, $15\,012 \pm 4\,953$ clauses).

Single Instances SWGCP$_{med}$, SWGCP$_{q075}$, and SWGCP$_{q095}$ These three single instances are the instances at the 50%, 75%, and 95% quantiles of the subset of satisfiable instances of SWGCP in terms of hardness for SAPS with its default configuration. They contain $2\,616$, $2\,586$, and $1\,956$ variables respectively; and $20\,056$, $19\,826$, and $14\,996$ clauses, respectively. (Note that although often instance hardness correlates with instance size, this is not a strict rule.)

Structured Instances from Industrial Verification Problems

SWV This set of SAT-encoded software verification instances comprises 604 instances generated with the CALYSTO static checker (Babić and Hu, 2007b), used for the verification of five

programs: the spam filter Dspam, the SAT solver HyperSAT, the Wine Windows OS emulator, the gzip archiver, and a component of xinetd (a secure version of inetd). These instances contain $64\,416 \pm 53\,912$ variables and $195\,058 \pm 174\,534$ clauses. (We only employed complete solvers for these instances, and thus did not create a separate subset of satisfiable instances.)

Single Instances SWV$_{med}$, SWV$_{q075}$, and SWV$_{q095}$ These three single instances are the instances at the 50%, 75%, and 95% quantiles of SWV in terms of hardness for SPEAR with its default configuration. They contain $92\,737$, $72\,596$, and $114\,395$ variables and $273\,793$, $214\,459$, and $370\,825$ clauses, respectively.

BMC This set of SAT-encoded bounded model checking instances comprises 765 instances generated by Zarpas (2005); these instances were selected as the instances in 40 randomly-selected folders from the IBM Formal Verification Benchmarks Library. These instances contained an average of $91\,041$ variables and $386\,171$ clauses, with respective standard deviations of $149\,815$ and $646\,813$. Some of the instances in this set are extremely hard, the largest instance containing $1\,400\,478$ variables and $5\,502\,329$ clauses. (As above, we only employed complete solvers for these instances and did not create a separate subset of satisfiable instances.)

Single Instances IBM$_{q025}$ and IBM$_{med}$ These two single instances are the instances at the 25% and 50% quantiles of BMC in terms of hardness for SPEAR with its default configuration. They contain $45\,853$ and $29\,725$ variables and $151\,611$ and $125\,883$ clauses, respectively. (Unlike in the case of the other single instances, we did not use instances from the 75% and 95% quantiles from this distribution since these could not be solved by any parameter configuration we studied.)

3.3.2 MIP Benchmarks

For our configuration experiments with CPLEX, we considered a variety of benchmark sets that have previously been used in computational experiments for CPLEX. As in the case of SAT, we used encodings of another $\mathcal{NP}$-hard problem prominent in the evaluation of MIP algorithms, as well as prominent instance types based on industrial applications.

Benchmarks from the Combinatorial Auction Test Suite

In some types of auctions, bidders can bid on sets of goods rather than on single goods. The winner determination problem in such *combinatorial auctions* is to determine the revenue-maximizing allocation of goods to bidders. This problem is $\mathcal{NP}$-hard (Rothkopf et al., 1998) and instances of it can be easily encoded as MILPs. We created two instance sets using the generator provided with the Combinatorial Auction Test Suite (Leyton-Brown et al., 2000). These two sets are very similar, only differing in instance size, which allows us to study how our methods scale to larger instances. Instances based on this generator have been used before to model CPLEX performance and to perform per-instance algorithm selection (Leyton-Brown

et al., 2009).

Regions100 For this benchmark set we generated 2 000 MILPs based on the above generator and using the `regions` option with the 'goods' parameter set to 100 and the 'bids' parameter set to 500. The resulting MILP instances contained 501 variables and 193 inequalities on average, with a standard deviation of 1.7 variables and 2.5 inequalities.

Regions200 This set is very similar to the `Regions100` set, but its instances are much larger. We generated 2 000 MILP instances as above, but now with the 'goods' parameter set to 200 and the 'bids' parameter set to 1 000. These instances contain an average of 1 002 variables and 385 inequalities, with respective standard deviations of 1.7 and 3.4.

Benchmarks from Berkeley Computational Optimization Lab

We obtained a variety of instance sets from the Berkeley Computational Optimization Lab.[4]. We only used sets of instances that were large and homogeneous enough to be split into disjoint training and test sets and still have the training set be quite representative.

MJA This benchmark set, introduced by Aktürk et al. (2007), comprises 343 machine-job assignment instances encoded as mixed integer quadratically constrained programs (MIQCP). These instances contain $2\,769 \pm 2\,133$ variables and $2\,255 \pm 1\,592$ constraints.

CLS This set comprises 100 capacitated lot-sizing instances encoded as mixed integer linear programs (MILP). It was introduced by Atamtürk and Muñoz (2004). All 100 instances contain 181 variables and 180 constraints.

MIK This set of 120 MILP-encoded mixed-integer knapsack instances was originally introduced by Atamtürk (2003). The instances in this set contain an average of 384 ± 309 variables and 151 ± 127 constraints.

Other MIP Benchmarks

QP This set of quadratic programs originated from RNA energy parameter optimization (Andronescu et al., 2007). Andronescu generated 2 000 instances for our experiments. These instances contain $9\,366 \pm 7\,165$ variables and $9\,191 \pm 7\,186$ constraints. Since the instances are polynomial-time solvable quadratic programs. Thus, in configuration scenarios in which we optimize CPLEX performance for these instances, we set a large number of inconsequential CPLEX parameters concerning the branch and cut mechanism to their default values, ending up with 27 categorical, 2 integer and 2 continuous parameters to be configured, for a discretized parameter configuration space of size $3.27 \cdot 10^{17}$.

[4] http://www.ieor.berkeley.edu/~atamturk/bcol/ (where set `MJA` has the name conic.sch)

ORLIB This benchmark set includes a heterogeneous mix of 140 instances from ORLIB (Beasley, 1990): 59 set covering instances, 20 capacitated p-median problems, 37 capacitated warehouse location instances, and 24 airplane landing scheduling instances. We obtained these instances from the website by Saxena (2008). We selected this set to study the characteristics of a very heterogeneous instance set consisting of various clusters of instances.

3.3.3 Test Functions for Continuous Global Optimization

For the optimization of the global continuous optimization algorithm CMA-ES, we considered four canonical 10-dimensional test functions that have previously been used in published evaluations of CMA-ES. We used one rather simple test function whose global optimum is its only local optimum, as well as three global test functions with many local optima. For each of these functions, the global optimum is $\boldsymbol{x}^* = [0, \ldots, 0]$ with function value $f(\boldsymbol{x}^*) = 0$. We now define the four functions, in n dimensions each:

$$f_{Sphere}(\boldsymbol{x}) = \sum_{i=1}^{n} x_i^2.$$

$$f_{Ackley}(\boldsymbol{x}) = 20 - 20 \cdot \exp\left[-0.2\sqrt{\sum_{i=1}^{n} x_i^2/n}\right] + \exp(1) - \exp\left[\sum_{i=1}^{n} \cos(2\pi x_i)/n\right]$$

$$f_{Rastrigin}(\boldsymbol{x}) = 10 \cdot \left[n - \sum_{i=1}^{n} (\cos(2\pi x_i))\right] + \sum_{i=1}^{n} x_i^2.$$

$$f_{Griewangk}(\boldsymbol{x}) = 1 - \prod_{i=1}^{n} \cos(x_i/\sqrt{(i)}) + \sum_{i=1}^{n} x_i^2/4000.$$

For all of these functions, we used Hansen's implementations, which are part of the CMA-ES source code.[5]

3.4 Optimization Objective: Penalized Average Runtime

As stated in Section 1.2.2, the optimization objective in algorithm configuration can be chosen by the user to best reflect the intended use of the configured algorithm. Indeed, we explored several optimization objectives in some of our published work: maximizing solution quality achieved in a given time, minimizing the runtime required to reach a given solution quality, and minimizing the runtime required to solve a single problem instance (Hutter et al., 2007b).

Throughout this thesis, however, we concentrate on the objective of minimizing the mean runtime over instances from a distribution $\mathcal{D}$. This optimization objective naturally occurs in many practical applications. It also implies a strong correlation between the true cost $c(\boldsymbol{\theta})$ of a configuration and the amount of time required to obtain a good empirical estimate of $c(\boldsymbol{\theta})$.

[5] http://www.lri.fr/~hansen/cmaes_inmatlab.html

Specifically, in a given time, the expected number of algorithm runs we can perform with a configuration $\boldsymbol{\theta}$ is directly proportional to the true cost of $\boldsymbol{\theta}$. In Chapter 7 we exploit this relationship to significantly speed up our configuration procedures.

One might wonder whether the mean is the best choice for aggregating empirical runtimes. While we used medians in our first publication on algorithm configuration (Hutter et al., 2007b), we later found that the optimization of quantiles can result in parameter configurations with poor robustness. This is intuitive because the Q-percent quantile does not reflect characteristics of the distribution above that quantile: $(100 - Q - \epsilon)$ percent of runs may have a very large, even infinite, runtime. Empirically, we found that minimizing mean runtime led to parameter configurations with overall good runtime performance, including rather competitive median runtimes, while minimizing median runtime yielded less robust parameter configurations that timed out on a large (but $< 50\%$) fraction of the benchmark instances.

However, when we encounter runs that do not terminate within the given cutoff time, we are unable to estimate the mean reliably; we only know a lower bound. In order to penalize timeouts, we define the *penalized average runtime (PAR)* of a set of runs with cutoff time κ to be the mean runtime over those runs, where unsuccessful runs are counted as $a \cdot \kappa$ with penalization constant $a \geq 1$. There is clearly more than one way of sensibly aggregating runtimes in the presence of capping. One reason we chose PAR is that it generalizes two other natural schemes;

1. "lexicographic": the number of instances solved, breaking ties by total runtime (used in the 2009 SAT competition[6]); and

2. total (or average) runtime across instances, treating timeouts as completed runs.

Scheme 1 is PAR with $a = \infty$, while Scheme 2 is PAR with $a = 1$. In this thesis, we use $a = 10$ throughout to emphasize the importance of timeouts more than in the second scheme, but to yield a more robust measure than the first scheme. KhudaBukhsh et al. (2009) studied different penalization constants for the comparison of various algorithms and found the relative rankings between them to be robust with respect to this choice.

There are two exceptions to our use of mean runtime in this thesis. Firstly, for historical reasons, in scenario SAPS-QWH we minimize SAPS median runtime, measured as the number of local search steps taken. Secondly, in the experiments with the global optimization algorithm CMA-ES, we followed the example of Bartz-Beielstein et al. (2008), measuring the solution cost of each CMA-ES run as the minimal function value found in a given number of function evaluations. We minimized the mean of this solution cost over repeated runs.

3.5 Configuration Scenarios

We now define the algorithm configuration scenarios we use throughout this thesis. These scenarios are organized into seven sets. We describe them in increasing complexity in the following sections.[7]

[6] http://www.satcompetition.org/2009/spec2009.html

[7] A few additional configuration scenarios are only used very locally and we defer their respective discussion to the points at which we use them: Section 5.4 for scenarios using SAT4J and GLS$^+$ and Section 8.2 for the

Scenario	Algorithm	Instance	κ_{max} [s]	# runs of target algo allowed
SAPS-QWH	SAPS	QWH	1	20 000
CMAES-ACKLEY	CMA-ES	Ackley	–	1 000
CMAES-GRIEWANGK	CMA-ES	Griewangk	–	1 000
CMAES-RASTRIGIN	CMA-ES	Rastrigin	–	1 000
CMAES-SPHERE	CMA-ES	Sphere	–	1 000

Table 3.3: Summary of our BLACKBOXOPT configuration scenarios. For the CMA-ES scenarios, the optimization objective to be minimized is average solution cost; for SAPS-QWH it is median runtime, measured in search steps. The CMA-ES scenario did not have a cutoff time. Rather, CMA-ES had a budget of 1 000 to 10 000 function evaluations—see Table 3.4.

Test function	Dimensionality	Initial point	# function evaluations CMA-ES was allowed
Sphere	10	$[10, \ldots, 10]^\intercal \in \mathbb{R}^{10}$	1 000
Ackley	10	$[20, \ldots, 20]^\intercal \in \mathbb{R}^{10}$	1 000
Griewangk	10	$[20, \ldots, 20]^\intercal \in \mathbb{R}^{10}$	10 000
Rastrigin	10	$[20, \ldots, 20]^\intercal \in \mathbb{R}^{10}$	10 000

Table 3.4: Experimental setup for the CMA-ES test cases.

3.5.1 Set of Configuration Scenarios BLACKBOXOPT

Our conceptually-easiest scenarios deal with the configuration of algorithms with only continuous parameters for single problem instances. This is equivalent to the problem of blackbox function optimization with noisy responses (where noise is due to randomness in the algorithm and is typically not Gaussian distributed). In blackbox function optimization, the available budget is typically defined in terms of a number of function evaluations. In our case, this corresponds to the number of target algorithm runs the configurators are allowed.

To evaluate algorithms following this paradigm (which we do in Chapters 9 and 10), we use our set of configuration scenarios BLACKBOXOPT, summarized in Table 3.3. This set contains five scenarios, each of which deals with the optimization of four continuous algorithm parameters for a single instance, given a budget on the number of target algorithm runs. In the first scenario, the objective is to minimize median SAPS runtime on instance QWH. The remaining four scenarios deal with the configuration of CMA-ES (on the four test functions defined in Section 3.3.3). The optimization objective for those scenarios is to minimize average solution cost.

In these CMA-ES experiments, we used different starting points and allowed CMA-ES different budgets. Following Bartz-Beielstein et al. (2008), for the Sphere function, we initialized CMA-ES at the point $[10, \ldots, 10]^\intercal \in \mathbb{R}^{10}$. To test global search performance, in the other three test functions, we initialized CMA-ES further away from the optima, at the point $[20, \ldots, 20]^\intercal \in \mathbb{R}^{10}$. For the first two functions (Sphere and Ackley), we optimized mean solution quality reached by CMA-ES within 1 000 function evaluations, while for the latter two functions we set a limit of 10 000 function evaluations. This setup is described in Table 3.4. Since the slow implementation of one configuration procedure (SKO) could not

self-configuration of PARAMILS.

Scenario	Algorithm	Benchmark Set	κ_{max} [s]	Configuration Time [h]
SAPS-QWH	SAPS	QWH	1	0.5
SAPS-QCP-MED	SAPS	QCP_{med}	5	0.5
SAPS-QCP-Q075	SAPS	QCP_{q075}	5	0.5
SAPS-QCP-Q095	SAPS	QCP_{q095}	5	0.5
SAPS-SWGCP-MED	SAPS	SWGCP_{med}	5	0.5
SAPS-SWGCP-Q075	SAPS	SWGCP_{q075}	5	0.5
SAPS-SWGCP-Q095	SAPS	SWGCP_{q095}	5	0.5

Table 3.5: Summary of our SINGLEINSTCONT scenarios. For all scenarios except SAPS-QWH, the optimization objective is to minimize PAR. For SAPS-QWH it is to minimize median runtime, measured in number of search steps.

Scenario	Algorithm	Benchmark Set	κ_{max} [s]	Configuration Time [h]
SPEAR-IBM-Q025	SPEAR	IBM_{q025}	5	0.5
SPEAR-IBM-MED	SPEAR	IBM_{med}	5	0.5
SPEAR-SWV-MED	SPEAR	SWV_{med}	5	0.5
SPEAR-SWV-Q075	SPEAR	SWV_{q075}	5	0.5
SPEAR-SWV-Q095	SPEAR	SWV_{q095}	5	0.5

Table 3.6: Summary of our SINGLEINSTCAT scenarios. For all scenarios, the optimization objective is to minimize PAR.

handle as many as 1 000 runs of the target algorithm, we only allowed 200 CMA-ES runs in the experiments of Chapter 9 (where SKO was one of the procedures being evaluated). In the experiments reported in Chapter 10 we did not use SKO and so used a budget of 1 000 CMA-ES runs for configuration.

3.5.2 Set of Configuration Scenarios SINGLEINSTCONT

In our SINGLEINSTCONT configuration scenarios (used in Chapters 11 and 12), we continue to study the configuration of algorithms with only continuous parameters for single problem instances. In particular, in these scenarios SAPS is optimized on seven single SAT instances. Table 3.5 summarizes these scenarios. Note that, in contrast to the BLACKBOXOPT scenarios, the computational budget for configuration is now measured in CPU time. We used short time budgets of 30 minutes per run of the configurator in order to facilitate algorithm development.

3.5.3 Set of Configuration Scenarios SINGLEINSTCAT

In our SINGLEINSTCAT configuration scenarios (used in Chapter 12), we study the configuration of algorithms with only *categorical* parameters for single problem instances. In particular, in these scenarios we optimize SPEAR on five single instances. Table 3.6 summarizes these scenarios. As for set SINGLEINSTCONT, we used short configuration budgets of 30 minutes per run of the configurator in order to facilitate algorithm development.

Scenario	Algorithm	Benchmark Set	κ_{max} [s]	Configuration Time [h]
SAPS-QCP	SAPS	QCP	5	5
SAPS-SWGCP	SAPS	SWGCP	5	5
SPEAR-QCP	SPEAR	QCP	5	5
SPEAR-SWGCP	SPEAR	SWGCP	5	5
CPLEX-REGIONS100	CPLEX	Regions100	5	5

Table 3.7: Summary of our BROAD configuration scenarios. The optimization objective is to minimize penalized average runtime (PAR).

Scenario	Algorithm	Benchmark Set	κ_{max} [s]	Configuration Time [h]
SPEAR-IBM	SPEAR	BMC	300	48
SPEAR-SWV	SPEAR	SWV	300	48

Table 3.8: Summary of our VERIFICATION configuration scenarios. The optimization objective is to minimize penalized average runtime (PAR).

3.5.4 Set of Configuration Scenarios BROAD

Our set of BROAD configuration scenarios (used in Chapters 5 and 7 and Section 8.2) comprises quite heterogeneous scenarios dealing with the configuration of categorical parameters for multiple instances. Summarized in Table 3.7, these scenarios reach from the configuration of SAPS (whose 4 continuous parameters were discretized) to the configuration of CPLEX (with 63 parameters). The benchmark sets are also quite different. While instance hardness for the SAPS default on the QCP and SWGCP is spread across many orders of magnitude, the Regions100 set used for CPLEX is quite homogeneous. To facilitate algorithm development, we employed rather short configuration times of five hours. This is still ten times larger than for sets SINGLEINSTCONT and SINGLEINSTCAT, due to the extra complexity stemming from heterogeneity across instances.

3.5.5 Set of Configuration Scenarios VERIFICATION

Our VERIFICATION configuration scenarios, summarized in Table 3.8 and used in Chapter 6, comprise two scenarios dealing with the configuration of SPEAR for the industrial benchmark sets BMC and SWV. Since the instances in these sets are very hard (at least for the algorithm defaults), we set rather large cutoff times of 300 seconds per run and allowed a time budget of two days for configuration.

3.5.6 Set of Configuration Scenarios CPLEX

Our CPLEX configuration scenarios, summarized in Table 3.8, serve in our study of configuring CPLEX in Section 8.1. As for set VERIFICATION, we set comparably large cutoff times of 300 seconds per run and allowed two days of configuration time.

Scenario	Algorithm	Benchmark Set	κ_{max} [s]	Configuration Time [h]
CPLEX-REGIONS200	CPLEX	Regions200	300	48
CPLEX-CLS	CPLEX	CLS	300	48
CPLEX-MJA	CPLEX	MJA	300	48
CPLEX-MIK	CPLEX	MIK	300	48
CPLEX-QP	CPLEX	QP	300	48

Table 3.9: Summary of our CPLEX configuration scenarios. The optimization objective is to minimize penalized average runtime (PAR).

Scenario	Algorithm	Benchmark Set	κ_{max} [s]
SATENSTEIN-QCP	SATENSTEIN	QCP	5
SATENSTEIN-SWGCP	SATENSTEIN	SWGCP	5
CPLEX-REGIONS100	CPLEX	Regions100	5
CPLEX-ORLIB	CPLEX	ORLIB	300
SPEAR-IBM	SPEAR	BMC	300
SPEAR-SWV	SPEAR	SWV	300

Table 3.10: Summary of our COMPLEX configuration scenarios. The optimization objective is to minimize penalized average runtime (PAR).

3.5.7 Set of Configuration Scenarios COMPLEX

Our set of COMPLEX configuration scenarios, summarized in Table 3.10, is a heterogeneous collection of scenarios, comprising the configuration of the most complex target algorithms we have experimented with. It is used in Chapter 4 to study the many ways in which configuration scenarios can differ. Note that we did not actually run configuration procedures for these scenarios in Chapter 4. Rather, we empirically analyzed the performance of 1 000 randomly-sampled configurations.

3.6 Experimental Preliminaries

Here, we summarize experimental preliminaries for the remainder of the thesis.

3.6.1 Selecting Instances and Seeds

The comparison of empirical cost statistics of two parameter configurations is inherently noisy. In order to reduce the variance in these comparisons, following common practice (see, *e.g.*, Birattari et al., 2002; Ridge and Kudenko, 2006), we blocked on instances. That is, we ensured that whenever two parameter configurations were compared, both were run on exactly the same instances. This serves to avoid noise effects due to differences between instances. For example, it prevents us from making the mistake of considering configuration $\boldsymbol{\theta}$ to be better than configuration $\boldsymbol{\theta}'$ just because $\boldsymbol{\theta}$ was tested on easier instances. For randomized algorithms we also blocked on random seeds to avoid similar noise effects.

When dealing with randomized target algorithms, there is also a tradeoff between the

Symbol	Meaning
$\boldsymbol{\theta}_{inc}(t)$	Incumbent parameter configuration at time t
PAR	Penalized average runtime (of a configuration on a benchmark set): mean runtime, counting timed-out runs at κ_{max} as $10 \cdot \kappa_{max}$
$p_{t,train}$	Training performance of $\boldsymbol{\theta}_{inc}(t)$ (PAR of $\boldsymbol{\theta}_{inc}(t)$ on used training instances)
$p_{t,test}$	Test performance of $\boldsymbol{\theta}_{inc}(t)$ (PAR of $\boldsymbol{\theta}_{inc}(t)$ on test instances). Also abbreviated as p_t

Table 3.11: Summary of performance measures for configurators.

number of problem instances used and the number of independent runs performed on each instance. In the extreme case—for a given sample size N—one could perform N runs on a single instance or a single run on N different instances. This latter strategy is known to result in minimal variance of the estimator for common optimization objectives, such as minimization of mean runtime (which we consider here) or maximization of mean solution quality (see, *e.g.*, Birattari, 2004). Consequently, we only performed multiple runs per instance when we wanted to acquire more samples of the cost distribution than there were instances in the training set.

Based on these considerations, the configuration procedures we study in this thesis have been implemented to take a list of ⟨instance, random number seed⟩ pairs as one of their inputs. Empirical estimates $\hat{c}_N(\boldsymbol{\theta})$ of the cost measure $c(\boldsymbol{\theta})$ to be optimized were determined from N ⟨instance, seed⟩ pairs in that list (in Part III of this thesis always from the *first* N pairs in that list). Each list was constructed as follows. Given a training set consisting of M problem instances, for $N \leq M$, we drew a sample of N instances uniformly at random and without replacement and added them to the list. If we wished to evaluate an algorithm on more samples than we had training instances, that is, in the case $N > M$ (which only is allowed for randomized algorithms), we repeatedly drew random samples of size M as described before. Each such batch corresponded to a random permutation of the N training instances; we then added a final sample of size $N \bmod M < M$, as in the case $N \leq M$. As each instance was drawn, it was paired with a random seed that was chosen uniformly at random from the set of all possible seeds (here, $\{1, \ldots, 2^{31} - 1\}$) and added to the list of ⟨instance, seed⟩ pairs.

3.6.2 Comparison of Configuration Procedures

We measure a configurator's performance given a time budget t by evaluating the performance of its incumbent configuration at time t, denoted $\boldsymbol{\theta}_{inc}(t)$. With few exceptions (which we explicitly state), our objective function throughout this thesis is penalized average runtime, PAR, with penalization constant 10. We denote training and test performance (as defined in Section 1.2.2) at time t as $p_{t,train}$ and $p_{t,test}$, respectively. Table 3.11 summarizes this notation for easy reference.

Since the choice of instances (and to some degree of seeds) is very important for the final outcome of the optimization, in our experimental evaluations we always performed a number of independent runs of each configuration procedure (typically 25) and report summary statistics of $p_{t,train}$ and $p_{t,test}$ across the 25 runs. We created a separate list of instances and

seeds for each run as explained above, where the kth run of each configuration procedure uses the same kth list of instances and seeds. (Note, however, that test set performance was measured on the same set for all runs.)

We performed two types of statistical tests to compare the (training or test) performance of two configuration procedures. In cases, where the kth run of each algorithm shared important external settings, we performed a two-sided paired Max-Wilcoxon test with the null hypothesis that there was no difference in the performances, considering p-values below 0.05 to be statistically significant. In cases where such pairings did not apply we used the (unpaired) Mann-Whitney U test. The p-values reported in all tables were derived using these tests; p-values shown in parentheses refer to cases where the procedure we expected to perform better actually performed worse.

3.6.3 Reference Machines

We carried out almost all of our experiments on a cluster of 55 dual 3.2GHz Intel Xeon PCs with 2MB cache and 2GB RAM, running OpenSuSE Linux 10.1, measuring runtimes as CPU time on these reference machines. The exception were experiments with configuration procedures other than our own that required the Windows operating system. For these experiments, local to Section 5.4 and Chapter 9, we ran the original configurator under Windows XP but still executed all target algorithm runs on the machines stated above. This was achieved by a wrapper script that connected to one of the above cluster nodes, performed the target algorithm run there and returned the result of the run. Thus, only the overhead times of these configurators are affected by the different computing environment; all runtimes of the target algorithms are for the above cluster nodes.

3.6.4 Implementation

There are two types of overhead in algorithm configuration procedures. The first type is implementation-specific and could be reduced with better-engineered code. For example, the overhead of calling a target algorithm on the command line can be substantial if performing tens of thousands of runs. It is about 0.1 seconds per algorithm run in our Ruby implementation of the model-free configuration framework in Part III and about 0.2 seconds for our Matlab implementation of the model-based configuration framework in Part IV. We do not include these overheads when we report the runtime of our methods since they could be drastically reduced by using a more native language. We also omit other overheads due to the use of Ruby and Matlab. For example, in "easy" configuration scenarios, where most algorithm runs finish in milliseconds, the overhead of our use of Ruby scripts can be substantial. Indeed, the configuration run with the largest overhead we observed took 24 hours to execute five hours worth of target algorithm runtime. In contrast, for scenarios with longer target algorithm runtimes we observed virtually no overhead.

The second type of overhead is due to the computational complexity of our approaches. While this complexity is negligible in model-free optimization, model-based optimization incurs substantial overheads from learning response surface models and optimizing the expected improvement criterion. We strove to keep the constants low by implementing computationally-

expensive bottleneck procedures in MEX—C code callable from Matlab. We *do* include the remaining overhead of these procedures when computing the runtime of a configuration procedure since it is not straightforward to reduce.

3.7 Chapter Summary

In this chapter, we defined a variety of configuration scenarios (instances of the algorithm configuration problem) we use in our experiments thoughout this thesis. The main components of a configuration scenario are a target algorithm, a set of benchmark instances, and an optimization objective. Our main target algorithms are state-of-the-art tree search and local search solvers for propositional satisfiability (SAT), the commercial optimization tool CPLEX for mixed integer programming (MIP), and CMA-ES for continuous blackbox optimization. We used prominent benchmark instances that have been previously used to assess the performance of these target algorithms. As our optimization objective, in most cases we chose average penalized runtime (PAR), counting timed-out runs at a cutoff time κ_{max} as $a \cdot \kappa_{max}$; we set $a = 10$ throughout the thesis.

We grouped configuration scenarios into seven sets, in increasing complexity: BLACKBOX-OPT, SINGLEINSTCONT, SINGLEINSTCAT, BROAD, VERIFICATION, CPLEX, and COMPLEX. Whenever we compare configuration procedures in this thesis, that comparison is based on one of these sets of scenarios.

Finally, we discussed experimental preliminaries. We described how we selected instances and seeds to enable blocking in our configuration procedures, how we compared configuration procedures (based on the test performance achieved in multiple independent runs of the configurators, using statistical tests), which computational environment we used, and some implementation details.

Chapter 4

Empirical Analysis of Algorithm Configuration Scenarios Based on Random Sampling

I think that in the discussion of natural problems we ought to begin not with the Scriptures, but with experiments, and demonstrations.
—Galileo Galilei, Italian physicist and astronomer

As stated in the introduction, algorithm configuration is hard for two reasons: first, there can be an enormous number of candidate parameter configurations; and second, empirically determining the performance of even a single configuration on a large set of non-trivial instances of an $\mathcal{NP}$-hard problem can be costly. Thus, it is necessary to make careful choices about (1) the number of configurations to consider and (2) the number of problem instances to use in evaluating them. Furthermore, it is ideally desirable to avoid prematurely terminating any run, or "censoring" it's runtime. (This issue arises particularly in the context of configuring algorithms with the goal of minimizing the runtime required for solving a given problem instance or reaching a certain solution quality.) Again, however, time constraints can make this impractical, requiring us to choose (3) some *captime* at which runs will be terminated whether or not they have completed.

In this chapter[1], we empirically study the tradeoff between these three dimensions of algorithm configuration. (2) and (3) are very related to the second and third dimensions of algorithm configuration presented in Section 1.2.2. Here, we study these dimensions in the context of a very simple instantiation of the first dimension: selecting the sequence of configurations to be evaluated uniformly at random.

4.1 Introduction

Automated methods for performing parameter optimization and algorithm configuration can be understood as sophisticated heuristics for deciding which configurations to consider and

[1]This chapter is based on joint work with Holger Hoos and Kevin Leyton-Brown about to be submitted for publication (Hutter et al., 2009d).

how many instances to use for their evaluation.[2] Racing algorithms (Maron and Moore, 1994; Birattari et al., 2002; Birattari, 2005; Balaprakash et al., 2007) emphasize using as few problem instances as possible to reliably choose among a fixed set of parameter configurations. More specifically, they incrementally expand the instance set (i.e., perform more runs for all configurations) and at each step eliminate configurations that are statistically significantly worse than others in terms of observed performance. In contrast, research on sequential search algorithms focuses on the question of *which* parameter configurations to evaluate. Many search algorithms, such as Multi-TAC (Minton, 1996), Calibra (Adenso-Diaz and Laguna, 2006), the mesh adaptive direct search algorithm (Audet and Orban, 2006) and BASICILS (Hutter et al., 2007b, see also Section 5.2), use a fixed, user-defined instance set. Other search algorithms include mechanisms for adapting the set of instances used for evaluating parameter configurations; examples are Composer (Gratch and Dejong, 1992), SPO (Bartz-Beielstein, 2006, see also Section 9.5) and FOCUSEDILS (Hutter et al., 2007b, see also Section 5.3).

The literature on automatic algorithm configuration places less emphasis on the choice of captime.[3] However, the issue has been studied in the context of evaluating heuristic algorithms. Segre et al. (1991) demonstrated that small captimes can lead to misleading conclusions when evaluating explanation-based learning algorithms. Etzioni and Etzioni (1994) extended statistical tests to deal with partially-censored runs in an effort to limit the large impact of captimes observed by Segre et al. (1991). Simon and Chatalic (2001) demonstrated the relative robustness of comparisons between SAT solvers for three different captimes.

Here, we present the first detailed empirical study of the role of this captime in algorithm configuration. We show that the impact of captime is similar to that of the size of the instance set based upon which configurations are compared. Large captimes lead to unreasonable time requirements for evaluating single parameter configurations (especially poor ones); this typically limits the number of problem instances on which configurations are evaluated, which in turn can lead to misleading performance results. On the other hand, evaluations based on overly aggressive captimes favour parameter configurations with good initial performance, and thus the configurations chosen on the basis of these evaluations may not perform competitively when allowed longer runs.

In this chapter, we do not yet propose a new method for making choices about which parameter configurations to explore, which benchmark set to use, or how much time to allocate to each run of the target algorithm. Rather, we first study the many ways in which algorithm configuration scenarios differ. This study is based on an empirical analysis approach we propose for investigating the tradeoffs between the choices *any* (manual or automated) approach to algorithm configuration must make.

Based on the data that we analyze in this chapter, we can answer a number of questions about a given configuration scenario that are important for both manual and automated algorithm configuration.[4] Here, we focus on the following eight questions:

[2] Of course, essentially the same point can be made about manual approaches, except that they are typically less sophisticated, and they have been discussed less rigorously in the literature.

[3] The only exception of which we are aware is our own recent extension of the PARAMILS framework, which dynamically adapts the per-run cutoff time (Hutter et al., 2009c). We discuss this adaptive capping mechanism in Chapter 7.

[4] We note, however, that our methods are typically very costly; our methods are applied only post hoc, not

1. How much does performance vary across parameter configurations?
2. How large is the variability in hardness across benchmark instances?
3. Which benchmark instances are useful for discriminating between parameter configurations?
4. Are the same instances "easy" and "hard" for all or most configurations?
5. Given a fixed computational budget and a fixed captime, how should we trade off the number of configurations evaluated *vs* the number of instances used in these evaluations?
6. Given a fixed computational budget and a fixed number of instances, how should we trade off the number of configurations evaluated *vs* the captime used for each evaluation?
7. Given a budget for identifying the best of a fixed set of parameter configurations, how many instances, N, and which captime, κ, should be used for evaluating each configuration?
8. Likewise, how should we trade off N and κ if the goal is to *rank* a fixed set of parameter configurations?

Our experimental analysis approach allows us to answer each of these questions for each of our six COMPLEX configuration scenarios. Throughout, we discuss these answers in detail for two rather different scenarios (CPLEX-REGIONS100 and SPEAR-IBM), and summarize our findings for the others. Overall, our experimental analysis demonstrates that the configuration scenarios we studied differ in important ways: the answers to the questions above differ widely across scenarios. As a consequence, in later chapters of this thesis we will develop *adaptive* methods for selecting captime and number of training instances in a data-driven way.

After covering experimental preliminaries in the next section, we introduce our empirical analysis approach. In Section 4.3, we investigate distributions of instance hardness, quality of parameter configurations and the interaction between the two, thereby answering Questions 1–4 above. In Section 4.4 we present an empirical study of the tradeoffs between the number of instances used and the captime, if the objective is to identify the best configuration $\boldsymbol{\theta} \in \boldsymbol{\Theta}$; this serves to answer Questions 5–7. In Section 4.5, we investigate the same tradeoffs in the context of the problem of *ranking* parameter configurations (or algorithms) in order to answer Question 8.

4.2 Gathering a Matrix of Runtimes

In this chapter, we study the COMPLEX configuration scenarios defined in Section 3.5.7. These scenarios include the configuration of SATENSTEIN (local search for SAT), SPEAR (tree search for SAT), and CPLEX (tree search for MIP). For detailed descriptions of these algorithms, see Section 3.2. For each of these algorithms, we study two benchmark distributions, resulting in the six COMPLEX configuration scenarios: SATENSTEIN-QCP, SATENSTEIN-SWGCP, SPEAR-IBM, SPEAR-SWV, CPLEX-REGIONS100, and CPLEX-ORLIB. The optimization objective in these configuration scenarios is penalized average runtime (PAR, which counts timed-out runs at captime κ as $10 \cdot \kappa$; see Section 3.4). We note, however, that all methods we study in

online.

Configuration Scenario	κ_{max} [s]	Default	Best known	Best sampled
SATENSTEIN-SWGCP	5	21.02	0.035 (from KhudaBukhsh et al., 2009)	0.043
SATENSTEIN-QCP	5	10.19	0.17 (from KhudaBukhsh et al., 2009)	0.21
SPEAR-IBM	300	1393	795 (from Hutter et al., 2007a)	823
SPEAR-SWV	300	466	1.37 (from Hutter et al., 2007a)	1.90
CPLEX-REGIONS100	5	1.76	0.32 (from Hutter et al., 2009b)	0.86
CPLEX-ORLIB	300	74.4	74.42 (CPLEX default)	54.1

Table 4.1: Quality of default, best-known, and best randomly-sampled configuration. For each of our six algorithm configuration scenarios, we give penalized average runtime (PAR) of the default, of the best known domain-specific parameter configuration (including its source), and of the best of our randomly sampled configurations.

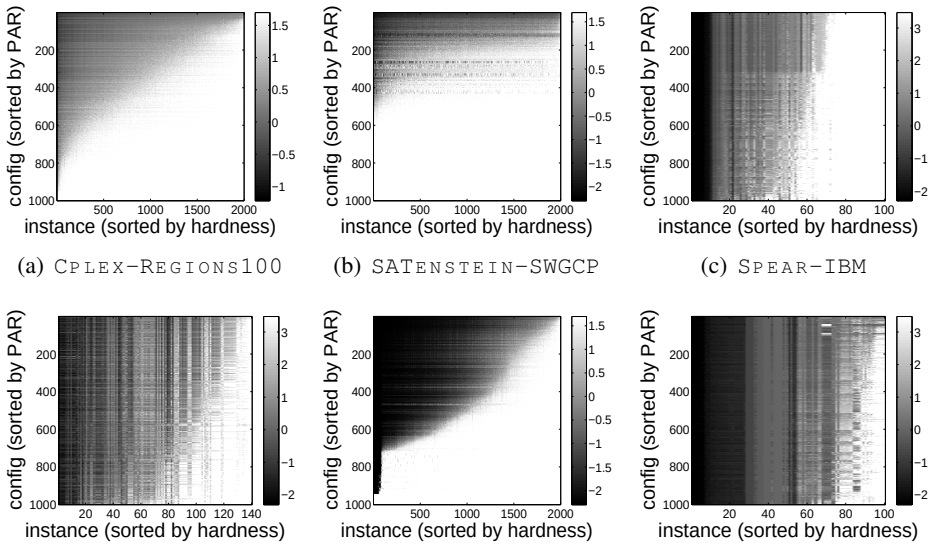
this chapter are also well defined and meaningful under other configuration objectives, such as median runtime and more complicated measures, such as, for example, the SAT competition scoring function.[5] When estimating PAR of a parameter configuration based on N runs, we only perform a single run for each of N problem instances since this yields an estimator with minimal variance for a given sample size N; see our discussion in Section 3.6.1. We also use a blocking scheme as described in that section, using the same N instances and seeds to evaluate each configuration.

In order to analyze the enormous parameter configuration spaces of SATENSTEIN, SPEAR and CPLEX in an unbiased way, we sampled parameter configurations by instantiating all free parameters at random. One may wonder how well these randomly-sampled configurations actually perform: after all, in many other optimization problems, sampling candidate solutions at random could yield very poor solutions. As it turns out, this is not the case in the algorithm configuration scenarios we consider here. In Table 4.1, for each scenario we compare the performance of the algorithm default, the best known configuration for the scenario, and the best out of 999 randomly sampled configurations. In all scenarios, the best randomly sampled configuration performed better than the default, typically by a substantial margin. For the SATENSTEIN and the SPEAR scenarios, its performance was close to that of the best known configuration, while for CPLEX-REGIONS100 the difference was somewhat larger. (For ORLIB, we are not aware of any published parameter setting of CPLEX and thus only compared to the CPLEX default.)

The empirical analysis approach for the study of algorithm configuration scenarios we propose in this chapter is based on very simple input data: a $M \times P$ matrix containing the performance of M parameter configurations on a set of P problem instances. Here, we used the algorithm default plus $M - 1 = 999$ random configurations. However, our methods are not limited to random configurations. Parameter configurations may also be obtained from trajectories of automated configuration procedures, or manually. Indeed, our empirical analysis approach does not even require that all configurations are instantiations of a single algorithm; it also works for entirely different *algorithms*. To demonstrate this, in Section 4.5, we use ten solvers from a recent SAT competition as "configurations".

For the comparably easy instances in configuration scenarios SATENSTEIN-QCP,

[5] See http://www.satcompetition.org/2007/rules07.html and http://www.satcompetition.org/2009/spec2009.html.



(a) CPLEX-REGIONS100 (b) SATENSTEIN-SWGCP (c) SPEAR-IBM

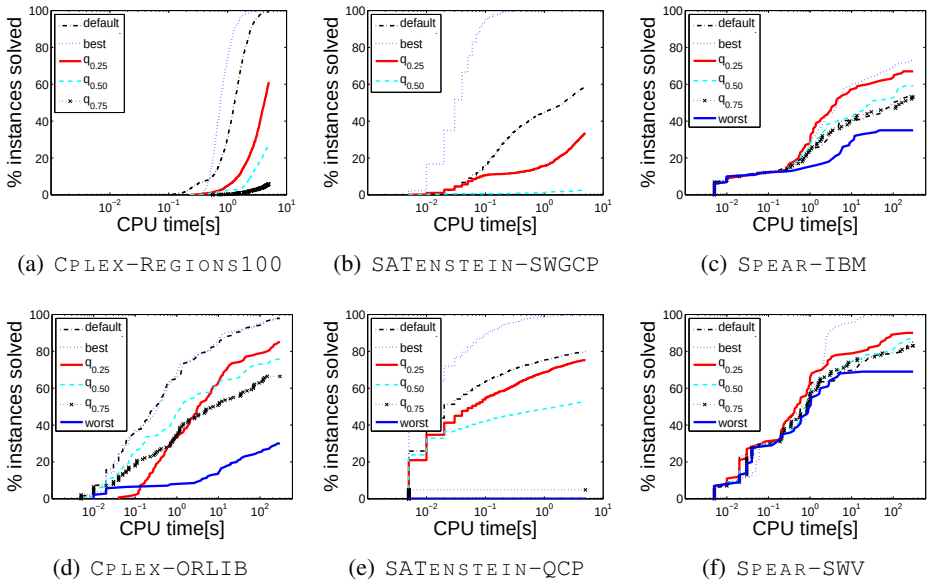
(d) CPLEX-ORLIB (e) SATENSTEIN-QCP (f) SPEAR-SWV

Figure 4.1: Raw data: matrix of runtime of each of the $M = 1\,000$ sampled parameter configurations on each of the P instances. Each dot in the matrix represents the runtime of a configuration on a single instance. Darker dots represent shorter runtimes; The colour scale is logarithmic with base 10. Configurations are sorted by their PAR score across all P instances. Instances are sorted by hardness (mean runtime of the M configurations, analogous to PAR counting runs that timed out at captime κ as $10 \cdot \kappa$).

SATENSTEIN-SWGCP and CPLEX-REGIONS100, we evaluated each of our $M = 1\,000$ configurations on $P = 2\,000$ instances, terminating unsuccessful runs after a captime of $\kappa_{max} = 5$ seconds. Scenarios SPEAR-IBM, SPEAR-SWV and CPLEX-ORLIB contain much harder instances, and we thus used a captime of $\kappa_{max} = 300$ seconds and benchmark instance sets of size $P = 140$, $P = 100$ and $P = 100$, respectively. Gathering the data for the input matrices in this chapter took around 1.5 CPU months for each of the three scenarios with $\kappa_{max} = 5$ seconds, one CPU month for SPEAR-SWV, and 2.5 CPU months for each of SPEAR-IBM and CPLEX-ORLIB. (For information on machines used, see Section 3.6.3.)

4.3 Analysis of Runtime Variability across Configurations and Instances

In this section, we provide an overview of the interaction between configurations, instances, and how much time is allocated to each run. Figures 4.1 and 4.2 together give an overall description of this space. In Figure 4.1, we plot the raw data: the runtime for all combinations of instances and parameter configurations. In Figure 4.2 we give more detailed information about the precise runtime values for six configurations (the default, the best, the worst, and three



(a) CPLEX-REGIONS100 (b) SATENSTEIN-SWGCP (c) SPEAR-IBM

(d) CPLEX-ORLIB (e) SATENSTEIN-QCP (f) SPEAR-SWV

Figure 4.2: Hardness variation across all P instances in different algorithm configuration scenarios. For each scenario, the plot shows the percentage of benchmark instances solved by six parameter configurations (default, best and worst sampled configuration, and configurations at the $q_{0.25}$, $q_{0.50}$, and $q_{0.75}$ quantiles of quality across the sampled configurations) as a function of allowed time. In cases where a configuration did not solve *any* of the P instances, we do not show it in the figure.

quantiles), plotting a cumulative distribution of the percentage of benchmark instances solved by $\boldsymbol{\theta}$ as a function of time. Based on these plots, we can make four key observations about our configuration scenarios, providing answers to Questions 1–4 posed in the introduction.

Q1: How much does performance vary across parameter configurations?
The variability of quality across parameter configurations differed substantially across scenarios. For example, in CPLEX-REGIONS100, the worst sampled parameter configuration did not solve a single problem instance, while the best one solved all instances in less than five seconds (see Figures 4.1(a) and 4.2(a)). In contrast, the difference between worst and best configuration was much smaller for scenario SPEAR-IBM (see Figures 4.1(c) and 4.2(c)). The SATENSTEIN scenarios showed even larger variation across configurations than CPLEX-REGIONS100, while CPLEX-ORLIB and SPEAR-SWV showed less variation, comparable to SPEAR-IBM. We believe that scenarios with large performance variations across parameter configurations require algorithm configuration procedures that emphasize an effective search for good configurations. In other scenarios, an effective mechanism for selecting the best number of instances and cutoff time to use might be more important.

Q2: How large is the variability in hardness across benchmark instances?
The variability of hardness across benchmark instances also differed substantially between scenarios. For example, in scenario CPLEX-REGIONS100, there was "only" about an order of magnitude difference between a configuration's runtime on the easiest and the hardest instances (see Figure 4.2(a)). In contrast, this difference was at least five orders of magnitude for scenario SPEAR-IBM (see Figure 4.2(c)). Scenario SATENSTEIN-SWGCP was similar to scenario CPLEX-REGIONS100 in having small variability of instance hardness, while the other scenarios were more similar to scenario SPEAR-IBM in this respect. In some scenarios (e.g., SATENSTEIN-SWGCP; see Figure 4.2(b)), the difference in hardness between the easiest and the hardest instance depended on the parameter configuration, good configurations showing more robust performance across all instances.

Q3: Which benchmark instances are useful for discriminating between parameter configurations?
In some—but not all—scenarios, only a subset of instances was useful for distinguishing between parameter configurations. For example, in scenario CPLEX-REGIONS100 all instances were useful. In contrast, in scenario SPEAR-IBM, over 35% of the instances were infeasible for *all* considered configurations within the cutoff time of $\kappa_{max} = 300s$ (see Figure 4.1(c)), and were thus useless for comparing configurations. Similarly, about 10% of the instances in that scenario were trivially solvable for all configurations. Next to SPEAR-IBM, only SATENSTEIN-QCP and SPEAR-SWV had substantial percentages of trivially solvable instances. While uniformly easy instances do not pose a problem in principle (since they can always be solved quickly) they can pose a problem for automated configuration procedures that tend to evaluate configurations based on a few instances. For example, the performance of FOCUSEDILS (see Section 5.3) and SPO (see Section 9.5) could be expected to degrade if many trivial instances were added. On the other hand, uniformly *infeasible* instances pose a serious problem, both for manual and automated configuration methods: every algorithm run on such an instance costs valuable time without offering any information.

Q4: Are the same instances "easy" and "hard" for all or most configurations?
In some scenarios, the ranking was fairly stable across instances: i.e., the runtime of a parameter configuration, $\boldsymbol{\theta}$, on an instance, π, was largely determined by the overall quality of $\boldsymbol{\theta}$ (averaged across instances) and the overall hardness of π (averaged across configurations). This was approximately the case for scenario CPLEX-REGIONS100 (see Figure 4.1(a)), where better-performing configurations solved the same instances as weaker configurations, plus some additional ones. The ranking was also comparably stable in scenario SPEAR-IBM and SATENSTEIN-QCP. In contrast, in some scenarios we observed instability in the ranking of parameter configurations from one instance to another: whether or not a configuration $\boldsymbol{\theta}$ performed better on an instance π than another configuration $\boldsymbol{\theta}'$ depended on the instance π. One way this is evidenced in the matrix is as a "checkerboard pattern"; see, e.g., rows 400–900 and the two instance sets in columns 76–83 and 84–87 in Figure 4.1(f): configurations that did well on the first set of instances tended to do poorly on the second set and vice versa. Overall, the most pronounced examples of this behaviour were CPLEX-ORLIB and

SPEAR-SWV (see Figures 4.1(d) and 4.1(f), and the crossings of cumulative distributions in Figures 4.2(d) and 4.2(f).) Scenarios in which algorithm rankings are unstable across instances are problematic to address with both manual and automated configuration methods, because different instances often require very different mechanisms to be solved effectively. This suggests splitting such heterogeneous sets of instances into more homogeneous subsets, using portfolio techniques (Gomes and Selman, 2001; Horvitz et al., 2001; Xu et al., 2008), or using per-instance algorithm configuration (Hutter et al., 2006). However, note that in some cases (*e.g.*, scenario SPEAR-SWV) the instability between relative rankings is local to poor configurations, and it is possible to find a single good configuration that performs very well on all instances, limiting the potential for improvements by more complicated per-instance approaches.

4.4 Tradeoffs in Identifying the Best Configuration

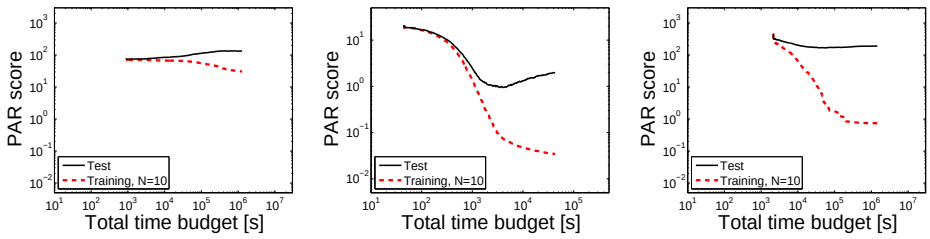
In this section, we study how we should trade off (a) the number of configurations evaluated, (b) the number of instances used in these evaluations and (c) the captime used for each evaluation when the objective is to identify the best parameter configuration.

4.4.1 Overconfidence and overtuning

One might imagine that without resource constraints but given a fixed set of instances from some distribution of interest, it would be easy to identify the best parameter configuration. Specifically, we could just evaluate every configuration on every instance, and then pick the best. This method indeed works for identifying the configuration with the best performance on the exact instances used for evaluation. However, when the set of instances is too small, the observed performance of the configuration selected may not be reflective of—and, indeed, may be overly optimistic about—performance on *other* instances from the same underlying distribution. We call this phenomenon *overconfidence*. This effect is notorious in machine learning, where it is well known that models fit on small datasets often generalize poorly (Hastie et al., 2001). Furthermore, generalization performance can actually *degrade* when the number of parameter configurations considered (in machine learning the *hypothesis space*) grows too large. This effect is called *overfitting* in machine learning (Hastie et al., 2001) and *overtuning* in optimization (Birattari et al., 2002; Birattari, 2005; Hutter et al., 2007b). In this section, we examine the extent to which overconfidence and overtuning can arise in our scenarios. Before doing so, we need to explain how we evaluate the generalization performance of a configuration.

Up to this point, we have based our analysis on the full input matrix. From now on, we partition instances into a *training set* Π and a *test set* Π'. We compute training performance and test performance at time step t as defined in Section 1.2.2.

To reduce variance, in this chapter we do not use fixed training and test sets, but instead average across many repetitions with different splits of the instances into training and test sets. We use an iterative sampling approach to estimate the *expected training and test performance* given a computational budget t, N training instances and captime κ. In each iteration, we draw a training set of N instances $\Pi \subseteq \Pi_{input}$ and start with an empty set of configurations



(a) CPLEX-ORLIB (b) SATENSTEIN-SWGCP (c) SPEAR-SWV

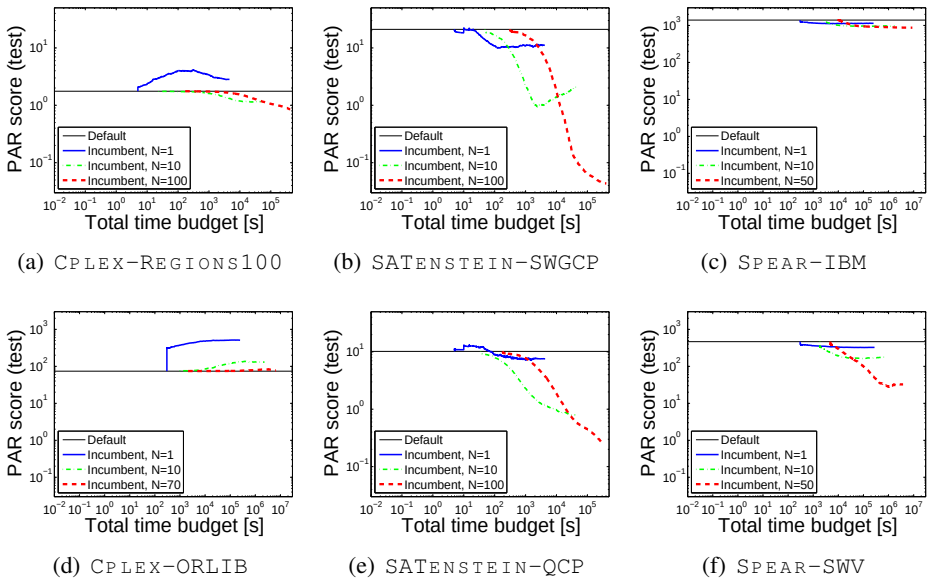
Figure 4.3: Overconfidence and overtuning for training sets of size $N = 10$. We plot training and test performance (penalized average runtime, PAR, for $N = 10$ training instances and $P/2$ test instances) of *IS-Best*$(N = 10, \kappa, t)$, where κ is the "full" captime of the configuration scenario ($\kappa = 300s$ for CPLEX-ORLIB and SPEAR-SWV, and $\kappa = 5s$ for SATENSTEIN-SWGCP).

$\boldsymbol{\Theta}_{train}$. We then expand $\boldsymbol{\Theta}_{train}$ by randomly adding elements of $\boldsymbol{\Theta}_{input} \setminus \boldsymbol{\Theta}_{train}$, until either $\boldsymbol{\Theta}_{train} = \boldsymbol{\Theta}_{input}$ or the time for evaluating all $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{train}$ on all instances $\pi \in \Pi_{train}$ with captime κ exceeds t. Finally, we evaluate training and test performance of $\boldsymbol{\Theta}_{train}$ based on Π_{train} and κ. In what follows, we always work with expected training and test performance, sometimes dropping the term "expected" for brevity. We calculate these quantities based on $K = 1\,000$ iterations, each of them using independently sampled disjoint training and test sets. We always use test sets of cardinality $|\Pi'| = P/2$, but vary the size of the training set Π_{train} from 1 to $P/2$. We refer to the resulting expected performance using a training set of N instances, captime κ, and as many configurations as can be evaluated in time t as *IS-Best*(N, κ, t), short for IterativeSampling-Best.

Using this iterative sampling approach, we investigated the difference between training and test performance for three scenarios in Figure 4.3. Based on training sets of size $N = 10$, we saw clear evidence for overconfidence (divergence between training and test performance) in these three scenarios, and evidence for overtuning (test performance that degrades as we increase the number of configurations considered) for CPLEX-ORLIB and SATENSTEIN-SWGCP. Of our six scenarios, the three shown gave rise to the most pronounced training/test performance gap. We believe that this occurred because of the relative instability of relative rankings of configurations across the respective instance sets (which we observed in Figures 4.1(b), 4.1(d), and 4.1(f)). Based on a small subset of instances, different parameter configurations performed best than for a large set of instances. Furthermore, for SPEAR-SWV the large percentage of trivial instances (see Figure 4.1(f)) effectively reduced the number of instances used for meaningful comparisons by about 50%.

4.4.2 Trading off number of configurations, number of instances, and captime

Now we investigate tradeoffs between the number of parameter configurations evaluated, the size of the benchmark set N, and the captime κ, given a time budget. This will serve to answer Questions 5–7 from the introduction.



(a) CPLEX-REGIONS100 (b) SATENSTEIN-SWGCP (c) SPEAR-IBM

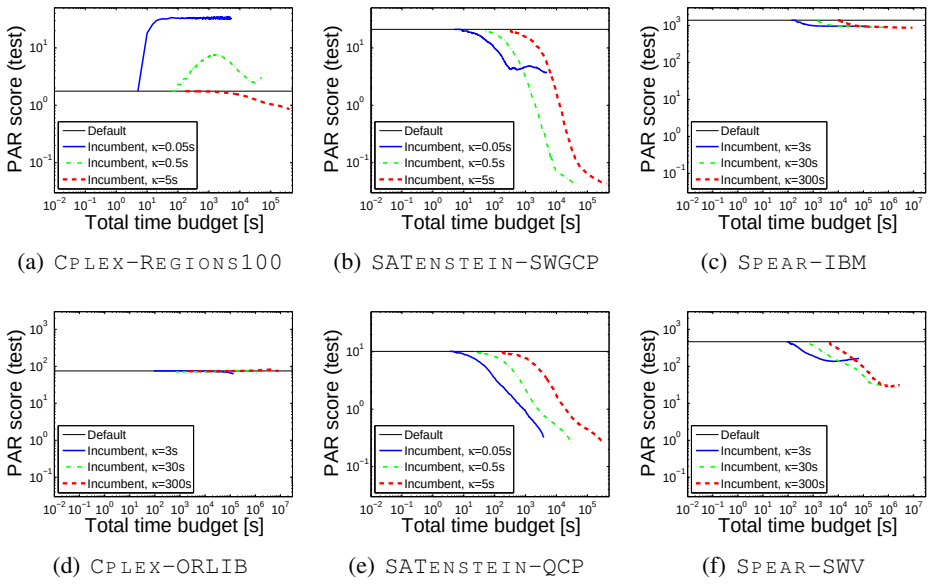
(d) CPLEX-ORLIB (e) SATENSTEIN-QCP (f) SPEAR-SWV

Figure 4.4: Test performance (PAR for $P/2$ test instances) of *IS-Best*(N, κ, t), where κ is the "full" captime of the configuration scenario ($\kappa = 300s$ for CPLEX-ORLIB and the SPEAR scenarios, and $\kappa = 5s$ for the rest); we plot graphs for $N = 1$, $N = 10$, and $N = \min(100, P/2)$. For reference, we plot test performance of the default.

Q5: Given a fixed computational budget and a fixed captime, how should we trade off the number of configurations evaluated* vs *the number of instances used in these evaluations?

To answer this question, we study how the performance of *IS-Best*(N, κ, t) progressed as we increased the time t for three different values of N and fixed $\kappa = \kappa_{max}$. In Figure 4.4, for each total amount of CPU time t available for algorithm configuration, we plot the performance arising from using each of these three values of N.[6] The optimal number of training instances, N, clearly depended on the overall CPU time available, and the impact of using different values of N differed widely across the scenarios. For example, for scenario CPLEX-REGIONS100 (see Figure 4.4(a)), using a single training instance ($N = 1$) yielded very poor performance throughout. For total time budgets t below about $5 \cdot 10^4$ seconds, the best tradeoff was achieved using $N = 10$. After that time, all $M = 1000$ configurations had been evaluated, but using $N = 100$ yielded better performance. For scenario SPEAR-IBM (see Figure 4.4(c)), the optimal tradeoff was quite different. For total time budgets below

[6]The plots for $N = 1$ and $N = 10$ end at the point where all of the M given parameter configurations have been evaluated. It would be appealing to extend the curves corresponding to lower values of N by considering more parameter configurations $\mathbf{\Theta}_{additional}$. Unfortunately, this is impossible without filling in our whole input matrix for the new configurations. This is because *IS-Best*(N, κ, t) averages across many different sets of N instances, and we would thus require the results of $\mathbf{\Theta}_{additional}$ for all instances. Furthermore, all curves are based on the same population of parameter configurations.



(a) CPLEX-REGIONS100 (b) SATENSTEIN-SWGCP (c) SPEAR-IBM

(d) CPLEX-ORLIB (e) SATENSTEIN-QCP (f) SPEAR-SWV

Figure 4.5: Test performance (PAR for $P/2$ test instances) of *IS-Best*$(\min(P/2, 100), \kappa, t)$ for various values of κ. We plot graphs for the "full" captime used in the scenario ($\kappa = 300s$ for CPLEX-ORLIB and the SPEAR scenarios, and $\kappa = 5s$ for the rest), a tenth of it, and a hundredth of it. For reference, we plot test performance of the default configuration.

3 000 seconds, using a single training instance ($N = 1$) actually performed best. For total time budgets between 3 000 and about 70 000 seconds, $N = 10$ yielded the best performance, and only for larger time budgets did $N = 100$ yield better results. For brevity, we do not discuss each scenario in depth, but rather summarize some highlights and general trends. In most scenarios, $N = 1$ typically led to poor test performance, particularly for scenario CPLEX-ORLIB, which showed very pronounced overtuning. $N = 10$ yielded good performance for scenarios that showed quite stable relative rankings of algorithms across instances, such as CPLEX-REGIONS100 and SATENSTEIN-QCP (see Figures 4.4(a) and 4.4(e)). In contrast, for scenarios where the relative ranking of algorithms depended on the particular subset of instances used, such as CPLEX-ORLIB and SPEAR-SWV, $N = 10$ led to overconfidence or even overtuning (see Figures 4.4(d) and 4.4(f)). For the very heterogeneous instance set in scenario CPLEX-ORLIB, even using $P/2 = 70$ instances led to slight overtuning, yielding a configuration worse than the default. This illustrates that even though the best sampled parameter configuration outperformed the CPLEX default, we could not actually *identify* this configuration based on a limited training set of 70 instances.

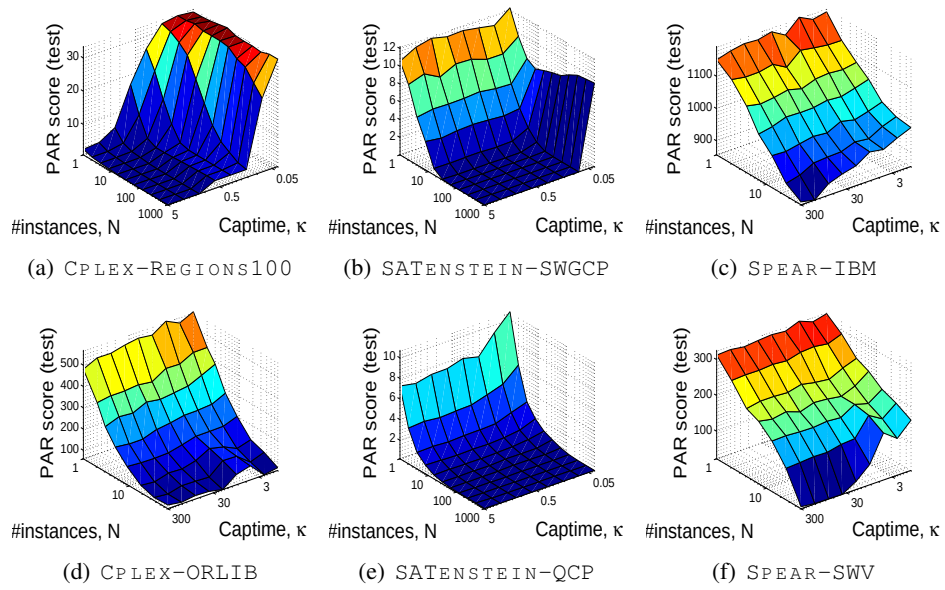
Q6: Given a fixed computational budget and a fixed number of instances, how should we trade off the number of configurations evaluated* vs *the captime used for each evaluation?

To answer this question, we study how the performance of *IS-Best*(N, κ, t) progressed as we

increased the time t for fixed $N = 100$ and three different values of κ (the "full" captime, κ_{max}, of the respective scenario, as well as $\kappa_{max}/10$ and $\kappa_{max}/100$). To our best knowledge, this constitutes the first detailed empirical investigation of captime's impact on the outcome of empirical comparisons between algorithms. In Figure 4.5, for each total amount of CPU time t available for algorithm configuration, we plot the performance arising from using each of these three values of κ. We observe that captime's impact depended on the overall CPU time available, and that this impact differed widely across the different scenarios. For scenario CPLEX-REGIONS100, the lowest captime, $\kappa_{max}/100$, performed extremely poorly, leading to extreme overtuning. Similarly, captime $\kappa_{max}/10$ led to overtuning, leaving the best choice to be κ_{max}, irrespective of the total time budget available. For scenario SPEAR-IBM, the situation was very different. The lowest captime, $\kappa_{max}/100$, performed extremely well and was the optimal choice for time budgets below $t = 10^4$ seconds. For time budgets between 10^4 seconds and $8 \cdot 10^5$ seconds, the optimal choice of captime was $\kappa_{max}/10$. Above that, all $M = 1\,000$ configurations had been evaluated, and using a larger captime of κ_{max} yielded better performance. Once again, we only summarize highlights of the other scenarios. SATENSTEIN-QCP is an extreme case of good performance with low captimes: using $\kappa_{max}/100$ yields very similar results as κ_{max}, at one-hundredth of the time budget. A similar effect is true for SATENSTEIN-SWGCP for captime $\kappa_{max}/10$. For scenario CPLEX-ORLIB, captime $\kappa_{max}/100$ actually seemed to result in *better* performance than larger captimes. We hypothesize that this is a noise effect related to the small number of instances and the instability in the relative rankings of the algorithms with respect to different instances. It is remarkable that for the SPEAR-IBM scenario, which emphasizes very hard instances (Zarpas, 2005), captimes as low as $\kappa_{max}/100 = 3s$ actually yielded good results. We attribute this to the relative stability in the relative rankings of algorithms across different instances in that scenario (see Figures 4.1(c) and 4.2(c)): parameter configurations that solved many instances quickly also tended to perform well when allowed longer runtimes.

Q7: Given a budget for identifying the best of a fixed set of parameter configurations, how many instances, N, and which captime, κ, should be used for evaluating each configuration?

To answer this question, we studied the performance of *IS-Best*(N, κ, ∞) for various combinations of N and κ. Figure 4.6 shows the test performance of these selected configurations. Unsurprisingly, given unlimited resources, the best results were achieved with the maximal number of training instances ($N = P/2$) and the maximal captime ($\kappa = 5s$), and performance degraded when N and κ decreased. However, *how much* performance degraded with lower N and κ differed widely across our configuration scenarios. For example, in scenario CPLEX-REGIONS100, with a budget of 100 seconds for evaluating each parameter configuration, much better performance could be achieved by using the full captime $\kappa_{max} = 5s$ and $N = 20$ instances than with other combinations, such as, for example, $\kappa = 0.5s$ $N = 200$. (To see this, inspect the diagonal of Figure 4.6(a) where $N \cdot \kappa = 300s$.) In contrast, in scenario SPEAR-IBM larger N were much more important, and, for example, with a budget of $1\,500$ seconds for evaluating each configuration, much better performance could be achieved by using all $N = 50$ training instances and $\kappa = 30s$ than with any other combination, such as, for



(a) CPLEX-REGIONS100 (b) SATENSTEIN-SWGCP (c) SPEAR-IBM

(d) CPLEX-ORLIB (e) SATENSTEIN-QCP (f) SPEAR-SWV

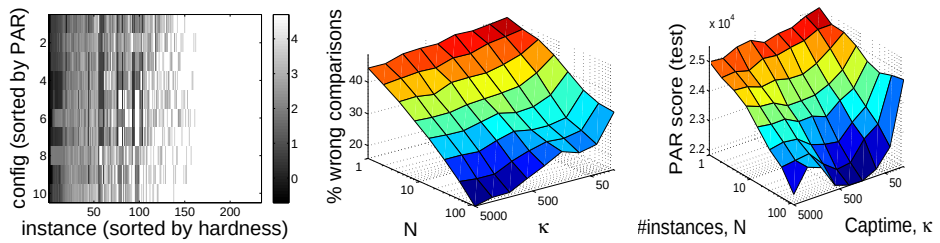
Figure 4.6: Test performance (PAR for $P/2$ test instances) of *IS-Best*(N, κ, ∞) as depending on the number of training instances, N, and the captime, κ, used. Note that each combination of N and κ was allowed to evaluate the same number of $M = 1\,000$ configurations, and that the time required for each combination was thus roughly proportionally to $N \cdot \kappa$.

example, $N = 5$ and $\kappa = 300s$ (inspect the diagonal of Figure 4.6(c) where $N \cdot \kappa = 1\,500s$). For the other scenarios, larger N was typically more important for good performance than larger κ. In particular, for some scenarios, reductions of κ to a certain point seemed to have no negative effect on performance at all. We note that it is quite standard in the literature for researchers to set captimes high enough to ensure that they will rarely be reached, and to evaluate fewer instances as a result. Our findings suggest that more reliable comparisons can often be achieved by inverting this pattern, evaluating a larger set of instances with a more aggressive cutoff.

4.5 Tradeoffs in Ranking Configurations

We now investigate the same tradeoff between number of training instances and captime studied in the previous section, but with the new objective of *ranking* parameter configurations rather than to simply choosing one as the best. This addresses Question 8 from the introduction.

Although not of immediate relevance to algorithm configuration, this problem arises in any comparative study of several parameter configurations—or indeed of several different *algorithms*—in which our interest is not restricted to the question of identifying the best-performing one.



(a) Raw data: result of 10 solvers on $P = 234$ instances

(b) % wrong pairwise comparisons based on N training instances and captime κ

(c) performance of "best" solver; "best" is determined based on N training instances and captime κ

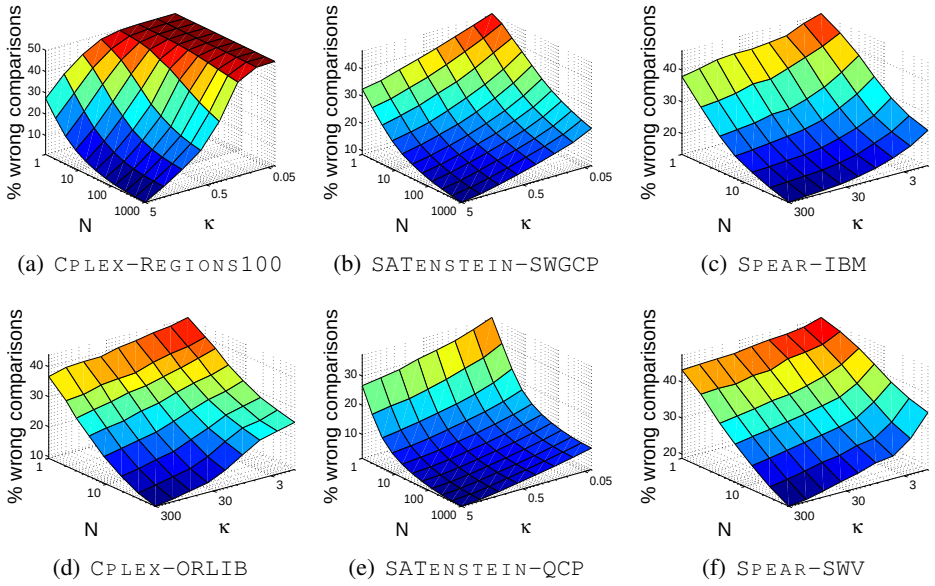
Figure 4.7: Reliability of comparisons between solvers for results from 2007 SAT competition (scoring metric: total runtime across instances). The 234 instances were split equally into training and test instances.

One prominent example of such a comparative study is the quasi-annual SAT competition[7], one purpose of which is to compare new SAT solvers with state of-the-art solvers. To demonstrate the versatility of our methods—and how those methods can be used in this expanded context—we obtained the runtimes of the ten finalists for the second phase of the 2007 SAT competition on the 234 industrial instances.[8] These runtimes constitute a matrix just like the input matrices we have used for our algorithm configuration scenarios throughout. Thus, we can employ our methods "out-of-the-box" to visualize the data (see Figure 4.7(a)); we observe a strong checker-board patterns indicating a lack of correlation between the solvers.

As before, we split the instances into training and test instances using a random permutation, and used the test instances solely to obtain an unbiased performance estimate. Beyond the characteristics evaluated before, we computed the percentage of "wrong" pairwise comparisons, that is, those with an outcome opposite than the one by a pairwise comparison based on the test set and the "full" captime of $\kappa = 5000$s per run. Figure 4.7(b) gives this percentage for various combinations of N and κ. We can see that large cutoff times were indeed necessary in the SAT competition to provide an accurate ranking of solvers. However, Figure 4.7(c) shows that much lower captimes would have been sufficient to identify a solver with very good performance. (In fact, in this case, test performance of algorithms selected based on captimes of around 300 seconds was *better* than that based on the full captime of 5000 seconds. Again, we hypothesize that this is related to the instability in the relative rankings of the algorithms with respect to different instances. We also observed multiple crossings in the cumulative distributions of the number of instances solved for the ten algorithms considered here.) This analysis demonstrates that if one wanted to run the SAT competition within one-tenth of the CPU budget, it would be better to impose a tighter captime than to drop instances.

[7] www.satcompetition.org

[8] http://www.cril.univ-artois.fr/SAT07/



(a) CPLEX-REGIONS100 (b) SATENSTEIN-SWGCP (c) SPEAR-IBM

(d) CPLEX-ORLIB (e) SATENSTEIN-QCP (f) SPEAR-SWV

Figure 4.8: Ratio of pairwise comparisons based on N training instances and captime κ whose outcome is different than the outcome based on $P/2$ test instances and the "full" captime of the configuration scenario ($\kappa = 300s$ for CPLEX-ORLIB and the SPEAR scenarios, and $\kappa = 5s$ for the rest).

Q8: Given a time budget, how should we trade off N and κ if the goal is to* rank *a fixed set of parameter configurations?

We applied the same method to the six COMPLEX configuration scenarios studied throughout this chapter, leading to the percentages of wrong comparisons shown in Figure 4.8. Comparing that plot to Figure 4.6, we note that when the objective is to obtain an accurate ranking, the optimal tradeoff tends to employ higher captimes than when the objective is to only identify the best algorithm. This is intuitive since only the best configurations solve instances within a low captime; the resulting runtime information is sufficient to determine the best configurations but not reliable rankings.

4.6 Chapter Summary

In this chapter, we introduced a novel, general empirical analysis approach to study algorithm configuration scenarios. The only input this empirical analysis approach requires is a matrix of runtimes for a set of parameter configurations (or, in fact a set of algorithms) and a set of instances. We demonstrated the generality of this approach by applying it to study our COMPLEX configuration scenarios and a set of runtime data available from the 2007 SAT competition.

From the empirical analysis presented in this chapter we gained valuable intuition about

the many ways in which algorithm configuration scenarios differ. Each of the eight questions we posed in the introduction of this chapter has different answers for the six COMPLEX configuration scenarios we studied here.

Most importantly, the optimal combination of fixed captime and number of training instances differs substantially across configuration scenarios. We thus conclude that these choices should be made *adaptively*, depending on the characteristics of the configuration scenario at hand. As a consequence, in Sections 5.3 and 10.3.1 we will describe adaptive methods for selecting the number of training instances to use for evaluating configurations. In Chapter 7, we will also introduce a general method for adaptively setting the captime.

Part III

Model-free Search for Algorithm Configuration

—in which we introduce a simple yet effective algorithm configuration framework and demonstrate its ability to configure the most complex algorithms of which we are aware

Chapter 5

Methods I: PARAMILS—Iterated Local Search in Parameter Configuration Space

Besides black art, there is only automation and mechanization.
—Federico Garcia Lorca, Spanish poet

In this part of the thesis, we focus on model-free search for algorithm configuration. Approaches in this category perform a search in parameter configuration space without relying on an explicit performance model. Many successful algorithm configuration approaches fall into this category (see Section 2.1.1). However, none of them are general enough to configure algorithms as complex as those we consider.

This part is organized as follows. First, we introduce the first general approach that scales to the configuration of algorithms with many categorical parameters (this chapter). Next, we report a case study for applying this approach to configure a state-of-the-art tree search algorithm for two sets of SAT-encoded industrial verification problems (Chapter 6). Then, we introduce a general method for adaptively setting the cutoff time to be used for each target algorithm run, thereby improving our approach further (Chapter 7). Finally, we demonstrate the applicability of the improved approach to configure a wide range of algorithms (Chapter 8).

In this chapter[1], we introduce a simple local search framework that provides one possible answer to the first dimension of algorithm configuration: which sequential search strategy should be used to select the parameter configurations $\vec{\boldsymbol{\Theta}}$ to be evaluated? We also provide one possible answer to the second dimension: how many runs should be performed to evaluate each configuration $\boldsymbol{\theta}$? For now, we fix the third dimension, using cutoff time κ_{max} for the evaluation of each parameter configuration.

In this chapter, we use the five BROAD configuration scenarios defined in Section 3.5.4: SAPS-SWGCP, SAPS-QCP, SPEAR-SWGCP, SPEAR-QCP, and CPLEX-REGIONS100. All these scenarios use short captimes of $\kappa_{max} = 5$ seconds per algorithm run and configuration times of 5

[1]This chapter is based on published joint work with Holger Hoos, Thomas Stützle, and Kevin Leyton-Brown (Hutter et al., 2007b, 2009c).

hours. We selected these comparably "easy" scenarios to facilitate an in-depth study of our algorithm components. Experiments for more challenging scenarios are discussed in Chapters 6 and 8.

Note that we introduce the components of our framework one piece at a time. Some of these pieces make an experimental difference that might appear small (albeit statistically significant). This is partly due to our measure of penalized average runtime (PAR), which de-emphasizes large differences: very large (potentially infinite) runtimes are counted as $10 \cdot \kappa_{max} = 50$ seconds, which limits the potential for improvements. However, with all components in place, our configuration procedures achieved substantial improvements; we defer this final evaluation to the end of Chapter 7 (Table 7.5 and Figure 7.5 on pages 112 and 113), after all components have been described. As we will see there, our configurators led to substantial improvements of our target algorithms even for our BROAD scenarios, with speedup factors between 2 and 3 540.

5.1 The PARAMILS framework

Consider the following manual parameter optimization process.

1. Begin with some initial parameter configuration;
2. Experiment with modifications to single parameter values, accepting new configurations whenever they result in improved performance;
3. Repeat step 2 until no single-parameter change yields an improvement.

This widely used procedure corresponds to a manually-executed local search in parameter configuration space. Specifically, it corresponds to an iterative first improvement procedure with a search space consisting of all possible configurations, an objective function that quantifies the performance achieved by the target algorithm with a given configuration, and a neighbourhood relation based on the modification of one single parameter value at a time (i.e., a "one-exchange" neighbourhood).

Viewing this manual procedure as a local search algorithm is advantageous because it suggests the automation of the procedure as well as its improvement by drawing on ideas from the stochastic local search community. For example, note that the procedure stops as soon as it reaches a local optimum (a parameter configuration that cannot be improved by modifying a single parameter value). A more sophisticated approach is to employ iterated local search (ILS, see Lourenço et al., 2002) to search for performance-optimizing parameter configurations. ILS is a stochastic local search method that builds a chain of local optima by iterating through a main loop consisting of (1) a solution perturbation to escape from local optima, (2) a subsidiary local search procedure and (3) an acceptance criterion that is used to decide whether to keep or reject a newly obtained candidate solution.

PARAMILS (outlined in pseudocode as Algorithm Framework 5.1) is an ILS method that searches parameter configuration space. It uses a combination of a given default and r random configurations for initialization, employs iterative first improvement as a subsidiary local search procedure, uses a fixed number, s, of random moves for perturbation, and always accepts better or equally-good parameter configurations, but re-initializes the search at random

Algorithm Framework 5.1: PARAMILS($\boldsymbol{\theta}_0, r, p_{restart}, s$)
Outline of iterated local search in parameter configuration space. The specific variants of PARAMILS we study, **BASICILS(N)** and **FOCUSEDILS**, are derived from this framework by instantiating procedure *better* (which compares $\boldsymbol{\theta}, \boldsymbol{\theta}' \in \boldsymbol{\Theta}$). BASICILS*(N)* uses *better*$_N$ (Procedure 5.3), while FOCUSEDILS uses *better*$_{Foc}$ (Procedure 5.5). Note that the incumbent, $\boldsymbol{\theta}_{inc}$, is a global variable that is updated whenever new target algorithm runs are performed in Procedure objective (see Procedure 7.1 on page 105; that function is called from the various *better* functions).

Input : Initial configuration $\boldsymbol{\theta}_0 \in \boldsymbol{\Theta}$, algorithm parameters r, $p_{restart}$, and s.
Output : Best parameter configuration $\boldsymbol{\theta}$ found.

for $i = 1, \ldots, r$ **do**
  $\boldsymbol{\theta} \leftarrow$ random $\boldsymbol{\theta} \in \boldsymbol{\Theta}$;
  if better*(*$\boldsymbol{\theta}, \boldsymbol{\theta}_0$*)* **then** $\boldsymbol{\theta}_0 \leftarrow \boldsymbol{\theta}$;
$\boldsymbol{\theta}_{ils} \leftarrow$ *IterativeFirstImprovement* ($\boldsymbol{\theta}_0$);
while not *TerminationCriterion()* **do**
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_{ils}$;
  // ===== Perturbation
  for $i = 1, \ldots, s$ **do**
    $\boldsymbol{\theta} \leftarrow$ random $\boldsymbol{\theta}' \in$ *Nbh*$(\boldsymbol{\theta})$;
  // ===== Basic local search
  $\boldsymbol{\theta} \leftarrow$ *IterativeFirstImprovement* ($\boldsymbol{\theta}$);
  // ===== AcceptanceCriterion
  if better*(*$\boldsymbol{\theta}, \boldsymbol{\theta}_{ils}$*)* **then**
    $\boldsymbol{\theta}_{ils} \leftarrow \boldsymbol{\theta}$;
  with probability $p_{restart}$ **do** $\boldsymbol{\theta}_{ils} \leftarrow$ random $\boldsymbol{\theta} \in \boldsymbol{\Theta}$;
return overall best $\boldsymbol{\theta}_{inc}$ found;

Procedure 5.2: IterativeFirstImprovement($\boldsymbol{\theta}$)
The neighbourhood *Nbh*$(\boldsymbol{\theta})$ of a configuration $\boldsymbol{\theta}$ is the set of all configurations that differ from $\boldsymbol{\theta}$ in one parameter, excluding configurations differing in a conditional parameter that is not relevant in $\boldsymbol{\theta}$.

Input : Starting configuration, $\boldsymbol{\theta} \in \boldsymbol{\Theta}$
Output : Locally optimal configuration, $\boldsymbol{\theta}$.

repeat
  $\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}$;
  foreach $\boldsymbol{\theta}'' \in Nbh(\boldsymbol{\theta}')$ *in randomized order* **do**
    if better*(*$\boldsymbol{\theta}'', \boldsymbol{\theta}'$*)* **then** $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}''$; **break;**
until $\boldsymbol{\theta}' = \boldsymbol{\theta}$*;*
return $\boldsymbol{\theta}$;

with probability $p_{restart}$.[2] PARAMILS is based on a one-exchange neighbourhood, that is, we

[2]Our original parameter choices $\langle r, s, p_{restart} \rangle = \langle 10, 3, 0.01 \rangle$ (from Hutter et al., 2007b) were somewhat arbitrary, though we expected performance to be quite robust with respect to these settings. We revisit this issue in

always consider changing only one parameter at a time. To deal with conditional parameters (whose setting is only relevant if some higher-level parameters take on certain values), it uses a neighbourhood of each configuration $\boldsymbol{\theta}$ that only contains configurations which differ in a parameter that is relevant in $\boldsymbol{\theta}$.

Since PARAMILS performs an iterated local search using a one-exchange neighbourhood, it is very similar in spirit to local search methods for other problems, such as SAT (Selman et al., 1992; Hoos and Stützle, 2000; Schuurmans and Southey, 2001), CSP (Minton et al., 1992), and MPE (Kask and Dechter, 1999; Hutter, 2004). Since PARAMILS is a local search method, existing theoretical frameworks (see, *e.g.*, Hoos, 2002b; Mengshoel, 2008), could in principle be used for its analysis. The main factor distinguishing our problem from those faced by "standard" local search algorithms is the stochastic nature of our optimization problem (for a discussion of local search for stochastic optimization, see, *e.g.*, Spall, 2003). Furthermore, there exists no compact representation of the objective function that could be used to guide the search. To illustrate this, consider local search for SAT, where the candidate variables to be flipped can be limited to those occurring in currently-unsatisfied clauses. In general algorithm configuration, on the other hand, such a mechanism cannot be used because of the problem's blackbox nature: the only information available about the target algorithm is its performance with the combinations of parameter configurations, instances, and seeds considered so far. While, obviously, other (stochastic) local search methods could be used as the basis for algorithm configuration procedures, we chose iterated local search, mainly because of its conceptual simplicity and flexibility.

5.2 The BASICILS Algorithm

In order to turn PARAMILS as specified in Algorithm Framework 5.1 into an executable configuration procedure, it is necessary to instantiate the function *better* that determines which of two parameter settings should be preferred. We will ultimately propose several different ways of doing this.

5.2.1 Algorithm Statement

Here, we describe the simplest approach, which we call BASICILS. Specifically, we use the term BASICILS(N) to refer to a PARAMILS algorithm in which the function *better*($\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$) is implemented as shown in Procedure 5.3: simply comparing estimates $\hat{c}_N$ of the cost measures $c(\boldsymbol{\theta}_1)$ and $c(\boldsymbol{\theta}_2)$ based on N runs each.

Like many other related approaches (see, *e.g.*, Minton, 1996; Coy et al., 2001; Adenso-Diaz and Laguna, 2006), BASICILS deals with the stochastic part of the optimization problem by using an estimate based on a fixed number, N, of training instances. It evaluates every parameter configuration by running it on the same N training benchmark instances using the same seeds. This amounts to a simple blocking strategy which reduces the variance in comparing two configurations. (See 3.6.1 for a more detailed discussion of blocking.) When benchmark instances are very heterogeneous or when the user can identify a rather small

Section 8.2, where we automatically optimize the parameters of PARAMILS itself.

Procedure 5.3: better$_N(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$
Procedure used in BASICILS(N) and RANDOMSEARCH(N) to compare two parameter configurations. Procedure objective($\boldsymbol{\theta}, N$) returns the user-defined objective achieved by $\mathcal{A}(\boldsymbol{\theta})$ on the first N instances and keeps track of the incumbent solution; it is detailed in Procedure 7.1 on page 105.

Input : Parameter configuration $\boldsymbol{\theta}_1$, parameter configuration $\boldsymbol{\theta}_2$
Output : True if $\boldsymbol{\theta}_1$ performs better than or equal to $\boldsymbol{\theta}_2$ on the first N instances; false otherwise
Side Effect : Adds runs to the global caches of performed algorithm runs $\mathbf{R}_{\boldsymbol{\theta}_1}$ and $\mathbf{R}_{\boldsymbol{\theta}_2}$; updates the incumbent, $\boldsymbol{\theta}_{inc}$

1 $\hat{c}_N(\boldsymbol{\theta}_2) \leftarrow objective(\boldsymbol{\theta}_2, N)$
2 $\hat{c}_N(\boldsymbol{\theta}_1) \leftarrow objective(\boldsymbol{\theta}_1, N)$
3 **return** $\hat{c}_N(\boldsymbol{\theta}_1) \leq \hat{c}_N(\boldsymbol{\theta}_2)$

"representative" subset of instances, this approach can find good parameter configurations with comparably low computational effort.

5.2.2 Experimental Evaluation

In this section, we evaluate the effectiveness of BASICILS(N) against two of its components:

1. a simple random search, used in BASICILS for initialization (we dub it RANDOMSEARCH(N) and provide pseudocode for it in Algorithm 5.4); and
2. a simple local search, the same type of iterative first improvement search used in BASICILS(N), ending in the first local minimum (we dub it SIMPLELS(N)).

If there is sufficient structure in the search space, we expect BASICILS to outperform RANDOMSEARCH. If there are local minima, we expect BASICILS to perform better than basic local search. Our experiments showed that BASICILS did indeed offer the best performance.

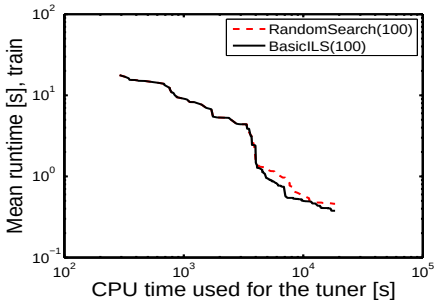
Here, we are solely interested in comparing how effectively the approaches search the space of parameter configurations, rather than how the found parameter configurations generalize to unseen test instances. Thus, in order to reduce variance in our comparisons, we compare the configuration methods in terms of their performance on the training set. For all configuration procedures, we performed 25 independent repetitions, each of them with a different training set of 100 instances (constructed as described in Section 3.6.1).

First, we present our comparison of BASICILS against RANDOMSEARCH. In Figure 5.1, we plot the performance achieved by the two approaches at a given time, for the two configuration scenarios with the least and the most pronounced differences between the two configuration procedures: SAPS-SWGCP and CPLEX-REGIONS100. BASICILS started with $r = 10$ random samples, which meant that performance for the first part of the trajectory was always identical (the kth run of each configurator used the same training instances and seeds). After these random samples, BASICILS performed better, quite clearly so for CPLEX-REGIONS100. Table 5.1 quantifies the performance of both approaches on our BROAD configuration scenarios. BASICILS always performed better on average, and in three of the five scenarios the difference was statistically significant as judged by a paired Max-Wilcoxon test

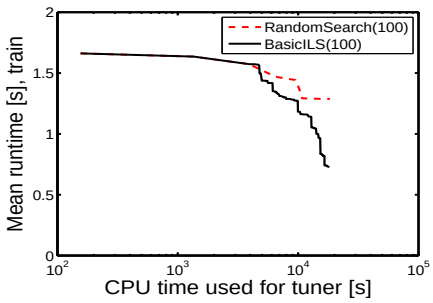
Algorithm 5.4: RANDOMSEARCH$(N, \boldsymbol{\theta}_0)$
Outline of random search in parameter configuration space; $\boldsymbol{\theta}_{inc}$ denotes the incumbent parameter configuration, $better_N$ compares two configurations based on the first N instances from the training set.

Input : Number of runs to use for evaluating parameter configurations, N; initial configuration $\boldsymbol{\theta}_0 \in \boldsymbol{\Theta}$.
Output : Best parameter configuration $\boldsymbol{\theta}_{inc}$ found.

```
θ_inc ← θ_0;
while not TerminationCriterion() do
    θ ← random θ ∈ Θ;
    if better_N(θ, θ_inc) then
        θ_inc ← θ;
return θ_inc
```



(a) SAPS-SWGCP, not statistically significant.



(b) CPLEX-REGIONS100, statistically significant.

Figure 5.1: Comparison of BASICILS(100) and RANDOMSEARCH(100). For each time step t, we compute training performance $p_{t,train}$ (penalized average runtime, PAR, over $N = 100$ training instances using the procedures' incumbents, $\boldsymbol{\theta}_{inc}(t)$). We plot median $p_{t,train}$ across 25 runs of each configurator (plots are similar for *means*, but more variable and thus cluttered). Note the logarithmic scale on the x-axis, and the difference in y-axis scales: we chose a log-scale for SAPS-SWGCP due the large performance variation seen for this scenario, and a linear scale for CPLEX-REGIONS100, where even poor configurations performed quite well. We will use these two configuration scenarios for visualization purposes throughout, always using the same y-axes as in this plot.

(see Section 3.6.2). Table 5.1 also lists the performance of the default parameter configuration for each of the configuration scenarios. We note that both BASICILS and RANDOMSEARCH consistently made substantial (and statistically significant) improvements over these default configurations.

Next, we compared BASICILS against its second component, SIMPLELS. This method is identical to BASICILS but stops at the first local minimum encountered. We used it in order to

Scenario	Training performance (PAR, CPU seconds) Default	RANDOMSEARCH(100)	BASICILS(100)	p-value
SAPS-SWGCP	19.93	0.46 ± 0.34	$\mathbf{0.38 \pm 0.19}$	0.94
SPEAR-SWGCP	10.61	7.02 ± 1.11	$\mathbf{6.78 \pm 1.73}$	0.18
SAPS-QCP	12.71	3.96 ± 1.185	$\mathbf{3.19 \pm 1.19}$	$\mathbf{1.4 \cdot 10^{-5}}$
SPEAR-QCP	2.77	0.58 ± 0.59	$\mathbf{0.36 \pm 0.39}$	$\mathbf{0.007}$
CPLEX-REGIONS100	1.61	1.45 ± 0.35	$\mathbf{0.72 \pm 0.45}$	$\mathbf{1.2 \cdot 10^{-5}}$

Table 5.1: Comparison of RANDOMSEARCH(100) and BASICILS(100). We performed 25 runs of the configurators and computed their training performance $p_{train,t}$ (PAR over $N = 100$ training instances using the procedures' final incumbents $\boldsymbol{\theta}_{inc}(t)$) for a configuration time of $t = 18\,000s = 5h$. We list training performance of the algorithm default, mean $\pm$ stddev of $p_{train,t}$ across the 25 runs for BASICILS(100) & FOCUSEDILS, and the p-value for a paired Max-Wilcoxon test (see Section 3.6.2) for the difference between the two configurators' performances.

Scenario	SIMPLELS(100) Performance	BASICILS(100) Performance	Avg. # ILS iterations	p-value
SAPS-SWGCP	0.5 ± 0.39	$\mathbf{0.38 \pm 0.19}$	2.6	$\mathbf{9.8 \cdot 10^{-4}}$
SAPS-QCP	3.60 ± 1.39	$\mathbf{3.19 \pm 1.19}$	5.6	$\mathbf{4.4 \cdot 10^{-4}}$
SPEAR-QCP	0.4 ± 0.39	$\mathbf{0.36 \pm 0.39}$	1.64	$\mathbf{0.008}$

Table 5.2: Comparison of SIMPLELS(100) and BASICILS(100). We performed 25 runs of the configurators and computed their training performance $p_{train,t}$ (PAR over $N = 100$ training instances using the procedures' final incumbents $\boldsymbol{\theta}_{inc}(18000)$). We list mean $\pm$ stddev of $p_{train,t}$ across the 25 runs and the p-value for comparing their performance. In configuration scenarios SPEAR-SWGCP and CPLEX-REGIONS100, BASICILS did not complete its first ILS iteration in any of the 25 runs; the two approaches were thus identical and are not listed here.

study whether local minima pose a problem for simple first improvement search. Table 5.2 shows that in the three configuration scenarios where BASICILS had time to perform multiple ILS iterations, its training set performance was statistically significantly better than that of SIMPLELS. Thus, we conclude that for these scenarios the parameter configuration space contains structure that can be exploited with a local search algorithm as well as local minima that can limit the performance of iterative improvement search.

5.3 FOCUSEDILS: Adaptively Selecting the Number of Training Instances

The question of how to choose the number of training instances, N, in BASICILS(N) has no straightforward answer: optimizing performance using too small a training set intuitively leads to good training performance but poor generalization to previously unseen test benchmarks. On the other hand, we clearly cannot evaluate every parameter configuration on an enormous training set—if we did, search progress would be unreasonably slow. Furthermore, as we saw in our empirical analysis in Chapter 4, the best fixed value for N differs across configuration

scenarios.

5.3.1 Algorithm Statement

FOCUSEDILS is a variant of PARAMILS that adaptively varies the number of training samples considered from one parameter configuration to another. We denote the number of runs available to estimate the cost of a parameter configuration, $\boldsymbol{\theta}$, by $N(\boldsymbol{\theta})$. Having performed different numbers of runs using different parameter configurations, we face the question of comparing two parameter configurations $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ for which $N(\boldsymbol{\theta}_1) \neq N(\boldsymbol{\theta}_2)$. One option would be simply to compute the empirical cost statistic based on the available number of runs for each configuration. However, this could lead to systematic biases if, for example, the first instances are easier than the average instance. For this reason, when comparing two parameter configurations $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$ with $N(\boldsymbol{\theta}) \leq N(\boldsymbol{\theta}')$, we simply compare them based on the first $N(\boldsymbol{\theta})$ runs. As in BASICILS, these first $N(\boldsymbol{\theta})$ runs use the same instances and seeds to evaluate both configurations, which amounts to a blocking strategy.

This approach to comparison leads us to a concept of domination. We say that $\boldsymbol{\theta}_1$ dominates $\boldsymbol{\theta}_2$ when at least as many runs have been conducted on $\boldsymbol{\theta}_1$ as on $\boldsymbol{\theta}_2$, and the performance of $\mathcal{A}(\boldsymbol{\theta}_1)$ on the first $N(\boldsymbol{\theta}_2)$ runs is at least as good as that of $\mathcal{A}(\boldsymbol{\theta}_2)$ on all of its runs.

Definition 6 (Domination)**.** $\boldsymbol{\theta}_1$ dominates $\boldsymbol{\theta}_2$ *if and only if* $N(\boldsymbol{\theta}_1) \geq N(\boldsymbol{\theta}_2)$ *and* $\hat{c}_{N(\boldsymbol{\theta}_2)}(\boldsymbol{\theta}_1) \leq \hat{c}_{N(\boldsymbol{\theta}_2)}(\boldsymbol{\theta}_2)$.

Now we are ready to discuss the comparison strategy encoded in Procedure *better*$_{Foc}(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$, which is used by the FOCUSEDILS algorithm (see Procedure 5.5). This procedure first acquires one additional sample for the configuration i having smaller $N(\boldsymbol{\theta}_i)$, or one run for both configurations if they have the same number of runs. Then, it continues performing runs in this way until one configuration dominates the other. At this point it returns true if $\boldsymbol{\theta}_1$ dominates $\boldsymbol{\theta}_2$, and false otherwise. We also keep track of the total number of configurations evaluated since the last improving step (i.e., since the last time *better*$_{Foc}$ returned true); we denote this number as B. Whenever *better*$_{Foc}(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$ returns true, we perform B "bonus" runs for $\boldsymbol{\theta}_1$ and reset B to 0. This mechanism ensures that we perform many runs with good configurations, and that the error made in every comparison of two configurations $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ decreases on expectation.

5.3.2 Theoretical Analysis

It is not difficult to show that in the limit, FOCUSEDILS will sample every parameter configuration an unbounded number of times. The proof relies on the fact that, as an instantiation of PARAMILS, FOCUSEDILS performs random restarts with positive probability.

Lemma 7 (Unbounded number of evaluations)**.** *Let* $N(J, \boldsymbol{\theta})$ *denote the number of runs* FOCUSEDILS *has performed with parameter configuration* $\boldsymbol{\theta}$ *at the end of ILS iteration* J *to estimate* $c(\boldsymbol{\theta})$*. Then, for any constant* K *and configuration* $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ *(with finite* $\boldsymbol{\Theta}$*),* $\lim_{J \to \infty} P[N(J, \boldsymbol{\theta}) \geq K] = 1$.

Procedure 5.5: better$_{Foc}(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$

Procedure used in FOCUSEDILS to compare two parameter configurations. Procedure $objective(\boldsymbol{\theta}, N)$ returns the user-defined objective achieved by $\mathcal{A}_{\boldsymbol{\theta}}$ on the first N instances, keeps track of the incumbent solution, and updates $\mathbf{R}_{\boldsymbol{\theta}}$ (a global cache of algorithm runs performed with parameter configuration $\boldsymbol{\theta}$); it is detailed in Procedure 7.1 on page 105. For each $\boldsymbol{\theta}$, $N(\boldsymbol{\theta}) = \text{length}(\mathbf{R}_{\boldsymbol{\theta}})$. B is a global counter denoting the number of configurations evaluated since the last improvement step.

Input : Parameter configuration $\boldsymbol{\theta}_1$, parameter configuration $\boldsymbol{\theta}_2$
Output : True if $\boldsymbol{\theta}_1$ dominates $\boldsymbol{\theta}_2$; false otherwise (in that case, here, $\boldsymbol{\theta}_2$ dominates $\boldsymbol{\theta}_1$)
Side Effect : Adds runs to the global caches of performed algorithm runs $\mathbf{R}_{\boldsymbol{\theta}_1}$ and $\mathbf{R}_{\boldsymbol{\theta}_2}$; updates the global counter B of bonus runs, and the incumbent, $\boldsymbol{\theta}_{inc}$

$B \leftarrow B + 1$
if $N(\boldsymbol{\theta}_1) \leq N(\boldsymbol{\theta}_2)$ **then**
  $\boldsymbol{\theta}_{min} \leftarrow \boldsymbol{\theta}_1; \boldsymbol{\theta}_{max} \leftarrow \boldsymbol{\theta}_2$
  if $N(\boldsymbol{\theta}_1) = N(\boldsymbol{\theta}_2)$ **then** $B \leftarrow B + 1$
else $\boldsymbol{\theta}_{min} \leftarrow \boldsymbol{\theta}_2; \boldsymbol{\theta}_{max} \leftarrow \boldsymbol{\theta}_1$
repeat
  $i \leftarrow N(\boldsymbol{\theta}_{min}) + 1$
  // ===== If $N(\boldsymbol{\theta}_{min}) = N(\boldsymbol{\theta}_{max})$, the next line adds a new run to $\mathbf{R}_{\boldsymbol{\theta}_{max}}$.
  $\hat{c}_i(\boldsymbol{\theta}_{max}) \leftarrow objective(\boldsymbol{\theta}_{max}, i)$
  $\hat{c}_i(\boldsymbol{\theta}_{min}) \leftarrow objective(\boldsymbol{\theta}_{min}, i)$ *// Adds a new run to $\mathbf{R}_{\boldsymbol{\theta}_{min}}$.*
until *dominates($\boldsymbol{\theta}_1$, $\boldsymbol{\theta}_2$)* **or dominates($\boldsymbol{\theta}_2$, $\boldsymbol{\theta}_1$)**
if *dominates($\boldsymbol{\theta}_1$, $\boldsymbol{\theta}_2$)* **then**
  // ===== Perform B bonus runs.
  $\hat{c}_{N(\boldsymbol{\theta}_1)+B}(\boldsymbol{\theta}_1) \leftarrow objective(\boldsymbol{\theta}_1, N(\boldsymbol{\theta}_1) + B)$ *// Adds B new runs to $\mathbf{R}_{\boldsymbol{\theta}_1}$.*
  $B \leftarrow 0$
  return true
else return false

Procedure 5.6: dominates($\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$)

Input : Parameter configuration $\boldsymbol{\theta}_1$, parameter configuration $\boldsymbol{\theta}_2$
Output : True if $\boldsymbol{\theta}_1$ dominates $\boldsymbol{\theta}_2$, false otherwise
Side Effect : Adds runs to the global caches of performed algorithm runs $\mathbf{R}_{\boldsymbol{\theta}_1}$ and $\mathbf{R}_{\boldsymbol{\theta}_2}$; updates the incumbent, $\boldsymbol{\theta}_{inc}$

if $N(\boldsymbol{\theta}_1) < N(\boldsymbol{\theta}_2)$ **then return** false
else return $objective(\boldsymbol{\theta}_1, N(\boldsymbol{\theta}_2)) \leq objective(\boldsymbol{\theta}_2, N(\boldsymbol{\theta}_2))$

Proof. After each ILS iteration of PARAMILS, with probability $p_{restart} > 0$ a new configuration is picked uniformly at random, and with a probability of $1/|\boldsymbol{\Theta}|$, this is configuration $\boldsymbol{\theta}$. The probability of visiting $\boldsymbol{\theta}$ in an ILS iteration is thus $p \geq \frac{p_{restart}}{|\boldsymbol{\Theta}|} > 0$. Hence, the number of runs performed with $\boldsymbol{\theta}$ is lower-bounded by a binomial random variable $\mathcal{B}(k; J, p)$. Then, for any constant $k < K$ we obtain $\lim_{J\to\infty} \mathcal{B}(k; J, p) = \lim_{J\to\infty} \binom{J}{k} p^k (1-p)^{J-k} = 0$. Thus, $\lim_{J\to\infty} P[N(J, \boldsymbol{\theta}) \geq K] = 1$. □

Definition 8 (Consistent estimator)**.** $\hat{c}_N(\boldsymbol{\theta})$ *is a* consistent estimator *for* $c(\boldsymbol{\theta})$ *iff*

$$\forall \epsilon > 0 \; : \; \lim_{N \to \infty} P(|\hat{c}_N(\boldsymbol{\theta}) - c(\boldsymbol{\theta})| < \epsilon) = 1.$$

When $\hat{c}_N(\boldsymbol{\theta})$ is a consistent estimator of $c(\boldsymbol{\theta})$, cost estimates become more and more reliable as N goes to infinity, eventually eliminating overconfidence and the possibility of mistakes in comparing two parameter configurations (and thus, over-tuning). This fact is captured in the following lemma.

Lemma 9 (No mistakes for $N \to \infty$)**.** *Let* $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2 \in \boldsymbol{\Theta}$ *be any two parameter configurations with* $c(\boldsymbol{\theta}_1) < c(\boldsymbol{\theta}_2)$*. Then, for consistent estimators* $\hat{c}_N$*,* $\lim_{N \to \infty} P(\hat{c}_N(\boldsymbol{\theta}_1) \geq \hat{c}_N(\boldsymbol{\theta}_2)) = 0$*.*

Proof. Write c_1 as shorthand for $c(\boldsymbol{\theta}_1)$, c_2 for $c(\boldsymbol{\theta}_2)$, $\hat{c}_1$ for $\hat{c}_N(\boldsymbol{\theta}_1)$, and $\hat{c}_2$ for $\hat{c}_N(\boldsymbol{\theta}_2)$. Define $m = \frac{1}{2} \cdot (c_2 + c_1)$ as the midpoint between c_1 and c_2, and $\epsilon = c_2 - m = m - c_1 > 0$ as its distance from each of the two points.

Since $\hat{c}_N$ is a consistent estimator for c, the estimate $\hat{c}_1$ comes arbitrarily close to the real cost c_1. That is, $\lim_{N \to \infty} P(|\hat{c}_1 - c_1| < \epsilon) = 1$. Since $|m - c_1| = \epsilon$, the estimate $\hat{c}_1$ cannot be greater than or equal to m: $\lim_{N \to \infty} P(\hat{c}_1 \geq m) = 0$. Similarly, $\lim_{N \to \infty} P(\hat{c}_2 < m) = 0$. Since

$$\begin{aligned} P(\hat{c}_1 \geq \hat{c}_2) &= P(\hat{c}_1 \geq \hat{c}_2 \wedge \hat{c}_1 \geq m) + P(\hat{c}_1 \geq \hat{c}_2 \wedge \hat{c}_1 < m) \\ &= P(\hat{c}_1 \geq \hat{c}_2 \wedge \hat{c}_1 \geq m) + P(\hat{c}_1 \geq \hat{c}_2 \wedge \hat{c}_1 < m \wedge \hat{c}_2 < m) \\ &\leq P(\hat{c}_1 \geq m) + P(\hat{c}_2 < m), \end{aligned}$$

we have $\lim_{N \to \infty} P(\hat{c}_1 \geq \hat{c}_2) \leq \lim_{N \to \infty} (P(\hat{c}_1 \geq m) + P(\hat{c}_2 < m)) = 0 + 0 = 0$. □

Combining our two lemmata we can now show that in the limit, FOCUSEDILS is guaranteed to converge to the true best parameter configuration.

Theorem 10 (Convergence of FOCUSEDILS)**.** *When* FOCUSEDILS *optimizes a cost measure* c *based on a consistent estimator* $\hat{c}_N$ *and a finite configuration space* $\boldsymbol{\Theta}$*, the probability that it finds the true optimal parameter configuration* $\boldsymbol{\theta}^* \in \boldsymbol{\Theta}$ *approaches one as the number of ILS iterations goes to infinity.*

Proof. According to Lemma 7, $N(\boldsymbol{\theta})$ grows unboundedly for each $\boldsymbol{\theta} \in \boldsymbol{\Theta}$. For each $\boldsymbol{\theta}_1$, $\boldsymbol{\theta}_2$, as $N(\boldsymbol{\theta}_1)$ and $N(\boldsymbol{\theta}_2)$ go to infinity, Lemma 9 states that in a pairwise comparison, the truly better configuration will be preferred. Thus eventually, FOCUSEDILS visits all (finitely-many) parameter configurations and prefers the best one over all others with probability arbitrarily close to one. □

In many practical scenarios, cost estimators may not actually be consistent. This limits the applicability of Theorem 10. For example, when a finite training set, Π, is used during configuration rather than a distribution over problem instances, then even for large N, $\hat{c}_N$ will only accurately reflect the cost of parameter configurations on Π. For small $|\Pi|$, the cost estimate based on Π may differ substantially from the true cost as defined by performance

across the distribution, $\mathcal{D}$. As $|\Pi|$ goes to infinity, the difference vanishes, assuming Π is sampled uniformly at random from $\mathcal{D}$.

To the best of our knowledge, Theorem 10 is the first convergence results for an algorithm configuration procedure. Prominent competitors, such as F-Race (Birattari et al., 2002; Birattari, 2004), do not admit such a result. However, we do note that essentially the same argument could be applied to a simple round-robin procedure that loops through all parameter configurations; nevertheless, we would not expect that procedure to perform very well in practice. Thus, in what follows, we will rely on experimental evidence to test FOCUSEDILS's practical performance.

5.3.3 Experimental Evaluation

In this section we investigate FOCUSEDILS' performance experimentally. In contrast to our previous comparison of RANDOMSEARCH, SIMPLELS, and BASICILS using *training* performance, we now compare FOCUSEDILS against (only) BASICILS using *test* performance. This is motivated by the fact that, while in our previous comparison all approaches used the same number of target algorithm runs to evaluate a parameter configuration, the number of target algorithm runs FOCUSEDILS uses to evaluate configurations grows over time. However, the cost measure to be optimized, c, remains constant; therefore test performance (an unbiased estimator of c) provides a fairer basis for comparison than training performance. We only compare FOCUSEDILS to BASICILS, since BASICILS was already shown to outperform RANDOMSEARCH and SIMPLELS in Section 5.2.2.

The first experiment we performed was for scenario SAPS-QWH, a simple configuration scenario with the objective of minimizing the median number of search steps SAPS requires to solve a single quasigroups with holes (QWH) instance (see Section 3.5 for details about this configuration scenario). This simple scenario is convenient because it allows us to perform a large number of configuration runs fairly cheaply. Figure 5.2 compares the test performance of FOCUSEDILS and BASICILS(N) with $N = 1$, 10 and 100. In this experiment, BASICILS(1) showed clear evidence of over-tuning. In contrast, using a large number of target algorithm runs to evaluate every configuration—as done in BASICILS(100)—resulted in a very slow search, but eventually led to configurations with good test performance. FOCUSEDILS combined the best of these two schemes: its search progress started about as quickly as BASICILS(1), but it also performed better than BASICILS(10) and BASICILS(100) at all times.

Figure 5.3 shows a similar comparison on the two BROAD scenarios we have used for visualization throughout. This comparison confirms the same overall pattern as the comparison for the simpler configuration scenario SAPS-QWH. For CPLEX-REGIONS100, over-tuning is much less of an issue (as we already saw in the empirical analysis in Section 4.4); thus, BASICILS(1) actually performed quite competitively for this scenario.

We compare the performance of FOCUSEDILS and BASICILS(100) for our BROAD configuration scenarios in Table 5.3. For three SAPS and CPLEX scenarios, FOCUSEDILS performed statistically significantly better than BASICILS(100). However, we found that in both scenarios involving the SPEAR algorithm, BASICILS(100) actually performed better on average than FOCUSEDILS, albeit not statistically significantly. We attribute this to the fact that for a

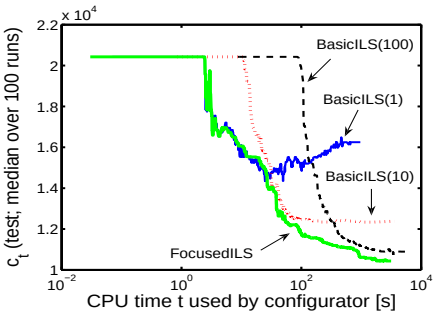
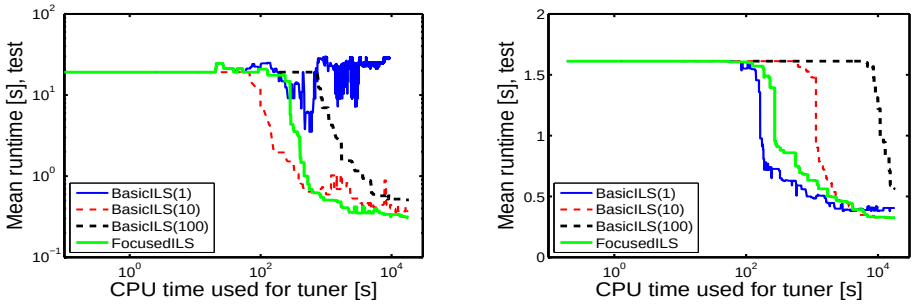


Figure 5.2: Comparison of BASICILS(N) with $N = 1$, 10, and 100 *vs* FOCUSEDILS for scenario SAPS-QWH. For each time step t, we compute test performance $p_{t,test}$ (SAPS median runlength across 1 000 test runs, using the procedures' incumbents, $\boldsymbol{\theta}_{inc}(t)$). We plot median $p_{test,t}$ across 100 runs of the configurators.



(a) SAPS-SWGCP (b) CPLEX-REGIONS100

Figure 5.3: Comparison of BASICILS(N) with $N = 1$, 10, and 100 *vs* FOCUSEDILS. For each time step t, we compute test performance $p_{test,t}$ (PAR across 1 000 test instances, using the procedures' incumbents, $\boldsymbol{\theta}_{inc}(t)$). We plot median $p_{test,t}$ across 25 runs of the configurators for two scenarios. Performance in the other three BROAD scenarios was qualitatively similar: BASICILS(1) was the fastest to move away from the starting parameter configuration, but its performance was not robust at all; BASICILS(10) was a rather good compromise between speed and generalization performance, but given enough time was outperformed by BASICILS(100). FOCUSEDILS started finding good configurations quickly (except for scenario SPEAR-QCP, where it took even longer than BASICILS(100) to improve over the default) and always was among the best approaches at the end of the configuration process.

Scenario	Test performance (PAR, CPU seconds) Default	BASICILS(100)	FOCUSEDILS	p-value
SAPS-SWGCP	20.41	0.59 ± 0.28	$\mathbf{0.32 \pm 0.08}$	$\mathbf{1.4 \cdot 10^{-4}}$
SPEAR-SWGCP	9.74	$\mathbf{8.13 \pm 0.95}$	8.40 ± 0.92	(0.21)
SAPS-QCP	12.97	4.87 ± 0.34	$\mathbf{4.69 \pm 0.40}$	**0.042**
SPEAR-QCP	2.65	$\mathbf{1.32 \pm 0.34}$	1.35 ± 0.20	(0.66)
CPLEX-REGIONS100	1.61	0.72 ± 0.45	$\mathbf{0.33 \pm 0.03}$	$\mathbf{1.2 \cdot 10^{-5}}$

Table 5.3: Comparison of BASICILS(100) and FOCUSEDILS. We performed 25 runs of the configurators and computed their test performance $p_{test,t}$ (PAR across 1 000 test instances, using $\boldsymbol{\theta}_{inc}(t)$) for a configuration time of $t = 18\,000s = 5h$. We give test performance of the algorithm default, mean $\pm$ stddev of $p_{test,t}$ across the 25 runs for BASICILS(100) and FOCUSEDILS, and p-values for a test comparing the configurators.

complete, industrial solver, such as SPEAR, the two benchmark distributions QCP and SWGCP are quite heterogeneous. We expect FOCUSEDILS to have problems in dealing with highly heterogeneous distributions, due to the fact that it tries to extrapolate performance based on only a few runs per parameter configuration.

5.4 Configuration of SAPS, GLS$^+$ and SAT4J

In our first publication on PARAMILS (Hutter et al., 2007b), we reported experiments on three target algorithms to demonstrate the effectiveness of the approach: SAPS, GLS$^+$, and SAT4J. These algorithms are described in more detail in Section 3.2. As a brief reminder, GLS$^+$ (with 1 binary and 4 numerical parameters) is a dynamic local search algorithm for solving the Most Probable Explanation (MPE) problem in Bayesian networks (Hutter et al., 2005). SAT4J[3] is a tree search SAT solver with 4 categorical and 7 numerical parameters. We studied four configuration scenarios: SAPS-QWH, GLS$^+$-GRID, SAPS-SW-HOM, SAT4J-SW-HOM. SAPS-QWH is as used throughout this thesis; it is discussed in detail in Section 3.5. The other three are only used locally in this section and we thus describe them here.

GLS$^+$-GRID In this scenario, we aimed to optimize mean solution quality achieved by GLS$^+$ for MPE in grid-structured Bayesian networks. Hutter et al. (2005) showed that GLS$^+$ was the state-of-the-art algorithm for these networks, and we are not aware of a better, more recent algorithm for solving them. In particular, we optimized the solution quality found (*i.e.*, the likelihood of the most likely instantiation found) within one minute of GLS$^+$ runtime. For each instance we computed the ratio ϵ of the likelihood found by the GLS$^+$ default within one hour and the likelihood GLS$^+$ found with a configuration $\boldsymbol{\theta}$ within one minute. We then averaged across instances. If $\epsilon < 1$, this means that GLS$^+$ with the automatically-found configurations yielded better solutions in one minute than with the default in one hour.

SAPS-SW-HOM Here, the objective is to minimize the median runtime of SAPS on a homo-

[3] http://www.sat4j.org

Scenario	Default	CALIBRA(100)	BASICILS(100)	FOCUSEDILS	p-value
GLS$^+$-GRID	$\epsilon = 1.81$	1.240 ± 0.469	0.965 ± 0.006	0.968 ± 0.002	
		1.234 ± 0.492	0.951 ± 0.004	$\mathbf{0.949 \pm 0.0001}$	**0.0016**
SAPS-QWH	$85.5K$	$8.6K \pm 0.7K$	$8.7K \pm 0.4K$	$10.2K \pm 0.4K$	
	steps	$10.7K \pm 1.1K$	$10.9K \pm 0.6K$	$\mathbf{10.6K \pm 0.5K}$	0.52
SAPS-SW-HOM	5.60	0.044 ± 0.005	0.040 ± 0.002	0.043 ± 0.005	
	seconds	0.053 ± 0.010	0.046 ± 0.01	$\mathbf{0.043 \pm 0.005}$	**0.0003**
SAT4J-SW-HOM	7.02	N/A (too many	0.96 ± 0.59	0.62 ± 0.21	
	seconds	parameters)	1.19 ± 0.58	$\mathbf{0.65 \pm 0.2}$	**0.00007**

Table 5.4: Configuration of GLS$^+$, SAPS, and SAT4J. For each scenario and approach, training and test performance are listed in the top and bottom row, respectively (mean $\pm$ stddev over 25 independent runs); "p-value" refers to the p-value of a Wilcoxon signed rank test for testing the null hypothesis "there is no difference between CALIBRA and FOCUSEDILS results" (BASICILS *vs* FOCUSEDILS for SAT4J).

geneous subset of benchmark set SWGCP (a set of SAT-encoded small-world graph-colouring instances discussed in Section 3.3.1). For this subset, we selected only those 2 202 instances that SAT4J could solve in a median runtime between five and ten seconds.

SAT4J-SW-HOM Here, the objective is to minimize the median runtime of SAT4J on a homogeneous subset of benchmark set SWGCP. For this subset, we selected only those 213 instances that SAPS could solve in a median runtime between five and ten seconds.

For scenarios SAPS-QWH, SAPS-SW-HOM, and GLS$^+$-GRID, we allowed each configuration procedure to execute 20 000 runs of the target algorithm. This led to time budgets of 1 hour for SAPS-QWH, 10 hours for SAPS-SW-HOM, and 55 hours for GLS$^+$-GRID. For SAT4J-SW-HOM, we used a time budget of 10 hours.

We compared the performance of the respective algorithm's default performance, and the performance with the configurations found by BASICILS, FOCUSEDILS, and the CALIBRA system (Adenso-Diaz and Laguna, 2006). We described CALIBRA in detail in Section 2.1.1. Recall that it is limited to tuning continuous parameters, and to a maximum of five parameters. Overall, automated parameter optimization using PARAMILS consistently achieved substantial improvements over the algorithm defaults.

We summarize the results in Table 5.4. GLS$^+$ was sped up by a factor of more than 360 (configured parameters found better solutions in 10 seconds than the default found in one hour), SAPS by factors of 8 and 130 on SAPS-QWH and SAPS-SW-HOM, respectively, and SAT4J by a factor of 11. CALIBRA led to speedups of a similar magnitude, but to significantly worse results than PARAMILS in two of three configuration scenarios. For the fourth, it was not applicable since it could not handle the categorical parameters of SAT4J.

We also studied the generalization of the parameter configuration found by PARAMILS to harder problem instances from benchmark set SWGCP. Figure 5.4 shows that this generalization was extremely good, leading to a speedup of more than a factor of 600 on average.

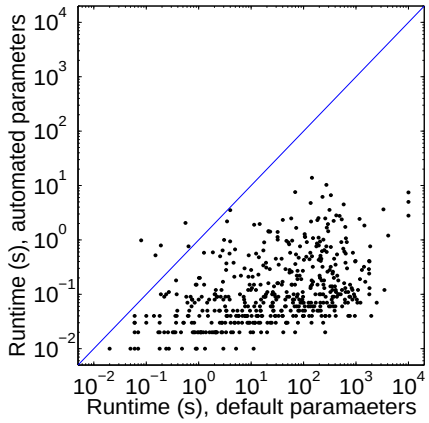


Figure 5.4: Comparison of manually-engineered default *vs* configuration found in single run of FOCUSEDILS for scenario SAPS-SW-HOM. Automated configuration used a homogeneous subset, but here we plot 500 randomly-sampled instances of the whole benchmark set SWGCP. Mean runtimes were 263 seconds (default) *vs* 0.43 seconds (automatically-configured), a speedup factor of over 600.

5.5 Chapter Summary

In this chapter, we introduced a general framework for iterated local search in parameter configuration space, providing one possible answer to the first dimension of algorithm configuration: which sequential search strategy to use? We provided two instantiations of this framework, BASICILS and FOCUSEDILS. The primary difference between the two is in their choice for the second dimension of algorithm design: how many runs, N, should be used to evaluate each configuration? BASICILS uses a fixed N, whereas FOCUSEDILS adaptively chooses a different number of runs, $N(\boldsymbol{\theta})$, for each configuration, $\boldsymbol{\theta}$. In this chapter, we fixed the third dimension of algorithm configuration, using a constant cutoff time of κ_{max}. We focus on that dimension in Chapter 7.

A comparison of BASICILS to a simple random search procedure on a broad collection of configuration scenarios showed that BASICILS often performed significantly and sometimes also substantially better. In scenarios where BASICILS had the time to perform several iterations, it also significantly improved upon the first local optimum. A comparison of FOCUSEDILS and BASICILS showed that FOCUSEDILS starts its search as quickly as BASICILS with a small N but also achieves good performance for long runtimes (similarly to BASICILS with a large N). For three of our five BROAD scenarios, it performed significantly better than BASICILS, and for two of them (SAPS-SWGCP and CPLEX-REGIONS100) the differences were substantial.

Finally, we reported experiments comparing PARAMILS to the previous CALIBRA system. In that comparison, FOCUSEDILS was significantly better than CALIBRA in 2 of 3 scenarios. More importantly, our PARAMILS approach is more generally-applicable, handling the configuration of complex algorithms with many categorical parameters. We showed that by configuring SAPS on SWGCP instances, we achieved a 600-fold speedup over the SAPS

default. In the next chapter, we demonstrate similar speedups for configuring a state-of-the-art tree search solver on a set of industrial SAT-encoded software verification problem instances.

Chapter 6

Applications I: A Case Study on the Configuration of SPEAR for Industrial Verification Problems

It is the weight, not numbers of experiments that is to be regarded.
—Sir Isaac Newton, English physicist and astronomer

In this chapter[1], we present a case study that demonstrates the potential of algorithm configuration in general, and PARAMILS in particular, to substantially improve the performance of an existing high-performance SAT solver. Our target algorithm in this case study is SPEAR (Babić, 2008), a modular arithmetic decision procedure and SAT solver, developed by Domagoj Babić in support of the CALYSTO static checker (Babić and Hu, 2007b). Here, we only use the SAT solver component of SPEAR. With 26 user-definable search parameters, this part has a complex parameter configuration space. Although the performance of an early, manually-configured version of SPEAR was comparable to that of a state-of-the-art SAT solver (MiniSAT 2.0 (Eén and Sörensson, 2003)), we ultimately achieved speedup factors of 4.5 for SAT-encoded industrial bounded model checking instances, and of over 500 for software verification instances. The use of PARAMILS also influenced the design of SPEAR and gave the algorithm designer some important insights about differences between (SAT-encoded) hardware and software verification problems. For example, the software verification instances generated by the CALYSTO static checker required more aggressive use of SPEAR's restart mechanism than the bounded model checking hardware verification benchmarks we studied.

6.1 Formal Verification and Decision Procedures

The problems encountered in automated formal verification are typically hard. As with other computationally difficult problems, the key to practical solutions lies in decision procedures

[1]This chapter is based on published joint work with Domagoj Babić, Holger Hoos and Alan Hu (Hutter et al., 2007a). Domagoj deserves credit for much of the material presented in this chapter; it is based on a joint case study performed when he was still a PhD student at UBC. In this case study, Domagoj played the role of algorithm designer while the author of this thesis performed the algorithm configuration using PARAMILS.

that use heuristic techniques. This is true for a wide range of decision procedures, be they based on a binary decision diagram (BDD, see, *e.g.*, Bryant, 1986) package, a Boolean satisfiability (SAT) solver (see, *e.g.*, Prasad et al., 2005), or an automated theorem prover based on the Nelson-Oppen framework (Nelson, 1979).

A high-performance decision procedure typically uses multiple heuristics that interact in complex ways. Some examples from the SAT-solving world include variable and value selection, clause deletion, next watched literal selection, and initial variable ordering heuristics (see, *e.g.*, Silva, 1999; Moskewicz et al., 2001; Bhalla et al., 2003). The behaviour and performance of these heuristics is typically controlled by parameters, and the complex effects and interactions between these parameters render their configuration extremely challenging.

The typical development of a heuristic decision procedure follows the manual process already outlined in Section 1.1.1: certain heuristic choices and parameter settings are tested incrementally, typically using a modest collection of benchmark instances that are of particular interest to the developer. Many choices and parameter settings thus made are "locked in" during early stages of the process, and typically, only few parameters are exposed to the users of the finished solver. In many cases, these users never change the default settings of the exposed parameters or manually tune them in a manner similar to that used earlier by the developer.

There are almost no publications on automated parameter optimization for decision procedures for formal verification. Seshia (2005) explored using support vector machine (SVM) classification to choose between two encodings of difference logic into Boolean SAT. The learned classifier was able to choose the better encoding in most instances tested, resulting in a hybrid encoding that mostly dominated the two pure encodings.

There is, however, a fair amount of previous work on optimizing SAT solvers for particular verification applications. For example, Shtrichman (2000) considered the influence of variable and phase decision heuristics (especially static ordering), restriction of the set of variables for case splitting, and symmetric replication of conflict clauses on solving bounded model checking (BMC) problems. He evaluated seven strategies on the Grasp SAT solver, and found that static ordering performed fairly well, although no parameter combination was a clear winner. Later, Shacham and Zarpas (2003) showed that Shtrichman's conclusions do not apply to zChaff's less greedy VSIDS heuristic on their set of benchmarks, claiming that Shtrichman's conclusions were either benchmark- or engine-dependent. Shacham and Zarpas evaluated four different decision strategies on IBM BMC instances, and found that static ordering performed worse than VSIDS-based strategies. Lu et al. (2003) exploited signal correlations to design a number of techniques for SAT solving specific for automatic test pattern generation (ATPG). Their technique showed roughly an order of magnitude improvement on a small set of ATPG benchmarks.

6.2 Algorithm Development

The core of SPEAR is a DPLL-style (Davis et al., 1962) SAT solver, but with several novelties. For example, SPEAR features an elaborate clause prefetching mechanism that improves memory locality. To improve the prediction rate of the prefetching mechanism, Boolean

constraint propagation (BCP) and conflict analysis were redesigned to be more predictable. SPEAR also features novel heuristics for decision making, phase selection, clause deletion, and variable and clause elimination.[2] Given all of these features, extensions, and heuristics, many components of SPEAR are parameterized. This includes the choice of heuristics, as well as enabling (or disabling) of various features, such as, for example, the pure-literal rule, randomization, clause deletion, and literal sorting in freshly learned clauses. In total, SPEAR has 26 search parameters (10 categorical, 12 continuous, and 4 integer).

6.2.1 Manual Tuning

After the first version of SPEAR was written and its correctness thoroughly tested, its developer, Domagoj Babić, spent one week on manual performance optimization, which involved: (i) optimization of the implementation, resulting in a speedup by roughly a constant factor, with no noticeable effects on the search parameters; and (ii) manual optimization of roughly twenty search parameters, most of which were hard-coded and scattered around the code at the time.

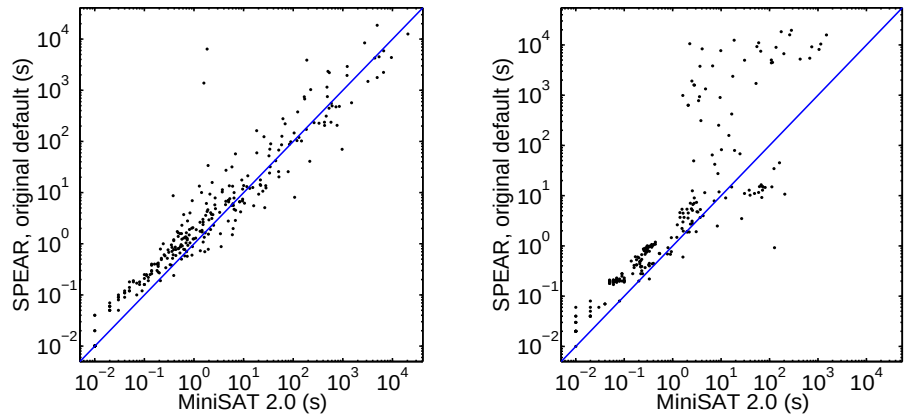
The manual parameter optimization was a slow and tedious process done in the following manner: first, the SPEAR developer collected several medium-sized benchmark instances that SPEAR could solve in at most $1\,000$ seconds, and attempted to come up with a parameter configuration that would result in a minimum total runtime on this set. The benchmark set was very limited and included several medium-sized BMC and some small software verification (SWV) instances generated by the CALYSTO static checker (Babić and Hu, 2007b).[3] Such a small set of test instances facilitates fast development cycles and experimentation, but can (and, as we will see, did) lead to over-tuning.

Quickly, it became clear that implementation optimization gave more consistent speedups than manual parameter optimization. The variations due to different parameter settings were huge, but no consistent pattern could be identified manually. The algorithm designer even found one case (Alloy analyzer instance handshake.als.3 (Jackson, 2000)) where the difference of floating point rounding errors between Intel's non-standard 80-bit and IEEE 64-bit precision resulted in an extremely large difference in the runtimes on the same processor. The same instance was solved in 0.34 sec with 80-bit precision and timed out after $6\,000$ sec with 64-bit precision. The difference in rounding initially caused minor differences in variable activities, which are used to compute the dynamic decision ordering. Those minor differences quickly diverged, pushing the solver into two completely different parts of search space. Since most parameters influence the decision heuristics in some way, the solver can be expected to be equally sensitive to parameter changes.[4] Given the costly and tedious nature of the process, no further manual parameter optimization was performed after finding a configuration that

[2] In addition, SPEAR has several enhancements for software verification, such as support for modular arithmetic constraints (Babić and Musuvathi, 2005), incrementality to enable structural abstraction/refinement (Babić and Hu, 2007b), and a technique for identifying context-insensitive invariants to speed up solving multiple queries that share common structure (Babić and Hu, 2007a). These components are, however, not used in our experiments with the core SAT solver.

[3] Small instances were selected because CALYSTO tends to occasionally generate very hard instances that would not be solved within a reasonable amount of time.

[4] This emphasizes the need to find a parameter configuration that lead to more robust performance, with different random seeds, as well as across instances.



(a) Bounded model checking (b) Software verification

Figure 6.1: MiniSAT 2.0 *vs* SPEAR using its original, manually-configured default parameter settings. (a) The two solvers performed comparably on bounded model checking instances, with average runtimes of 298 seconds (MiniSAT) *vs* 341 seconds (SPEAR) for the instances solved by both algorithms. (b) Performance on easy and medium software verification instances was comparable, but MiniSAT scaled better for harder instances. The average runtimes for instances solved by both algorithms were 30 seconds (MiniSAT) and 787 seconds (SPEAR).

seemed to work well on the chosen test set.

6.2.2 Initial Performance Assessment

At this point, the author of this thesis got involved since SPEAR promised to be an ideal challenging testbed for automated algorithm configuration. As a first step, we established the baseline performance of this manually-configured version of SPEAR. For this purpose, we compared it against MiniSAT 2.0 (Eén and Sörensson, 2003), the winner of the industrial category of the 2005 SAT Competition and of the 2006 SAT Race. (This study was done in early 2007, at which time MiniSAT 2.0 was the clear state-of-the-art algorithm for industrial instances.) Throughout our case study, we used the two VERIFICATION benchmarks sets BMC and SWV described in Section 3.5.5: two benchmarks sets of industrial problem instances, namely 754 BMC instances from IBM created by Zarpas (2005) and 604 verification conditions generated by the CALYSTO static checker (Babić and Hu, 2007b).

As can be seen from the runtime correlation plots shown in Figure 6.1, MiniSAT 2.0 and the manually-configured version of SPEAR performed quite similarly for bounded model checking and easy software verification instances. For difficult software verification instances, however, MiniSAT clearly performed better. We expect that this effect was due to the focus of manual configuration on a small number of rather easy instances.

6.3 Automated Configuration of SPEAR

While parameter optimization typically ends with the identification of a well-performing default setting, this was only the starting point for our automatic configuration experiments.

6.3.1 Parameterization of SPEAR

The availability of an automatic configuration procedure encouraged its developer, Domagoj Babić, to parameterize many aspects of SPEAR. While the first versions of SPEAR we used for preliminary algorithm configuration experiments only exposed a few important parameters, the results of automated configuration of those first versions prompted its developer to expose more and more search parameters, up to the point where not only every single hard-coded parameter was exposed, but also a number of new parameter-dependent features were incorporated. This process not only significantly improved SPEAR's performance, but also has driven the development of SPEAR itself.

The resulting version of the SPEAR SAT solver, version 1.2.1.1, was the one we used throughout this thesis. It has 26 parameters:

- 7 types of heuristics (with the number of different heuristics available shown in parentheses):
  - Variable selection heuristics (20)
  - Heuristics for sorting learned clauses (20)
  - Heuristics for sorting original clauses (20)
  - Resolution ordering heuristics (20)
  - Value selection heuristics (7)
  - Clause deletion heuristics (3)
  - Resolution heuristics (3)
- 12 double-precision floating point parameters, including variable and clause decay, restart increment, variable and clause activity increment, percentage of random variable and value decisions, heating/cooling factors for the percentage of random choices, etc.
- 4 integer parameters which mostly control restarts and variable/clause elimination.
- 3 Boolean parameters which enable/disable simple optimizations such as the pure literal rule.

For each of SPEAR's floating point and integer parameters we chose lower and upper bounds on reasonable values and considered a number of values spread uniformly across the respective interval. This number ranged from three to eight, depending on our intuition about importance of the respective parameter. The total number of possible combinations after this discretization was $3.78 \cdot 10^{18}$. Exploiting conditional parameters reduced this to $8.34 \cdot 10^{17}$ distinct parameter configurations.

We then performed two sets of experiments: automated configuration of SPEAR on a general set of instances for the 2007 SAT competition, and application-specific configuration for two real-world benchmark sets.

6.3.2 Configuration for the 2007 SAT Competition

The first round of automatic algorithm configuration was performed in the context of preparing a version of SPEAR for submission to the 2007 SAT Competition. We used this as a case study in algorithm configuration for real-world problem domains. The algorithm developer provided an executable of SPEAR and information about its parameters as well as reasonable values for each of them. The default parameter configuration, however, was not revealed. The goal of this study was to see whether the performance achieved with automatic methods could rival the performance achieved by the manually-engineered default parameters.

Since the optimization objective was to achieve good performance on the industrial benchmarks of the 2007 SAT Competition (which were not disclosed before the solver submission deadline), we used a collection of instances from previous competitions for configuration: the 176 industrial instances from the 2005 SAT Competition, the 200 instances from the 2006 SAT Race, as well as 30 SWV instances generated by the CALYSTO static checker. A subset of 300 randomly selected instances was used for training, and the remaining 106 instances were used as a test set. Since the 2007 SAT competition scoring function rewarded per-instance performance relative to other solvers, the optimization objective used in this phase was geometric mean speedup over SPEAR with the (manually-optimized) default parameter settings.

We ran a single run of BASICILS(300) for three days on the 300 designated training instances. (FOCUSEDILS had not yet been developed at this time.) We refer to the parameter configuration thus identified as Satcomp. During configuration, we took the risk of setting a low captime of $\kappa_{max} = 10$ seconds in order to save time. This exposed us to the risk of over-tuning the solver for good performance on short runs but poor performance on longer runs, and we expected that parameter configuration Satcomp might be too aggressive and hence perform poorly on harder instances. Indeed, when presented with the automatically-configured parameter configuration, the algorithm designer observed that the multiplicative factor for increasing the allowed number of learned clauses was quite large (1.3) and feared that this "might cause the solver to run really well on small instances and really terribly on large ones".

In contrast, our experimental results showed that configuration Satcomp performed very well, both on the data from previous SAT competitions and on the BMC and SWV benchmark sets. Figure 6.2 compares the performance of SPEAR's default and configuration Satcomp using $1\,000$ seconds as a timeout for each of the 300 instances used for SAT competition configuration as well as the 106 held-out test instances. We plot training and test instances into the same figure in order to enable a visual check for over-tuning. Qualitatively, the results for training and test instances are comparable (if anything, the improvements due to automatic configuration seem a little *larger* for the test set instances). Quantitatively, while the SPEAR default timed out on 96 instances, configuration Satcomp only timed out on 85 (74 instances remained unsolved by either approach). For the remaining instances, Satcomp achieved a geometric mean speedup of 21%, with a trend to perform better for larger instances. Figure 6.3 demonstrates that this speedup and the favourable scaling behaviour carried over to both our verification benchmark sets: Satcomp performed better than the SPEAR default on BMC (with an average speedup factor of about 1.5) and clearly dominated it for SWV (with an average speedup factor of about 78). The fact that these empirical results contradicted the

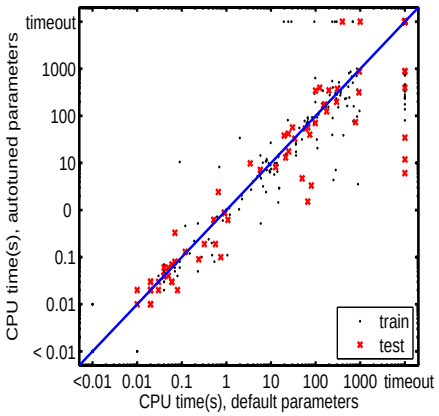


Figure 6.2: Improvements by automated parameter optimization for 2007 SAT competition. Performance of SPEAR with automatically-configured parameter configuration `Satcomp` parameters vs. its default. We plot performance for both training and test instances. Timeouts after 1 000 seconds are plotted at 10 000 seconds for visual distinction from runs taking close to 1 000 seconds.

algorithm designer's intuition illustrates clearly the limitations of even an expert's ability to comprehend the complex interplay between the many parameters of a sophisticated heuristic algorithm such as SPEAR.

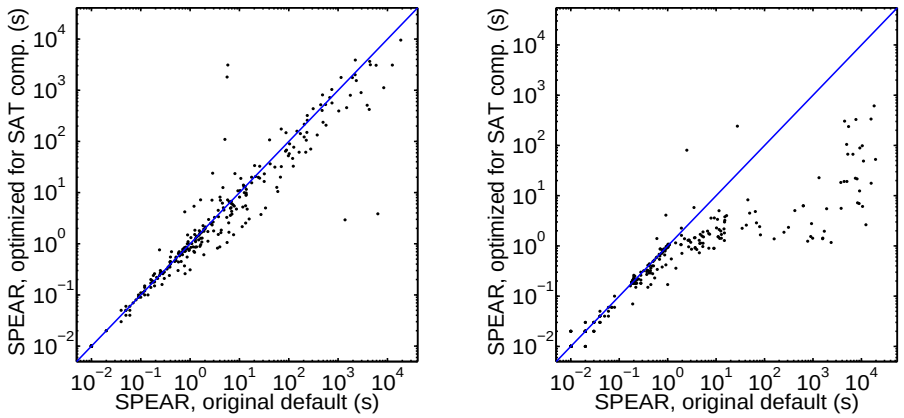
Based on these improvements, SPEAR performed very well in the 2007 SAT competition.[5] We submitted three versions of SPEAR (Babić and Hutter, 2007): the manually-configured default, version `Satcomp` and a newer version with further code optimizations.[6] As can be seen from the data available for the first stage on industrial instances, SPEAR placed 17^{th} with its default configuration (solving 82 instances), 8^{th} with configuration `Satcomp` (solving 93 instances), and 5^{th} with the further-optimized version (solving 99 instances). For reference, MiniSAT—which eventually won three medals in the industrial category—placed 6th in this first stage (solving 97 instances). For a convenient summary of those results, see the bottom of http://www.satcompetition.org/2009/spec2009.html. SPEAR was not allowed to participate in the second round of this competition (and thus was not eligible to win any medals) since its developer preferred to not make the source code available.

6.3.3 Configuration for Specific Applications

While general algorithm configuration on a mixed set of instances as performed for the 2007 SAT competition resulted in a solver with strong overall performance, in practice, one often mostly cares about excellent performance on a specific type of instances, such as `BMC` or `SWV`. For this reason we performed a second set of experiments — configuring SPEAR for these two

[5] See http://www.cril.univ-artois.fr/SAT07.

[6] Note that all results we report for SPEAR, with default or optimized parameter configurations, use version 1.2.1.1 of SPEAR, the version that already includes these code optimizations.



(a) Bounded model checking (b) Software verification

Figure 6.3: Improvements by automated parameter optimization on a mix of industrial instances: SPEAR with the original default parameter configuration *vs* SPEAR with configuration `Satcomp`. (a) Even though a few instances could be solved faster with the SPEAR default, parameter configuration `Satcomp` was considerably faster on average (mean runtime 341 *vs* 223 seconds). Note that speedups were larger than they may appear in the log-log plot: for the bulk of the instances `Satcomp` was about twice as fast. (b) `Satcomp` improved much on the scaling behaviour of the SPEAR default, which failed to solve four instances in 10 000 seconds. Mean runtimes on the remaining instances were 787 seconds *vs* 10 seconds, a speedup factor of 78.

specific sets of problems.

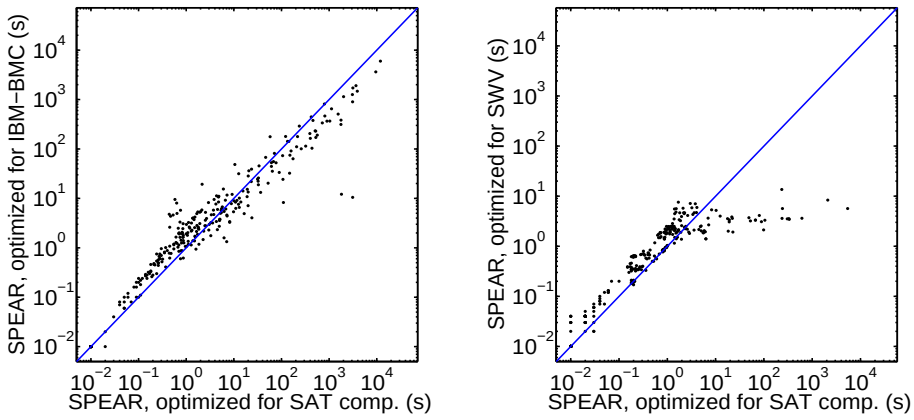
For both sets, we chose penalized average runtime (PAR) with penalization constant 10 as our optimization objective. As the training captime we chose 300 seconds, which according to SPEAR's internal book-keeping mechanisms turned out to be sufficient for exercising all techniques implemented in the solver.

In order to speed up the optimization, in the case of `BMC` we removed 95 hard instances from the training set that could not be solved by SPEAR with its default parameter configuration within one hour, leaving 287 instances for training. Note the relation to our discussion on selecting the most useful instances for comparing parameter configurations in Section 4.3. Here, we performed that step manually.

We performed parameter optimization by executing 10 runs of FOCUSEDILS for three days in the case of `SWV` and for two days for `BMC`. We performed these runs in parallel, on different machines, and for each instance set, picked the parameter configuration with the best training performance.

Figure 6.4 demonstrates that these application-specific parameter configurations further improved over the optimized configuration for the SAT competition, `Satcomp`. SPEAR's performance was boosted for both application domains, by an average factor of over 2 for `BMC` and over 20 for `SWV`. The scaling behaviour also clearly improved, especially for `SWV`.

Figure 6.5 shows the total effect of automatic configuration by comparing the performance



(a) Bounded model checking (b) Software verification

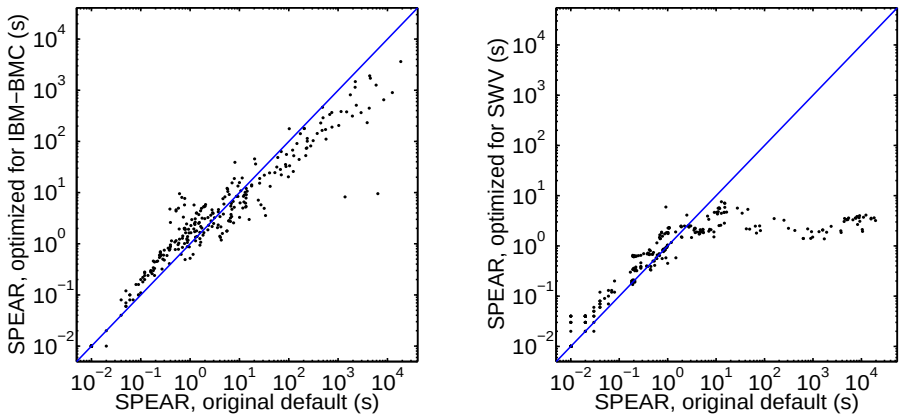
Figure 6.4: Improvements by automated parameter optimization on specific instance distributions: SPEAR with configuration `Satcomp` *vs* SPEAR with parameters optimized for the specific applications `BMC` and `SWV`. Results are on independent test sets disjoint from the instances used for parameter optimization. (a) The parameter configuration configured for set `BMC` solved four instances for which configuration `Satcomp` timed out after 10 000 seconds. For the remaining instances, mean runtimes were 223 seconds (`Satcomp`) and 96 seconds (configured for `BMC`), a speedup factor of about 2.3. (b) Both parameter settings solved all 302 instances, mean runtimes were 36 seconds (`Satcomp`) and 1.5 seconds (configured for `SWV`), a speedup factor of 24.

of SPEAR with the (manually-optimized) default settings against that achieved when using the automatically-found parameter configurations for the `BMC` and `SWV` benchmark sets. For both sets, the scaling behaviour of the automatically-found configuration was much better and on average large speedups were achieved: the speedup factors were as high as 4.5 for `BMC` and 500 for `SWV`. For the `SWV` benchmark set, SPEAR with default parameters timed out on four test instances after 10 000 seconds, while the automatically-found configuration solved every instance in less than 20 seconds.

A final comparison against our baseline, MiniSAT 2.0, demonstrates that the configured SPEAR versions performed much better. Figure 6.6 shows average speedup factors of 4.8 for `BMC` and of 100 for `SWV`. For both domains, SPEAR showed much better scaling behavior for harder instances. Table 6.1 summarizes the performance of MiniSAT 2.0 and SPEAR with parameter settings default, `Satcomp`, and the settings specifically configured for `BMC` and `SWV`.[7]

After the completion of the case study presented in this chapter, we used PARAMILS to optimize SPEAR for Satisfiability Modulo Theories (SMT) instances. Performance did not

[7]The averages in this table differ from the runtimes given in the captions of Figures 6.1-6.5, because averages are taken with respect to different instance sets. For each solver, this table takes averages over all instances solved by that solver, whereas the figure captions state averages over the instances solved by both solvers compared in the respective figure.



(a) Bounded model checking (b) Software verification

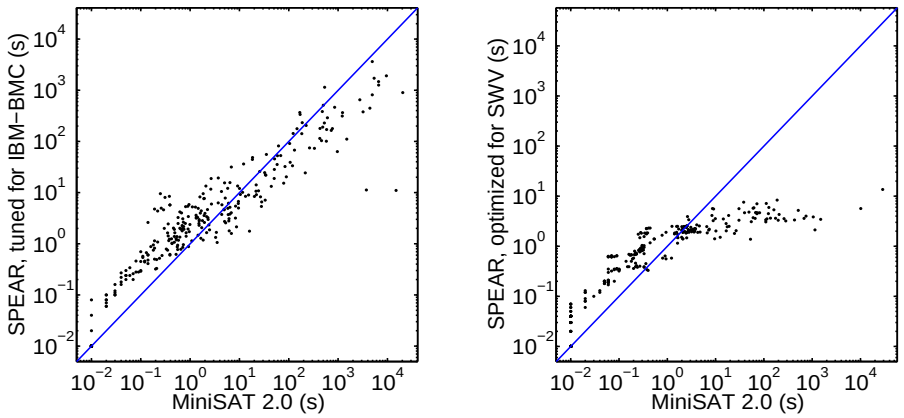
Figure 6.5: Overall improvements achieved by automatic configuration: SPEAR with its manually engineered default parameter configuration *vs* the optimized versions for sets BMC and SWV. Results are on test sets disjoint from the instances used for parameter optimization. (a) The default timed out on 90 instances after 10 000 seconds, while the configured configuration solved four additional instances. For the instances that the default solved, mean runtimes are 341 seconds (default) and 75 seconds (automatically-configured), a speedup factor of 4.5. (b) The default timed out on four instances after 10 000 seconds, the configured configuration solved all instances in less then 20 seconds. For the instances that the default solved, mean runtimes are 787 seconds (default) and 1.35 seconds (automatically-configured), a speedup factor of over 500.

improve over the configuration found for benchmark set SWV, but that parameter configuration was already strong enough to win the quantifier-free bit-vector arithmetic category of the 2007 Satisfiability Modulo Theories Competition.[8]

6.4 Discussion

The automated configuration of SPEAR provided its developer, Domagoj Babić, with a number of insights into properties of the benchmark instances used in our study. Since these insights are *not* part of the contributions of this thesis we only briefly summarize the most interesting parts here for completeness, focusing on the SWV benchmark set (for full details, see Hutter et al., 2007a). The algorithm designer combined knowledge about the generation process of the software verification instances in set SWV with the knowledge of which parameter configuration performed well for these instances, yielding interesting conclusions. The best-found configuration for benchmark set SWV uses an activity-based heuristic that resolves ties by picking the variable with the larger product of occurrences. This heuristic might seem too aggressive, but helps the solver to focus on the most frequently used common subexpressions. It seems that a relatively small number of expressions play a crucial role in (dis)proving each

[8] See http://www.smtcomp.org/2007/.



(a) Bounded model checking (b) Software verification

Figure 6.6: Final comparison against baseline: MiniSAT 2.0 *vs* the configured versions of SPEAR for BMC and SWV. Results are on independent test sets disjoint from the instances used for parameter optimization. (a) MiniSAT timed out for 88/377 test instances after 10,000 seconds, SPEAR configured for BMC instances could solve an additional two instances. For the remaining 289 instances, the average runtimes were 361 seconds (MiniSAT) *vs* 75 seconds (SPEAR), a 4.8-fold speedup. (b) Both MiniSAT and SPEAR solved all 302 SWV test instances; the average runtimes were 161.3 seconds for MiniSAT *vs* 1.5 seconds for SPEAR, a hundred-fold speedup.

verification condition, and this heuristic quickly narrows the search down to such expressions. The SWV instances favoured very aggressive restarts (first after only 50 conflicts), which in combination with our experimental results shows that most such instances can be solved quickly if the right order of variables is found. Interestingly, out of seven possible value selection heuristics, one of the simplest was selected for benchmark set SWV (always assign FALSE first). The SWV instances correspond to NULL-pointer dereferencing checks, and this value selection heuristic attempts to propagate NULL values first (all FALSE), which explains its effectiveness.

The use of automated parameter optimization also influenced the design of SPEAR in various ways. An early version of SPEAR featured a rudimentary implementation of clause and variable elimination. Prior to using automated algorithm configuration, these mechanisms did not consistently improve performance, and therefore, considering the complexity of finalizing their implementation, the SPEAR developer considered removing them. However, these elimination techniques turned out to be effective after PARAMILS found good heuristic settings to regulate the elimination process. Another feature that was considered for removal was the pure literal rule, which ended up being useful for BMC instances (but not for SWV). Similarly, manual optimization gave inconclusive results about randomness, but automated optimization found that a small amount of randomness actually did help SPEAR in solving BMC (but not SWV) instances.

From this case study, we can draw a number of general conclusions about algorithm

Solver	Bounded model checking		Software verification	
	#(solved)	runtime for solved	#(solved)	runtime for solved
MiniSAT 2.0	289/377	360.9	302/302	161.3
SPEAR original	287/377	340.8	298/302	787.1
SPEAR general configured	287/377	223.4	302/302	35.9
SPEAR specific configured	291/377	113.7	302/302	1.5

Table 6.1: Summary of results for configuring SPEAR. For each solver and instance set, #(solved) denotes the number of instances solved within a CPU time of 10 hours, and the runtimes are the arithmetic mean runtimes for the instances solved by that solver. (Geometric means were not meaningful here, as all solvers solved a number of easy instances in "0 seconds"; arithmetic means better reflect practical user experience as well.) If an algorithm solves more instances, the shown average runtimes include more, and typically harder, instances. In particular, SPEAR configured for BMC solved two more instances than MiniSAT. On the instances MiniSAT solved, it took an average of 75 seconds, a speedup factor of 4.8. The two additional hard instances pushed SPEAR's average runtime up to 113.7. Likewise, two missing hard instances for the SPEAR default reduce its average runtime below that of MiniSAT; however, as we saw in Figure 6.1, MiniSAT is faster: if these two instances are not counted for MiniSAT either, its average runtime is 298 seconds.

configuration. We believe that the optimized versions of SPEAR resulting from our application of PARAMILS represent a considerable improvement in the state of the art of solving decision problems from hardware and software verification using SAT solvers. Not too surprisingly, our experimental results suggest that optimized search parameters are benchmark dependent—which highlights the advantages of automated algorithm configuration over the conventional manual approach. The fact that SPEAR's designer invested considerable time to manually configure SPEAR prior to applying PARAMILS, combined with the large speedups automated configuration yielded, argues strongly in favour of applying automated algorithm configuration procedures in the development of complex algorithms for solving computationally hard problems. Doing so can save costly human expert time and exploit the full performance potential of a highly parameterized heuristic solver.

6.5 Chapter Summary

In this chapter, we presented a case study for the configuration of SPEAR, a state-of-the-art tree search SAT solver, on industrially-relevant instances. SPEAR has 26 parameters (10 categorical, 12 continuous, 4 integer), which had initially been set by its designer to optimize performance on a mix of bounded model-checking (BMC) and software verification (SWV) instances; manual tuning took about one week.

We applied PARAMILS to optimize SPEAR's performance in two usage scenarios. First, we optimized SPEAR for overall good performance on industrial instances; this resulted in a very competitive solver in the 2007 SAT competition (which couldn't win any medals due to its proprietary source code). This solver achieved a 1.5-fold speedup in average-case performance over the manually-engineered SPEAR default for a set of BMC instances, and a 78-fold speedup for a set of SWV instances. (These results—and all results of this kind in this thesis—are for

disjoint test sets of instances that were unknown to the configuration procedures.)

Second, we achieved even more impressive speedups by optimizing performance on homogeneous instance distributions. When optimizing performance for BMC, we achieved a 4.5-fold speedup over the default. When optimizing for SWV, we achieved a 500-fold speedup; while the SPEAR default timed out on the four hardest SWV test instances after ten hours of runtime, the optimized parameter setting solved each of the test instances within 20 seconds. (Again, the test instances were unknown at the time of configuration.) Importantly, in all cases there was a trend for the automatically-found parameter configurations to perform increasingly better than the default for harder instances.

We demonstrated that the resulting versions of SPEAR considerably outperformed the state-of-the-art industrial solver MiniSAT 2.0, and thereby substantially advanced the state of the art for solving these types of problem instances. The speedups over the previous state of the art were 4.5-fold (BMC) and over 100-fold (SWV).

Chapter 7

Methods II: Adaptive Capping of Algorithm Runs

Don't say you don't have enough time. You have exactly the same number of hours per day that were given to Helen Keller, Pasteur, Michaelangelo, Mother Teresa, Leonardo da Vinci, Thomas Jefferson, and Albert Einstein.
—H. Jackson Brown Jr., American author

In this chapter[1], we improve upon the methods introduced in Chapter 5. In particular, we consider the third dimension of automated algorithm configuration, the cutoff time for each run of the target algorithm. We introduce an effective and simple capping technique that adaptively determines the cutoff time for each run. The motivation for this capping technique comes from a problem encountered by all configuration procedures considered in this thesis: often the search spends a lot of time evaluating parameter configurations that are much worse than other, previously-seen configurations.

Consider, for example, a case where parameter configuration $\boldsymbol{\theta}_1$ takes a total of 10 seconds to solve $N = 100$ instances (i.e., it has a mean runtime of 0.1 seconds per instance), and another parameter configuration $\boldsymbol{\theta}_2$ takes 100 seconds to solve the first of these instances. In order to compare the mean runtimes of $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ based on this set of instances, knowing all runtimes for $\boldsymbol{\theta}_1$, it is not necessary to run $\boldsymbol{\theta}_2$ on all 100 instances. Instead, we can already terminate the first run of $\boldsymbol{\theta}_2$ after $10 + \epsilon$ seconds. This results in a lower bound on $\boldsymbol{\theta}_2$'s mean runtime of $0.1 + \epsilon/100$ since the remaining 99 instances could take no less than zero time. This lower bound exceeds the mean runtime of $\boldsymbol{\theta}_1$, and so we can already be certain that the comparison will favour $\boldsymbol{\theta}_1$. This insight provides the basis for our *adaptive capping* technique.

7.1 Adaptive Capping for RANDOMSEARCH

The simplest case for our adaptive capping technique is RANDOMSEARCH. Whenever this configurator evaluates a new parameter configuration $\boldsymbol{\theta}$, it performs a comparison $better_N(\boldsymbol{\theta}, \boldsymbol{\theta}_{inc})$

[1]This chapter is based on published joint work with Holger Hoos, Kevin Leyton-Brown, and Thomas Stützle (Hutter et al., 2009b,c).

(see Algorithm 5.4 on page 79). Without adaptive capping, these comparisons can take a long time, since runs with poor parameter configurations often take orders of magnitude longer than those with good configurations (see Section 4.3).

For the case of optimizing the mean of non-negative cost functions (such as runtime or solution cost), we implement a bounded evaluation of a parameter configuration $\boldsymbol{\theta}$ based on N runs and a given performance bound in Procedure $objective$ (see Procedure 7.1). This procedure sequentially performs runs for parameter configuration $\boldsymbol{\theta}$ and after each run computes a lower bound on $\hat{c}_N(\boldsymbol{\theta})$ based on the $i \leq N$ runs performed so far. Specifically, for our objective of mean runtime we sum the runtimes of each of the i runs, and divide this sum by N; since all runtimes must be nonnegative, this quantity lower bounds $\hat{c}_N(\boldsymbol{\theta})$. Once the lower bound exceeds the bound passed as an argument, we can skip the remaining runs for $\boldsymbol{\theta}$. In order to pass the appropriate bounds to Procedure $objective$, we need to slightly modify Procedure *better*$_N$ (see Procedure 5.3 on page 78) for adaptive capping. Procedure $objective$ now has a bound as an additional third argument, which is set to ∞ in line 1 of *better*$_N$, and to $\hat{c}_N(\boldsymbol{\theta}_2)$ in line 2. Note that the reason for computing $\hat{c}_N(\boldsymbol{\theta}_2)$ before $\hat{c}_N(\boldsymbol{\theta}_1)$ is that $\boldsymbol{\theta}_2$ is typically a good configuration that can provide a strong bound. For example, in RANDOMSEARCH, $\boldsymbol{\theta}_2$ is always the current incumbent, $\boldsymbol{\theta}_{inc}$.

Because the use of adaptive capping results in the computation of exactly the same function *better*$_N$ as without it, RANDOMSEARCH(N) with capping follow exactly the same search trajectory it would have followed without it, but typically requires much less runtime. Hence, within the same amount of overall running time, this modified RANDOMSEARCH version tends to be able to search a larger part of the parameter configuration space.

Although in this work we focus on the objective of minimizing mean runtime for decision algorithms, we note that our adaptive capping technique can be applied to other configuration objectives. This is straightforward in the case of any other objective that is based on a mean (e.g., mean solution quality). It also works for other configuration objectives, in particular for quantiles, such as the median or the 90% quantile, which we have considered in previous work. In the case of the median, with $M \leq N/2$ runs we can only obtain a trivial bound of zero; however, for $M > N/2$, the $\lceil N/2 \rceil$th-lowest cost of the M runs provides a lower bound on $\hat{c}_N(\boldsymbol{\theta})$. A similar result holds for other quantiles. Interestingly, the more emphasis is put on robustness, the better for our adaptive capping technique. When the Q% runtime quantile is to be minimized, bounds become non-trivial after observing the performance for $M > N/(100 - Q)$ runs. For example, after 11 timeouts at κ for a configuration $\boldsymbol{\theta}$, we know that $\boldsymbol{\theta}'s$ 90% quantile based on 100 runs is at least κ.

We now present evidence demonstrating that adaptive capping speeds up RANDOMSEARCH. Figure 7.1 compares training performance of RANDOMSEARCH on two configuration scenarios with and without adaptive capping, and Table 7.1 quantifies the speedups for all BROAD scenarios. Overall, adaptive capping enabled RANDOMSEARCH to evaluate between 2.8 and 33 times as many parameter configurations in the same time, while yielding exactly the same result based on the same number of configurations considered (given the same random seeds). This improved training performance statistically significantly in all BROAD scenarios. (As before, we use training rather than test performance here to yield a lower variance estimate.) Note that the speedups are small at the beginning and grow as

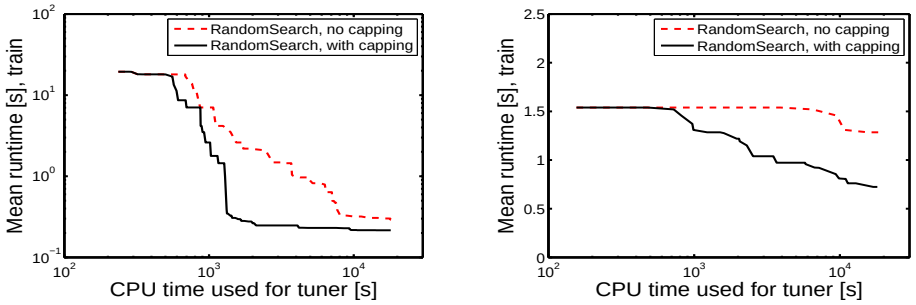
Procedure 7.1: $objective(\boldsymbol{\theta}, N,$ optional parameter $bound)$
This procedure computes $\hat{c}_N(\boldsymbol{\theta})$, either by performing new runs or by exploiting previous cached runs; 'bound' specifies a bound on the computation to be performed. When this parameter is not specified, the bound is taken to be ∞. For each $\boldsymbol{\theta}$, $N(\boldsymbol{\theta})$ is the number of runs performed for $\boldsymbol{\theta}$, *i.e.*, the length of the global array $\mathbf{R}_{\boldsymbol{\theta}}$. We count runtimes for unsuccessful runs according to the PAR criterion.

Input : Parameter configuration $\boldsymbol{\theta}$, number of runs, N, optional bound $bound$
Output : $\hat{c}_N(\boldsymbol{\theta})$ if $\hat{c}_N(\boldsymbol{\theta}) \leq bound$, otherwise a large constant (maxPossibleObjective) plus the number of instances that remain unsolved when the bound was exceeded
Side Effect: Adds runs to the global cache of performed algorithm runs, $\mathbf{R}_{\boldsymbol{\theta}}$; updates global incumbent, $\boldsymbol{\theta}_{inc}$

// ===== Maintain invariant: $N(\boldsymbol{\theta}_{inc}) \geq N(\boldsymbol{\theta})$ for any $\boldsymbol{\theta}$
if $\boldsymbol{\theta} \neq \boldsymbol{\theta}_{inc}$ *and* $N(\boldsymbol{\theta}_{inc}) < N$ **then**
  $\hat{c}_N(\boldsymbol{\theta}_{inc}) \leftarrow objective(\boldsymbol{\theta}_{inc}, N, \infty)$ *// Adds $N - N(\boldsymbol{\theta}_{inc})$ runs to $\mathbf{R}_{\boldsymbol{\theta}_{inc}}$*
// ===== For aggressive capping, update bound.
if *Aggressive capping* **then** $bound \leftarrow \min(bound, bm \cdot \hat{c}_N(\boldsymbol{\theta}_{inc}))$
// ===== Update the run results in tuple $\mathbf{R}_{\boldsymbol{\theta}}$.
for $i = 1...N$ **do**
  sum_runtime $\leftarrow$ sum of runtimes in $\mathbf{R}_{\boldsymbol{\theta}}[1], \ldots, \mathbf{R}_{\boldsymbol{\theta}}[i-1]$ *// Tuple indices starting at 1.*
  $\kappa'_i \leftarrow \max(\kappa, N \cdot bound -$ sum_runtime$)$
  if $N(\boldsymbol{\theta}) \geq i$ **then** $(\boldsymbol{\theta}, \pi_i, \kappa_i, o_i) \leftarrow \mathbf{R}_{\boldsymbol{\theta}}[i]$
  if $N(\boldsymbol{\theta}) \geq i$ **and** *($\kappa_i \geq \kappa'_i$* **and** *$o_i =$ "unsuccessful")* **or** *($\kappa_i < \kappa'_i$* **and** $o_i \neq$ *"unsuccessful"))* **then**
    $o'_i \leftarrow o_i$ *// Previous run is longer yet unsuccessful or shorter yet successful ⇒ can re-use result*
  else
    $o'_i \leftarrow$ objective from a newly executed run of $\mathcal{A}(\boldsymbol{\theta})$ on instance π_i with seed s_i and captime κ_i
  $\mathbf{R}_{\boldsymbol{\theta}}[i] \leftarrow (\boldsymbol{\theta}, \pi_i, \kappa'_i, o'_i)$
  if $1/N \cdot (sum\_runtime + o'_i) > bound$ **then**
    return maxPossibleObjective + $(N+1) - i$
if $N = N(\boldsymbol{\theta}_{inc})$ **and (sum of runtimes in $\mathbf{R}_{\boldsymbol{\theta}}$) < (sum of runtimes in $\mathbf{R}_{\boldsymbol{\theta}_{inc}}$) then**
  $\boldsymbol{\theta}_{inc} \leftarrow \boldsymbol{\theta}$
return $1/N \cdot$ (sum of runtimes in $\mathbf{R}_{\boldsymbol{\theta}}$)

configuration time increases. This is because every time a better incumbent is found, the runtime needed for ruling out new configurations decreases.

Intuitively, the speedups are greatest for scenarios with large differences between very good and very bad configurations. This is reflected in our results. For example, consider scenarios SPEAR-SWGCP and SAPS-SWGCP, for which the speedups were the least and the most pronounced. While the default configuration yielded considerably faster runs for SPEAR-SWGCP than for SAPS-SWGCP, the best found configurations were more than 10 times faster for SAPS-SWGCP. Thus, there is a much larger potential for time savings in scenario SAPS-SWGCP.



(a) SAPS-SWGCP, statistically significant. (b) CPLEX-REGIONS100, statistically significant.

Figure 7.1: Speedup of RANDOMSEARCH due to adaptive capping. For RANDOMSEARCH(100) with and without adaptive capping and for each time step t, we compute training performance $p_{t,train}$ (PAR across 100 training instances, using the procedures' incumbents, $\boldsymbol{\theta}_{inc}(t)$). We plot median $p_{train,t}$ across the 25 runs.

Scenario	Training performance (PAR, CPU seconds)			Avg. # conf. evaluated	
	No capping	Capping	p-value	No capping	Capping
SAPS-SWGCP	0.46 ± 0.34	$\mathbf{0.21 \pm 0.03}$	$\mathbf{1.2 \cdot 10^{-5}}$	60	**2004**
SPEAR-SWGCP	7.02 ± 1.11	$\mathbf{6.70 \pm 1.20}$	$\mathbf{6.1 \cdot 10^{-5}}$	71	**199**
SAPS-QCP	3.73 ± 1.53	$\mathbf{3.40 \pm 1.53}$	$\mathbf{4.0 \cdot 10^{-5}}$	127	**434**
SPEAR-QCP	0.58 ± 0.59	$\mathbf{0.45 \pm 0.51}$	$\mathbf{6.0 \cdot 10^{-5}}$	321	**1951**
CPLEX-REGIONS100	1.29 ± 0.28	$\mathbf{0.70 \pm 0.12}$	$\mathbf{1.8 \cdot 10^{-5}}$	45	**1004**

Table 7.1: Speedup of RANDOMSEARCH due to adaptive capping. We performed 25 runs of RANDOMSEARCH(100) with and without adaptive capping and computed their training performance $p_{train,t}$ (PAR across 100 training instances, using the procedures' final incumbents, $\boldsymbol{\theta}_{inc}(t)$) for a configuration time of $t = 18\,000s = 5h$. We report mean $\pm$ stddev of $p_{train,t}$ across the 25 runs, p-values for comparing the configurators (as judged by a paired Max-Wilcoxon test, see Section 3.6.2), and the average number of configurations they evaluated. Note that the speedup is highly significant in all scenarios.

7.2 Adaptive Capping in BASICILS

Now we consider the application of adaptive capping to BASICILS. First, we introduce a trajectory-preserving capping (**TP capping**) technique that maintains BASICILS' trajectory. Then we modify this strategy heuristically to perform more aggressive adaptive capping (**Aggr capping**), potentially yielding even better performance in practice.

7.2.1 Trajectory-Preserving Capping

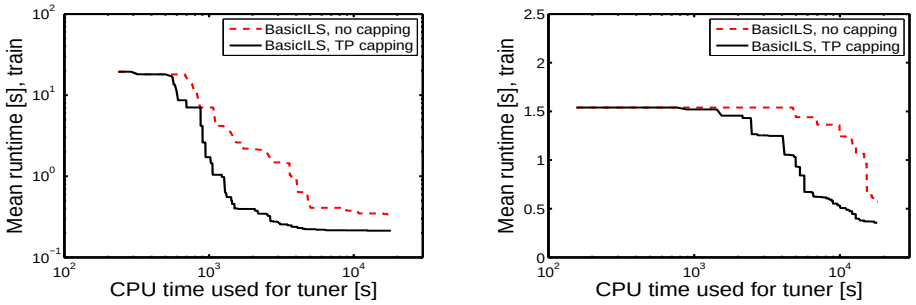
In essence, adaptive capping for BASICILS can be implemented by applying the same changes to Procedure *better*$_N$ as described above for RANDOMSEARCH. In fact, for SIMPLELS (one iteration of BASICILS), no changes are required at all. However, in BASICILS—unlike in RANDOMSEARCH and SIMPLELS—the current configuration is not always compared to the incumbent, $\boldsymbol{\theta}_{inc}$. In contrast, the second argument of Procedure *better*$_N$ is typically the best configuration encountered in the current ILS iteration (where a new ILS iteration begins after each perturbation). During most of the search process, this configuration is quite good; thus, adaptive capping can lead to significant time savings.

7.2.2 Aggressive Capping

There are, however, cases where the trajectory-preserving capping mechanism described above is less efficient in combination with the PARAMILS framework than it could be. This is because the best configuration in the current ILS iteration can be substantially worse than the overall incumbent. (Note that in RANDOMSEARCH and SIMPLELS every pairwise comparison of configurations involves the current incumbent, such that TP capping and the aggressive variant discussed here are identical.) In particular, at the beginning of a new iteration it is initialized to a potentially very poor configuration resulting from a random perturbation. In the frequent case that this configuration performs poorly, the capping criterion uses a much weaker bound than the performance of the overall incumbent. To counteract this effect, we introduce a more aggressive capping strategy that can terminate the evaluation of a poorly-performing configuration at any time. In this heuristic extension of our adaptive capping technique, we bound the evaluation of *any* parameter configuration by the performance of the incumbent parameter configuration multiplied by a factor that we call the *bound multiplier*, bm. When a comparison between any two parameter configurations, $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$, is performed and the evaluations of both are terminated preemptively, the configuration having solved more instances within the allowed time is taken to be the better one. (This behaviour is achieved by line 14 in Procedure $objective$, which keeps track of the number of instances solved when exceeding the bound.) Ties are broken to favour moving to a new parameter configuration instead of staying with the current one.

Depending on the bound multiplier, the use of this aggressive capping mechanism may change the search trajectory of BASICILS. For $bm = \infty$, the heuristic method reduces to our trajectory-preserving method, while a very aggressive setting of $bm = 1$ means that once we know a parameter configuration to be worse than the incumbent, we stop its evaluation. In our experiments we set $bm = 2$: once the lower bound on the performance of a configuration exceeds twice the performance of the incumbent solution, its evaluation is terminated. (In Section 8.2, we revisit this choice of $bm = 2$, optimizing the parameters of PARAMILS itself.)

In four of our five BROAD scenarios, TP capping significantly sped up SIMPLELS and BASICILS. Figure 7.2 illustrates this speedup for two configuration scenarios. In both cases, capping led to substantial speedups and, as a consequence, improved training performance at the end of the trajectory. Table 7.2 quantifies these improvements for all five BROAD configuration scenarios. TP capping enabled up to four times as many ILS iterations and



(a) SAPS-SWGCP, statistically significant, (b) CPLEX-REGIONS100, statistically significant.

Figure 7.2: Speedup of BASICILS due to adaptive capping. For BASICILS(100) without adaptive capping and with TP capping and for each time step t, we compute training performance $p_{t,train}$ (PAR across 100 training instances, using the procedures' incumbents, $\boldsymbol{\theta}_{inc}(t)$). We plot median $p_{train,t}$ across the 25 runs.

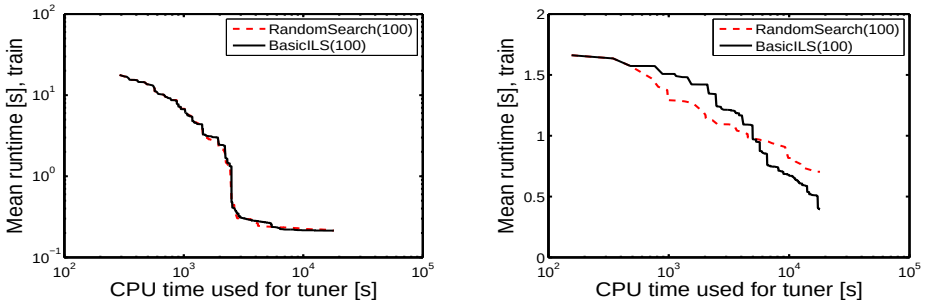
Scenario	Training performance (PAR, CPU seconds)			Avg. # ILS iterations	
	No capping	TP capping	p-value	No capping	TP capping
SAPS-SWGCP	0.38 ± 0.19	$\mathbf{0.24 \pm 0.05}$	$\mathbf{6.1 \cdot 10^{-5}}$	3	**12**
SPEAR-SWGCP	6.78 ± 1.73	$\mathbf{6.65 \pm 1.48}$	**0.01**	1	1
SAPS-QCP	3.19 ± 1.19	$\mathbf{2.96 \pm 1.13}$	$\mathbf{9.8 \cdot 10^{-4}}$	6	**10**
SPEAR-QCP	0.361 ± 0.39	$\mathbf{0.356 \pm 0.44}$	0.66	2	**3**
CPLEX-REGIONS100	0.67 ± 0.35	$\mathbf{0.47 \pm 0.26}$	$\mathbf{7.3 \cdot 10^{-4}}$	1	1

Table 7.2: Speedup of BASICILS due to adaptive capping. We performed 25 runs of BASICILS(100) with and without adaptive capping and computed their training performance $p_{train,t}$ (PAR across 100 training instances, using the procedures' final incumbents, $\boldsymbol{\theta}_{inc}(t)$) for a configuration time of $t = 18\,000s = 5h$. We report mean $\pm$ stddev of $p_{train,t}$ across the 25 runs, p-values for comparing the configurators, and the average number of ILS iterations they performed.

improved average performance in all scenarios. The improvement was statistically significant in all scenarios except SPEAR-QCP.

In two scenarios (SPEAR-SWGCP and CPLEX-REGIONS100), BASICILS did not finish its first iteration and was thus identical to SIMPLELS. In both of these scenarios, training performance improved significantly due to capping. For CPLEX-REGIONS100, this improvement was also quite substantial, leading to a 1.4-fold speedup of CPLEX (compared to its performance with the configurations found without capping).

Aggressive capping further improved BASICILS performance for scenario SAPS-SWGCP. Here, it increased the number of ILS iterations completed within the configuration time from 12 to 219, leading to a significant improvement in performance. For the other configuration scenarios, differences were not significant.



(a) SAPS-SWGCP, not statistically significant. (b) CPLEX-REGIONS100, statistically significant.

Figure 7.3: Comparison of BASICILS(100) and RANDOMSEARCH(100), with adaptive capping. This is the analogue of Figure 5.1, but with adaptive capping (Aggr capping with $bm = 2$).

Scenario	Training performance (PAR, CPU seconds)		p-value
	RANDOMSEARCH(100)	BASICILS(100)	
SAPS-SWGCP	0.215 ± 0.034	$\mathbf{0.214 \pm 0.034}$	0.35
SPEAR-SWGCP	6.70 ± 1.20	$\mathbf{6.65 \pm 1.48}$	0.51
SAPS-QCP	3.48 ± 1.24	$\mathbf{3.02 \pm 1.15}$	$\mathbf{5.2 \cdot 10^{-5}}$
SPEAR-QCP	0.45 ± 0.51	$\mathbf{0.36 \pm 0.41}$	0.28
CPLEX-REGIONS100	0.73 ± 0.1	$\mathbf{0.47 \pm 0.26}$	**0.0013**

Table 7.3: Comparison of RANDOMSEARCH(100) and BASICILS(100), with adaptive capping. This is the analogue of Table 5.1, but with adaptive capping (Aggr capping with $bm = 2$).

Since adaptive capping improved RANDOMSEARCH more than BASICILS, one might now wonder how our previous comparison between these approaches changes when adaptive capping is used. We found that the gap between the techniques narrowed, but that the qualitative difference persisted. Figure 7.3 compares the training performance of RANDOMSEARCH and BASICILS for two tuning scenarios; it is the analogue of Figure 5.1 but with adaptive capping. We note that the capping improved RANDOMSEARCH more than BASICILS, and that for scenario SAPS-SWGCP their performances are visually indistinguishable. For scenario CPLEX-REGIONS100, RANDOMSEARCH outperformed BASICILS for short configuration times, but with larger configuration times BASICILS still performed much better. Table 7.3 compares the training performance of RANDOMSEARCH(100) and BASICILS(100) with adaptive capping enabled. Comparing this to the performance differences without capping (see Table 5.1), now BASICILS only performed significantly better in two of the five BROAD scenarios (as opposed to three without capping). The gap between the methods narrowed, but BASICILS still performed better on average for all configuration scenarios.

7.3 Adaptive Capping in FOCUSEDILS

The main difference between BASICILS and FOCUSEDILS is that the latter adaptively varies the number of runs used to evaluate each parameter configuration. This difference complicates, but does not preclude the use of adaptive capping. This is because FOCUSEDILS always compares pairs of parameter configurations based on the same number of runs for each configuration, even though this number can differ from one comparison to the next.

Thus, we can extend adaptive capping to FOCUSEDILS by using separate bounds for every number of runs, N. Recall that FOCUSEDILS never moves from one configuration, $\boldsymbol{\theta}$, to a neighbouring configuration, $\boldsymbol{\theta}'$, without performing at least as many runs for $\boldsymbol{\theta}'$ as have been performed for $\boldsymbol{\theta}$. Since we keep track of the performance of $\boldsymbol{\theta}$ with any number of runs $M \leq N(\boldsymbol{\theta})$, a bound for the evaluation of $\boldsymbol{\theta}'$ is always available. Therefore, we can implement both trajectory-preserving and aggressive capping as we did for BASICILS.

As for BASICILS, for FOCUSEDILS the inner workings of adaptive capping are implemented in Procedure $objective$ (see Procedure 7.1). We only need to modify Procedures $better_{Foc}$ and $dominates$ (see Procedures 5.5 and 5.6 on page 82) to call $objective$ with the appropriate bounds. This leads to the following changes. Procedure $dominates$ now takes a bound as an additional argument and passes it on to the two calls to $objective$ in line 2. The two calls of $dominates$ in line 10 of $better_{Foc}$ and the one call in line 11 all use the bound $\hat{c}_{\boldsymbol{\theta}_{max}}$. The three direct calls to $objective$ in lines 8, 9, and 12 use bounds ∞, $\hat{c}_{\boldsymbol{\theta}_{max}}$, and ∞, respectively.

We now evaluate the usefulness of capping for FOCUSEDILS. Training performance is not a useful quantity in the context of comparing different versions of FOCUSEDILS, since the number of target algorithm runs this measure is based on varies widely between runs of the configurator. Instead, we used two other measures to quantify search progress: the number of ILS iterations performed and the number of target algorithm runs performed for the incumbent parameter configuration. Table 7.4 shows these two measures for our five BROAD configuration scenarios and the three capping schemes (none, TP, Aggr). For all BROAD scenarios, on average, TP capping improved both measures, and Aggr capping achieved further improvements. Most of these improvements were statistically significant. Figure 7.4 shows that for two configuration scenarios FOCUSEDILS reached the same solution qualities more quickly with capping than without. (However, after finding the respective configurations, the performance of FOCUSEDILS showed no further noticeable improvement.)

Adaptive capping improved BASICILS somewhat more than FOCUSEDILS, and thereby reduced the gap between the two. In particular, for SAPS-SWGCP, where, even without adaptive capping, FocusedILS achieved the best performance we have encountered for this scenario, BasicILS caught up when using adaptive capping. Similarly, for CPLEX-REGIONS100, FocusedILS already performed very well without adaptive capping while BasicILS did not. Here, BasicILS improved based on adaptive capping, but still could not rival FocusedILS. For the other scenarios, adaptive capping did not affect the relative performance much; compare Tables 5.3 (without capping) and 7.5 (with capping) for details.

Number of ILS iterations performed					
Scenario	No capping	TP capping	p-value	Aggr capping	p-value
SAPS-SWGCP	121 ± 12	166 ± 15	$\mathbf{1.2 \cdot 10^{-5}}$	$\mathbf{244 \pm 19}$	$\mathbf{1.2 \cdot 10^{-5}}$
SPEAR-SWGCP	37 ± 12	43 ± 15	$\mathbf{0.0026}$	$\mathbf{47 \pm 18}$	$\mathbf{9 \cdot 10^{-5}}$
SAPS-QCP	142 ± 18	143 ± 22	0.54	$\mathbf{156 \pm 28}$	$\mathbf{0.016}$
SPEAR-QCP	153 ± 49	165 ± 41	$\mathbf{0.03}$	$\mathbf{213 \pm 62}$	$\mathbf{1.2 \cdot 10^{-5}}$
CPLEX-REGIONS100	36 ± 13	40 ± 16	0.26	$\mathbf{54 \pm 15}$	$\mathbf{1.8 \cdot 10^{-5}}$
Number of runs $N(\boldsymbol{\theta}_{inc})$ performed for the incumbent parameter configuration					
Scenario	No capping	TP capping	p-value	Aggr capping	p-value
SAPS-SWGCP	993 ± 211	1258 ± 262	$\mathbf{4.7 \cdot 10^{-4}}$	$\mathbf{1818 \pm 243}$	$\mathbf{1.2 \cdot 10^{-5}}$
SPEAR-SWGCP	503 ± 265	476 ± 238	(0.58)	$\mathbf{642 \pm 288}$	$\mathbf{0.009}$
SAPS-QCP	1575 ± 385	1701 ± 318	0.065	$\mathbf{1732 \pm 340}$	0.084
SPEAR-QCP	836 ± 509	1130 ± 557	$\mathbf{0.02}$	$\mathbf{1215 \pm 501}$	$\mathbf{0.003}$
CPLEX-REGIONS100	761 ± 215	795 ± 184	0.40	$\mathbf{866 \pm 232}$	0.07

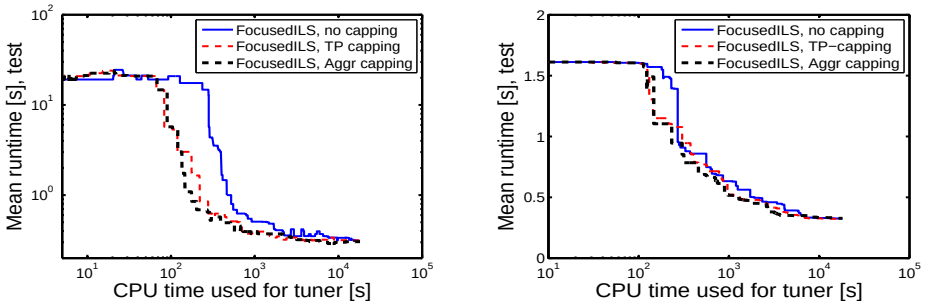
Table 7.4: Speedup of FOCUSEDILS due to adaptive capping. We give the number of ILS iterations performed and the number of runs $N(\boldsymbol{\theta}_{inc})$ performed for the incumbent parameter configuration. We report mean of both of these measures across 25 runs of the configurator without capping, with TP capping, and with Aggr capping, p-values for comparing no capping *vs* TP capping, and no capping *vs* Aggr capping.

7.4 Final Evaluation of Default *vs* Optimized Parameter Configurations

Now that we have introduced all components of our model-free configuration procedures, we conduct a more thorough experimental evaluation. In particular, we show that our configuration procedures lead to very substantial improvements over the default. For each of our five BROAD configuration scenarios, we compare the performance of the respective algorithm's default configuration against the configurations found by BASICILS(100) and FOCUSEDILS, both with adaptive capping. Table 7.5 shows penalized average runtimes; Figure 7.5 demonstrates the magnitude of the speedups when we use larger cutoff times to evaluate configurations.

Table 7.5 shows that both BASICILS and FOCUSEDILS consistently found configurations much better than the default. There was often a rather large variance in performance across configuration runs, and the configuration found in the run with best *training* performance also tended to yield better *test* performance than the others. Thus, we used that configuration as the combined result of the various runs. Note that choosing the configuration with the best training set performance is legitimate since it does not require knowledge of the test set. Of course, the improvements thus achieved come at the price of increased overall running time, but the independent runs of the configurator can easily be performed in parallel.

The speedups this automatically-found parameter configuration achieved over the default are clearest in Figure 7.5. For that figure, runs were allowed to last up to one hour. This reveals the magnitude of the improvements achieved: while the PAR score reported in Table 7.5 counts runtimes larger than $\kappa_{max} = 5$ seconds as $10 \cdot \kappa_{max} = 50$ seconds, here timeouts and mean runtimes for the instances solved by both approaches are reported separately. Speedups were greatest for scenarios SAPS-SWGCP, SAPS-QCP, and SPEAR-QCP: 3540-fold, 416-fold and

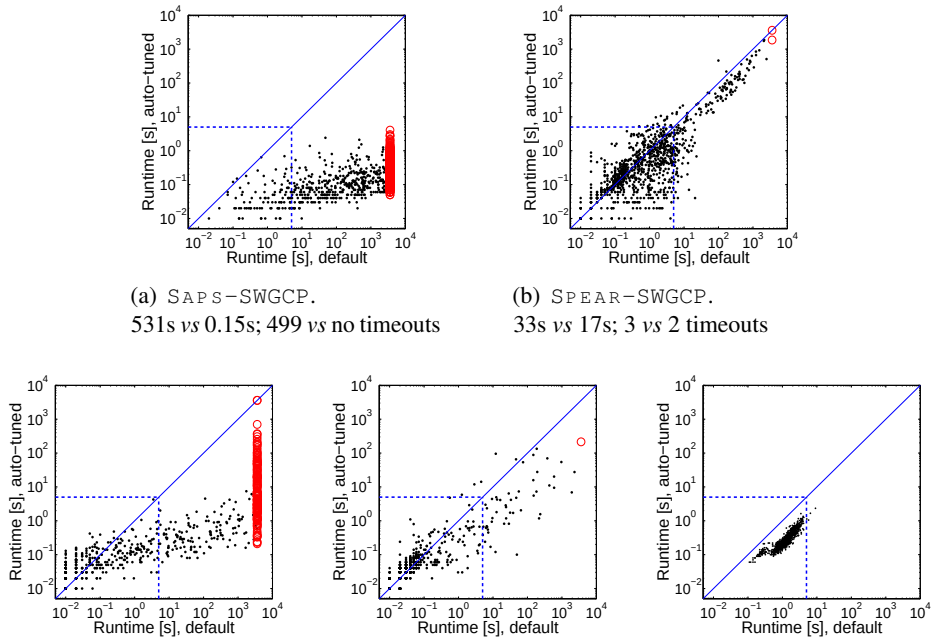


(a) SAPS-SWGCP (b) CPLEX-REGIONS100

Figure 7.4: Speedup of FOCUSEDILS due to adaptive capping. For FOCUSEDILS without adaptive capping, with TP capping and with Aggr capping and for each time step t, we compute test performance $p_{t,test}$ (PAR across 1 000 test instances, using the procedures' incumbents, $\boldsymbol{\theta}_{inc}(t)$). We plot median $p_{test,t}$ across 25 runs of the configurators. The differences at the end of the trajectory were not statistically significant. However, with capping the time required to achieve that quality was lower in these two configuration scenarios. In the other three scenarios, the gains due to capping were smaller.

Scenario	Test performance (PAR, CPU seconds)				
		mean ± stddev. for 25 runs		Run with best training perf.	
	Default	BASICILS	FOCUSEDILS	BASICILS	FOCUSEDILS
SAPS-SWGCP	20.41	0.32 ± 0.06	$\mathbf{0.32 \pm 0.05}$	**0.26**	**0.26**
SPEAR-SWGCP	9.74	$\mathbf{8.05 \pm 0.9}$	8.30 ± 1.06	6.8	**6.6**
SAPS-QCP	12.97	4.86 ± 0.56	$\mathbf{4.70 \pm 0.39}$	4.85	**4.29**
SPEAR-QCP	2.65	1.39 ± 0.33	$\mathbf{1.29 \pm 0.2}$	**1.16**	1.21
CPLEX-REGIONS100	1.61	0.5 ± 0.3	$\mathbf{0.35 \pm 0.04}$	0.35	**0.32**

Table 7.5: Final evaluation of default configuration *vs* configurations found with BASICILS and FOCUSEDILS. We performed 25 runs of the configurators (both with Aggr Capping and $bm = 2$) and computed their test performance $p_{test,t}$ (PAR across 1 000 test instances, using the procedures' final incumbents, $\boldsymbol{\theta}_{inc}(t)$) and training performance $p_{train,t}$ (PAR, across used training instances, using $\boldsymbol{\theta}_{inc}(t)$) for a configuration time of $t = 18\,000s = 5h$. We list test performance of the algorithm default, mean ± stddev of $p_{test,t}$ across the 25 runs for BASICILS(100) & FOCUSEDILS, and $p_{test,t}$ of the BASICILS and FOCUSEDILS runs with lowest *training* performance $p_{train,t}$. Boldface indicates the better of BASICILS and FOCUSEDILS.



(a) SAPS−SWGCP.
531s *vs* 0.15s; 499 *vs* no timeouts

(b) SPEAR−SWGCP.
33s *vs* 17s; 3 *vs* 2 timeouts

(c) SAPS−QCP.
72s *vs* 0.17s; 149 *vs* 1 timeouts

(d) SPEAR−QCP.
9.6s *vs* 0.85s; 1 *vs* 0 timeouts

(e) CPLEX−REGIONS100.
1.61s *vs* 0.32s; no timeouts

Figure 7.5: Comparison of default *vs* automatically-determined parameter configurations for our five BROAD configuration scenarios. Each dot represents one test instance; timeouts (after one CPU hour) are denoted by circles. The dashed line at five CPU seconds indicates the cutoff time of the target algorithm used during the configuration process. The subfigure captions give mean runtimes for the instances solved by both of the configurations (default *vs* optimized), as well as the number of timeouts for each.

11-fold, respectively. The number of timeouts was also greatly reduced (see Figure 7.5).

7.5 Chapter Summary

In this chapter, we focused on the third dimension of algorithm configuration: how to set the cutoff time for each run of the target algorithm? We introduced a novel method for adaptively setting this cutoff time for each algorithm run. To the best of our knowledge, this is the first mechanism of its kind.

Our adaptive capping method is general and can be applied to a variety of configuration procedures. In particular, here we applied it to RANDOMSEARCH, BASICILS, and FOCUSED-ILS. The speedups due to this mechanism were greatest in configuration scenarios with a large spread in the quality of configurations. Without adaptive capping, the search spends most of its time with costly evaluations of poor configurations. With adaptive capping, once good configurations are detected these enable aggressive bounds and can thereby dramatically

speed up the search.

Speedups were largest for RANDOMSEARCH, which only compares parameter configurations to the current incumbent—this allows us always to use the strongest bound known so far, the performance of the incumbent. Here, adaptive capping led to significant and substantial improvements for all five BROAD configuration scenarios. For BASICILS and FOCUSEDILS, we introduced both a trajectory-preserving variant and a heuristic extension. These mechanisms significantly improved BASICILS performance in three of our five BROAD configuration scenarios, on two of them substantially. In the case of FOCUSEDILS, adaptive capping did not lead to significantly better configurations in the end of the trajectory. However, we found that it *did* speed up FOCUSEDILS significantly, enabling it to find configurations of the same quality faster.

Chapter 8

Applications II: Configuration of Various Target Algorithms based on PARAMILS

Science means simply the aggregate of all the recipes that are always successful. All the rest is literature.
—Paul Valéry, French poet

In this chapter[1], we discuss applications of PARAMILS to configure algorithms for a wide variety of computationally hard problems. These problems include SAT, MIP, timetabling, protein folding, and algorithm configuration itself. We first discuss configuration experiments performed by the author of this thesis and then review further work by others. These experiments demonstrate both the generality and maturity of PARAMILS.

8.1 Configuration of CPLEX

When motivating the use of algorithm configuration procedures in Section 1.1, we have prominently discussed CPLEX as an example of an important algorithm with an intimidatingly large number of parameters. Improvements of this solver are highly relevant: with users at more than 1 300 corporations and government agencies, as well as 1 000 universities, it is the most widely-used commercial optimization tool for mathematical programming.[2]

Here, we demonstrate that PARAMILS can improve CPLEX's performance for a variety of interesting benchmark distributions, in particular our `CPLEX` configuration scenarios, defined in Section 3.5.6. To the best of our knowledge, these results, which we first presented in Hutter et al. (2009b), are the first published results for the automated configuration of CPLEX or any algorithm of comparable complexity.[3]

[1] This chapter is based on published joint work with Holger Hoos, Kevin Leyton-Brown, and Thomas Stützle (Hutter et al., 2009b,c).

[2] See http://www.ilog.com/products/cplex/.

[3] Note that the newest CPLEX version, 11.2, contains an automated procedure for optimizing CPLEX's parameters on a given problem set. The inner workings of this mechanism are not disclosed and we have not yet

Scenario	Type of benchmark instances & citation	# training	# test
CPLEX-REGIONS200	Combinatorial Auctions (CATS) (Leyton-Brown et al., 2000)	1 000	1 000
CPLEX-MJA	Machine-Job Assignment (BCOL) (Aktürk et al., 2007)	172	171
CPLEX-CLS	Capacitated Lot Sizing (BCOL) (Atamtürk and Muñoz, 2004)	50	50
CPLEX-MIK	Mixed-integer knapsack (BCOL) (Atamtürk, 2003)	60	60
CPLEX-QP	Quadratic programs from RNA energy parameter optimization (Andronescu et al., 2007)	1 000	1 000

Table 8.1: Overview of our five CPLEX configuration scenarios. BCOL stands for Berkeley Computational Optimization Lab, CATS for Combinatorial Auction Test Suite.

Note that much effort has gone into establishing the CPLEX default:

> "A great deal of algorithmic development effort has been devoted to establishing default ILOG CPLEX parameter settings that achieve good performance on a wide variety of MIP models." (ILOG CPLEX 10.0 user manual, page 247)

Nevertheless, we will demonstrate that our automatically-found configurations performed substantially better than this default. However, our goal is *not* to improve this general-case default, but rather to automatically find better parameter settings for homogeneous subsets of instances.

8.1.1 Experimental Setup

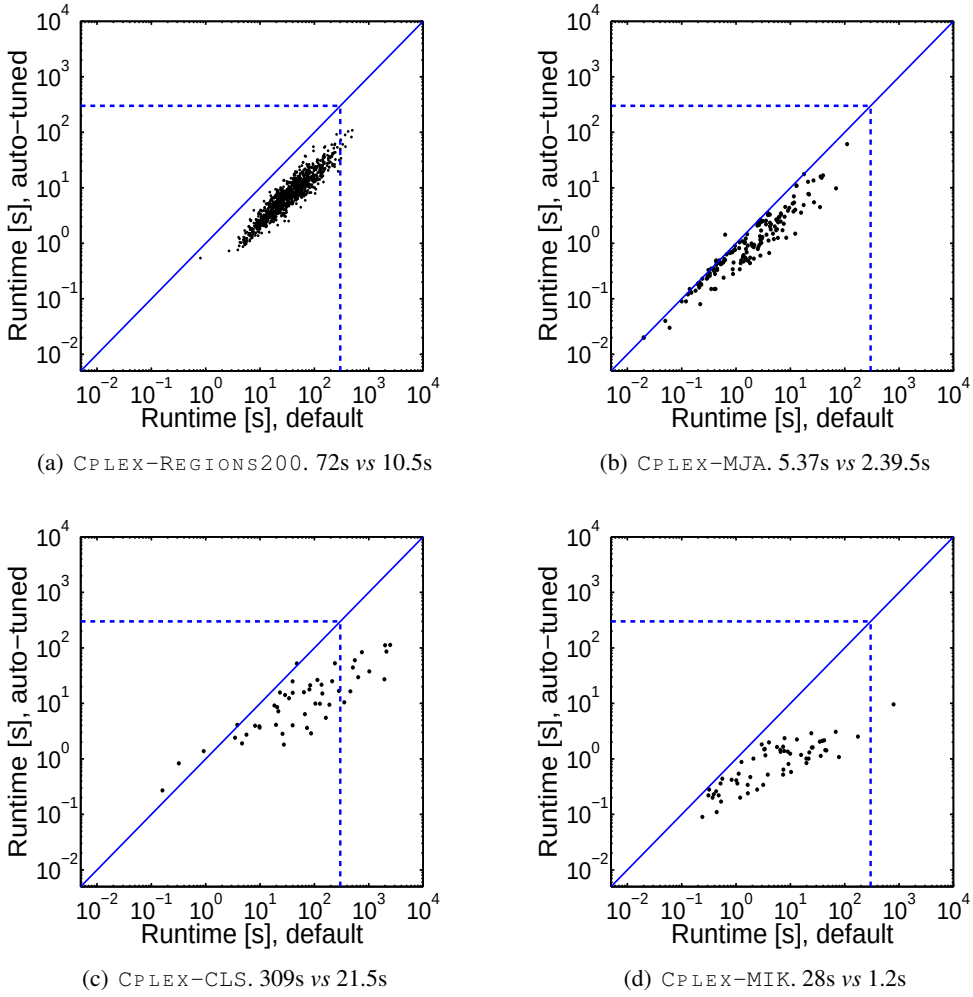
For our experimental evaluation, we used a broad range of MIP benchmarks described in Section 3.3.2. We summarize the CPLEX configuration scenarios using these benchmarks in Table 8.1. Note that the quadratic programs from RNA energy parameter optimization (Andronescu et al., 2007) could be solved in polynomial time.

As our optimization objective, we used penalized average runtime, with a cutoff time of $\kappa_{max} = 300$ seconds and a penalization constant of 10. We used our final versions of BASICILS(100) and FOCUSEDILS, both with aggressive capping ($bm = 2$), with a configuration time of two days (see Section 3.6.3 for a description of the machines used for these experiments). We performed ten runs of each configurator, and measured training and test set performance of each run. As before (see Section 7.4), we report mean and standard deviation of test performance. We also report the test set performance of the parameter configuration found in the repetition with the best *training* performance, and show scatter plots comparing that configuration against the CPLEX default.

8.1.2 Experimental Results

In Table 8.2, Figure 8.1, and Figure 8.2, we summarize our experimental results. The table lists the CPLEX default performance and summarizes results of the ten runs of each configurator. It

experimented with it. The research presented here preceded that development; it dates back to joint work with Holger Hoos, Kevin Leyton-Brown, and Thomas Stützle in summer 2007, which was presented and published at a doctoral symposium (Hutter, 2007). At that time no other mechanism for automatically configuring CPLEX was available.



(a) CPLEX−REGIONS200. 72s *vs* 10.5s

(b) CPLEX−MJA. 5.37s *vs* 2.39.5s

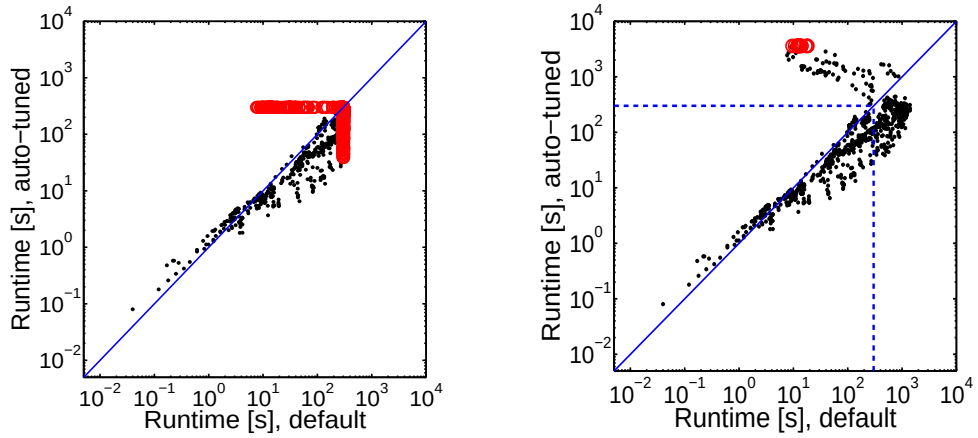
(c) CPLEX−CLS. 309s *vs* 21.5s

(d) CPLEX−MIK. 28s *vs* 1.2s

Figure 8.1: Comparison of default *vs* automatically-determined parameter configuration for the four CPLEX configuration scenarios with $\mathcal{NP}$-hard instances. Each dot represents one test instance. The blue dashed line at 300 CPU seconds indicates the cutoff time of the target algorithm used during the configuration process. The subfigure captions give mean runtimes for the test instances (default *vs* optimized); in none of these cases there were any timeouts.

also gives the performance of the configuration found in the run with best training performance, both for BASICILS and FOCUSEDILS. The two figures compare the performance of that latter configuration FOCUSEDILS found against the CPLEX default.

Figure 8.1 clearly shows the large speedups we achieved for the four sets of instances of $\mathcal{NP}$-hard MIP problems. As before, for that offline evaluation, we allowed larger runtimes of up to one hour. All runs completed successfully in that time. The speedup factors achieved by automated configuration were 6.9, 2.2, 14.4, and 23.3 for scenarios CPLEX−REGIONS200,



(a) CPLEX-QP, with test cutoff of 300 seconds. 81s *vs* 44s; 305 *vs*150 timeouts out of 1 000 instances

(b) CPLEX-QP. 296s *vs* 234s; 0 *vs* 21 timeouts out of 1 000 instances

Figure 8.2: Comparison of default *vs* automatically-determined parameter configuration for CPLEX-QP, for two different test cutoff times. Subfigure (a) shows cutoff time $\kappa_{max} = 300$ seconds, which we chose for the automated configuration; subfigure (b) shows a higher cutoff time of 3 000 seconds used in an offline test. The subfigure captions give mean runtimes for the instances solved by each of the configurations (default *vs* optimized), as well as the number of timeouts for each.

CPLEX-MJA, CPLEX-CLS, and CPLEX-MIK, respectively. For CPLEX-CLS, and CPLEX-MIK, there was also a clear trend towards more pronounced improvements for harder instances. (Recall that this was also the case for our experiments in Section 5.4, Chapter 6, and Section 7.4.)

Figure 8.2 reports our results for scenario CPLEX-QP. Recall that we used a cutoff time of $\kappa_{max} = 300$ seconds in all CPLEX configuration scenarios discussed here. As the figure shows, with that cutoff time, the optimized parameter configuration achieved a roughly two-fold improvement over the CPLEX default on the test instances, both in terms of number of timeouts and in terms of average time spent for solved instances (see Figure 8.2(a)). However, this improvement did not carry over to the higher cutoff of one hour used in an offline test (see Figure 8.2(b)): within that one hour, the CPLEX default solved all test instances, while the configuration optimized for $\kappa_{max} = 300$ second timed out on 21 instances. Thus, the parameter configuration found by FOCUSEDILS *did* generalize well to previously unseen test data, but *not* to larger cutoff times. This is a "real-life" manifestation of the over-tuning problem with low captimes we discussed in our empirical analysis in Section 4.4.2.

We now compare the performance of BASICILS(100) and FOCUSEDILS in some more detail. Table 8.2 shows that, similar to the situation described in Section 7.4, in some configuration scenarios (*e.g.*, CPLEX-CLS, CPLEX-MIK) there was substantial variance between the different runs of the configurators, and the run with the best training performance indeed yielded a parameter configuration with good test performance. Furthermore, FOCUSEDILS tended to show higher performance variation than BASICILS. Even in configuration scenarios

Scenario	Default	Test performance (mean runtime over test instances, in CPU seconds) mean ± stddev. for 10 runs		Run with best training performance	
		BASICILS	FOCUSEDILS	BASICILS	FOCUSEDILS
CPLEX-REGIONS200	72	45 ± 24	$\mathbf{11.4 \pm 0.9}$	15	**10.5**
CPLEX-MJA	5.37	$\mathbf{2.27 \pm 0.11}$	2.4 ± 0.29	**2.14**	2.35
CPLEX-CLS	712	443 ± 294	$\mathbf{327 \pm 860}$	80	**23.4**
CPLEX-MIK	64.8	$\mathbf{20 \pm 27}$	301 ± 948	1.72	**1.19**
CPLEX-QP	969	$\mathbf{755 \pm 214}$	827 ± 306	528	**525**

Table 8.2: Experimental results for our CPLEX configuration scenarios. We performed 25 runs of BASICILS and FOCUSEDILS (both with Aggr Capping and $bm = 2$) and computed their training performance $p_{train,t}$ (PAR across used training instances, using the methods' final incumbents $\boldsymbol{\theta}_{inc}(t)$) and test performance $p_{test,t}$ (PAR across 1 000 test instances, using $\boldsymbol{\theta}_{inc}(t)$) for a configuration time of $t = 2$ days. We list test performance of the CPLEX default, mean ± stddev of $p_{test,t}$ across the 25 runs of the, and $p_{test,t}$ of the configurators' runs with lowest *training* performance $p_{train,t}$. Boldface indicates the better of BASICILS and FOCUSEDILS.

where BASICILS performed better on average, this high variance sometimes helped FOCUSED-ILS to achieve better performance than BASICILS when only using its run with best training performance. (Examples of this effect are the scenarios CPLEX-MIK and CPLEX-QP.) While BASICILS outperformed FOCUSEDILS in three of these five scenarios in terms of mean test performance across the ten runs, FOCUSEDILS achieved the better test performance in the run with the best training performance for all but one scenario, in which it performed almost as well. For scenarios CPLEX-REGIONS200 and CPLEX-CLS, FOCUSEDILS performed substantially better than BASICILS.

One run for configuration scenario CPLEX-MIK demonstrated an interesting failure mode of FOCUSEDILS. In this scenario 9 out of 10 FOCUSEDILS runs yielded parameter configurations with average runtimes below two seconds. One run, however, was very unfortunate. There, $\boldsymbol{\theta}_{inc}$, one of the first visited configurations solved the first instance, π_1, in 0.84 seconds, and no other configuration managed to solve π_1 in less than $2 \cdot 0.84 = 1.68$ seconds. Thus, every configuration $\boldsymbol{\theta} \neq \boldsymbol{\theta}_{inc}$ timed out on π_1 due to the aggressive capping strategy with $bm = 2$ used in these experiments. FOCUSEDILS then iterated the following steps: perturbation to a new configuration $\boldsymbol{\theta}$; comparison of $\boldsymbol{\theta}$ against a neighbour $\boldsymbol{\theta}'$ using a single run each on π_1, both of which timed out after 1.68 seconds, breaking the tie in favour of $\boldsymbol{\theta}'$ since $N(\boldsymbol{\theta}) = N(\boldsymbol{\theta}') = 1$; two bonus runs for $N(\boldsymbol{\theta}')$; comparison of $\boldsymbol{\theta}'$ against all its neighbours $\boldsymbol{\theta}''$ using a single run on π_1, and breaking ties in favour of $\boldsymbol{\theta}'$ since $N(\boldsymbol{\theta}') > N(\boldsymbol{\theta}'')$; 202 bonus runs for $\boldsymbol{\theta}'$. In the seven iterations performed within two CPU days, this process did not find a configuration better than $\boldsymbol{\theta}_{inc}$, which, when evaluated on the test set did not manage to solve a single instance. Since the runtime of unsuccessful runs was counted as ten times the cutoff time, this resulted in an average runtime of $10 \cdot 300 = 3\,000$ seconds for this unfortunate run. This example demonstrates the risk of capping too aggressively, and underlines the importance of using the result of multiple FOCUSEDILS runs with different orderings of the training instances.[4] As discussed above, the best of 10 FOCUSEDILS runs found a configuration with

[4]Another option to avoid this failure would be to not use a fixed order of instances in FOCUSEDILS. In more recent work, we have experimented with a different mechanism in the context of model-based algorithm

average runtime 1.2 seconds, which outperformed the CPLEX default by a factor of more than 20.

To summarize our CPLEX experiments, in all scenarios studied here PARAMILS managed to find parameter configurations that substantially improved upon the CPLEX default in terms of the objective being optimized. For benchmark set QP, a timeout of $\kappa_{max} = 300$ seconds turned out to be too low, leading to over-tuning. In the four other scenarios, performance improvements did scale to larger runtimes and we achieved substantial speedups, between a factor of 2 and 23.

8.2 Self-Configuration of PARAMILS

As discussed in the introduction, algorithm configuration is itself a hard combinatorial problem. Not surprisingly, therefore, most configuration procedures use heuristic mechanisms and are typically controlled by a number of parameters. In the case of PARAMILS, these parameters are as follows:

- r, the number of random configurations to be sampled at the beginning of search;
- s, the perturbation strength;
- $p_{restart}$, the probability of random restarts; and
- bm, the bound multiplier used in our aggressive capping mechanism.

In all experiments reported in previous sections, we have used the manually-determined default values $\langle r, s, p_{restart}, bm\rangle = \langle 10, 3, 0.01, 2\rangle$. These parameters were determined based on preliminary experiments during the early design of PARAMILS. Although the time spent for this preliminary manual parameter optimization is hard to quantify (since it was interleaved with algorithm development), we estimate the overhead caused by this manual experimentation to be about one week of developer time. Since PARAMILS is a general method for automated algorithm configuration, it is natural to wonder whether we could find settings of comparable quality by using PARAMILS to configure itself.

Figure 8.3 illustrates this process of self-configuration. We use PARAMILS with its default parameters as a *meta-configurator* in order to configure the *target configurator* PARAMILS, which in turn is run and evaluated on instances of the algorithm configuration problem. These instances correspond to configuration scenarios, each of which consists of a parameterized target *base algorithm* and a set of input data. In other words, configuration scenarios, such as SPEAR-QCP, play the same role in configuring the target configurator (here: PARAMILS), as SAT instances do in configuring a SAT algorithm, such as SPEAR. The objective to be optimized is performance across a number of configuration scenarios (here we chose geometric mean across 20 runs on each of 5 configuration scenarios).

For the self-configuration experiment described in the following, we chose FOCUSEDILS with aggressive capping as the target configurator and used the sets of parameter values shown

configuration procedures (see Section 13.6.1). We plan to implement this or a similar mechanism in PARAMILS in the near future.

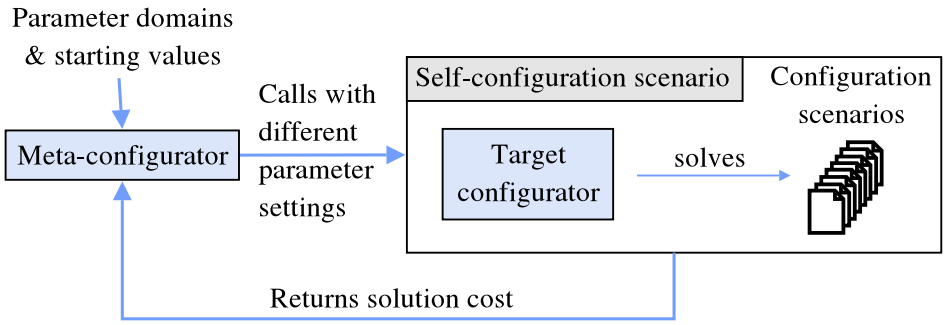


Figure 8.3: Visualization of self-configuration. The target algorithm is now PARAMILS itself, and its "benchmark instances" are configuration scenarios. The meta-configurator PARAMILS (with default parameters) searches for good parameter settings for the target configurator, PARAMILS, evaluating each parameter setting by running PARAMILS on the appropriate configuration scenarios, with geometric mean quality of the results serving as the performance measure. The same binary was used for the meta-configurator and the target configurator.

in Table 8.3. During the development of PARAMILS, we had already considered making the perturbation strength s dependent on the number of parameters to be configured, but ended up not implementing this idea, because we did not want to introduce too many parameters. However, equipped with an automated configuration tool, we now introduced a Boolean parameter that would switch between two conditional parameters: fixed absolute perturbation strength s_{abs} and relative perturbation strength s_{rel} (a factor to be multiplied by the number of parameters to be configured), which yields different perturbation strengths in different configuration scenarios. In total, we considered $4 + 4 = 8$ settings for perturbation strength, 2 different restart probabilities, 3 possible numbers of random steps in the beginning, and 5 options for the bound multiplier, leading to $8 \cdot 2 \cdot 3 \cdot 5 = 240$ possible parameter configurations.

In order to run the self-configuration experiment within a reasonable amount of time on our cluster, parallelization turned out to be crucial. Because BASICILS is easier to parallelize than FOCUSEDILS, we chose BASICILS(100) as the meta-configurator. Furthermore, to avoid potentially problematic feedback between the meta-configurator and the target configurator, we treated them as two distinct procedures with separate parameters; therefore, changes to the parameters of the target configurator, FOCUSEDILS, had no impact on the parameters of the meta-configurator, BASICILS(100), which we kept fixed to the previously-discussed default values.

BASICILS(100) evaluated each parameter configuration $\boldsymbol{\theta}$ of the target configurator, FOCUSEDILS, by performing 20 runs of FOCUSEDILS($\boldsymbol{\theta}$) on each of our five BROAD configuration scenarios. Unlike in previous experiments, we limited the total running time of FOCUSEDILS to one CPU hour per configuration scenario. This reduction was necessary to keep the overall computational burden of our experiments within reasonable limits.[5] We

[5]We note, however, that, similar to the effect observed for scenario CPLEX-QP in the previous section, there is a risk of finding a parameter configuration that only works well for runs up to one hour and works poorly thereafter.

Parameter	Note	Values considered
r	Number of initial random configurations	0, ***10***, 100
bm	Bound multiplier for adaptive capping	1.1, 1.3, 1.5, *2*, **4**
$p_{restart}$	Probability of random restart	*0.01*, **0.05**
pert	Type of perturbation	*absolute*, **relative**
s_{abs}	Abs. perturbation strength; only active when *pert*=absolute	1,*3*,5,10
s_{rel}	Rel. perturbation strength; only active when *pert*=relative	0.1,0.2,**0.4**,0.8

Table 8.3: Parameters of PARAMILS, the possible values considered for the self-configuration, and the values determined for FOCUSEDILS in the self-configuration experiment. Values in italic font are the default values, bold faced values are chosen by the meta-configurator. (Default: $\langle r, bm, p_{restart}, \textit{pert}, s_{\text{abs}} \rangle = \langle 10, 2, 0.01, \text{absolute}, 3 \rangle$; Self-configured: $\langle r, bm, p_{restart}, \textit{pert}, s_{\text{rel}} \rangle = \langle 10, 4, 0.05, \text{relative}, 0.4 \rangle$) In the case of relative perturbation strength, we computed the actual perturbation strength as $\max(2, s_{\text{rel}} \cdot M)$, where M is the number of parameters to be set in a configuration scenario.

then evaluated the incumbent parameter configuration of the base algorithm found in each of these one-hour runs of the target configurator on a validation set (consisting of the $1\,000$ training instances but using different random seeds), using $1\,000$ runs and a cutoff time of five CPU seconds. Thus, including validation, each run of the target configurator required up to approximately 2.5 CPU hours (one hour for the configuration process[6] plus up to $5\,000$ CPU seconds for validation).

The meta-configurator used the geometric mean of the 100 validation results obtained from the 20 runs for each of the 5 configuration scenarios to assess the performance of configuration $\boldsymbol{\theta}$. (We used geometric instead of arithmetic means because the latter can easily be dominated by the results from a single configuration scenario.) We note that, while this performance measure is ultimately based on algorithm runtimes (namely those of the base algorithms configured by the target configurator), unlike the objective used in previous sections, it does not correspond to the runtime of the target configurator itself. Rather, it corresponds to the solution quality the target configurator achieves within a bounded runtime of one hour.

The 100 runs of the target configurator performed in order to evaluate each parameter configuration were run concurrently on 100 CPUs. We ran the meta-configurator BASICILS(100) for a total of five iterations in which 81 parameter configurations were evaluated. This process took a total of about 500 CPU days (where some of the 100 CPUs used in the experiment were occasionally idle after validation tasks requiring less than $5\,000$ CPU seconds). The best configuration of the target configurator, FOCUSEDILS, found in this experiment is shown in Table 8.3; it had a geometric mean runtime of 1.40 CPU seconds across the 100 validation runs (as compared to a geometric mean runtime of 1.45 achieved

[6] PARAMILS's CPU time is typically dominated by the time spent running the target algorithm. However, in configuration scenarios where each execution of the base algorithm is very fast this was not always the case, because of overhead due to bookkeeping and calling algorithms on the command line. In order to achieve a wall clock time close to one hour for each PARAMILS run in the self-configuration experiments, we minimized the amount of bookkeeping in our simple Ruby implementation of PARAMILS and counted every algorithm execution as taking at least one second. This lower bound on algorithm runtimes is a parameter of PARAMILS that was set to 0.1 seconds for all other experimental results.

Scenario	Default PARAMILS	Self-configured PARAMILS	p-value
SAPS-SWGCP	$\mathbf{0.316 \pm 0.054}$	0.321 ± 0.043	(0.50)
SPEAR-SWGCP	8.30 ± 1.06	$\mathbf{8.26 \pm 0.88}$	0.72
SAPS-QCP	5.21 ± 0.39	$\mathbf{5.16 \pm 0.32}$	0.69
SPEAR-QCP	1.29 ± 0.20	$\mathbf{1.22 \pm 0.18}$	0.28
CPLEX-REGIONS100	0.35 ± 0.05	$\mathbf{0.34 \pm 0.02}$	0.34

Table 8.4: Effect of self-configuration. We compare test set performance (mean runtime of best configuration found, in CPU seconds) for FOCUSEDILS with its default parameter settings and with the parameter settings found via self-configuration. For each configuration scenario, we report mean $\pm$ stddev of the test performance over 25 repetitions of the configurators, and the p-value for a paired Max-Wilcoxon test (see Section 3.6.2).

using FOCUSEDILS's default settings) and was found after a total runtime of 71 CPU days; it was the 17th of the 81 configurations evaluated by BASICILS(100) and corresponded to the first local minimum reached. The next three iterations of BASICILS all led to different and slightly worse configurations. In the fifth iteration, the same best configuration was found again.

Table 8.4 reports the performance achieved by FOCUSEDILS with this new, automatically determined parameter setting, on the original BROAD configuration scenarios. We note that the performance measure used here differs from the objective optimized by the meta-configurator. Nevertheless, the self-configuration process resulted in competitive performance. There was no statistically significant difference between the performance of the resulting configuration and the manually-engineered PARAMILS default, but the automatically-determined configuration performed better on average for four of the five configuration scenarios.

Even though self-configuration led to slightly improved results, we believe that this experiment did not exploit its full potential. The objective function used for training (geometric mean performance across $5 \cdot 20$ configuration runs) was very different than the one measured at test time (comparison to default for each scenario separately). As we showed in Chapter 4, configuration scenarios are also very heterogeneous: the optimal approach to use differs across scenarios. Furthermore, the cutoff times during training were most likely too short to properly reflect the performance of the configurator in longer runs. Finally, PARAMILS seems to be quite robust with respect to its parameter settings. We expect that configuration procedures that rely more on good settings of their parameters will benefit more from self-configuration. For example, note that our empirical analysis in Section 4.4.2 showed that the best combination of N and κ_{max} strongly depends on the configuration scenario and on the budget available for configuration. Thus, if we were to configure BASICILS(N) on single configuration scenarios, making N and κ_{max} configurable parameters, we would expect self-configuration to improve upon the configurator's default.

8.3 Applications of PARAMILS by Other Researchers

Here, we review applications by researchers other than the author of this thesis. This demonstrates the generality and the maturity of the approach. Note that the work reviewed here is *not* a contribution of this thesis. The configuration of SATENSTEIN is described in more detail than the other applications since it demonstrates an important usage scenario of PARAMILS.

8.3.1 Configuration of SATenstein

In the introduction, we prominently discussed the possibility of using automated algorithm configuration procedures to automatically design algorithms from components. Here, we ground that discussion by reviewing an application of PARAMILS to that purpose.

KhudaBukhsh et al. (2009) used PARAMILS to perform automatic algorithm design in the context of stochastic local search algorithms for SAT. Specifically, they introduced a new framework for local search SAT solvers called SATenstein, and used PARAMILS to choose good instantiations of the framework for given instance distributions. SATenstein spans three broad categories of SLS-based SAT solvers: WalkSAT-based algorithms, dynamic local search algorithms and G^2WSAT variants. All of these are combined in a highly parameterized framework solver with a total of 41 parameters and $4.82 \cdot 10^{12}$ distinct instantiations; see Section 3.2.1 for more details.

FOCUSEDILS was used to configure SATenstein on six different problem distributions, and the resulting solvers were compared to eleven state-of-the-art SLS-based SAT solvers. The results, summarized in Table 8.5, show that the automatically configured versions of SATenstein outperformed all of the eleven state-of-the-art solvers in all six categories, sometimes by a large margin. (On R3SAT, PAWS performed better in terms of median runtime, but SATenstein-LS[R3SAT] solved hard instances more effectively and, consequently, achieved better mean runtime.)

These results clearly demonstrate that automated algorithm configuration methods can go beyond simple "parameter tuning" and can be used to construct *new* algorithms by combining a wide range of components from existing algorithms in novel ways. Due to the low degree of manual work required by this approach, we believe that such *automated design of algorithms from components* will become a mainstream technique in the development of algorithms for hard combinatorial problems.

Key to the successful application of FOCUSEDILS for configuring SATENSTEIN was the careful selection of homogeneous instance distributions, most instances of which could be solved within a comparably low cutoff time of 10 seconds per run. As in Section 7.4, for each configuration scenario multiple independent runs of FOCUSEDILS were performed in parallel (here 10), and the configuration with the best training quality was selected.

8.3.2 Configuration of a Monte Carlo Algorithm for Protein Folding

Thachuk et al. (2007) used BASICILS in order to determine performance-optimizing parameter settings of a new replica exchange Monte Carlo algorithm for protein folding in the 2D-HP and

Distribution	SATenstein-LS[D]	GNOV	AG20	AG2+	RANOV	G2	VW	ANOV	AG2p	SAPS	RSAPS	PAWS
	0.13	422.33	1051	1080.29	76.22	2952.56	1025.9	28.3	1104.42	1256.2	1265.37	1144.2
QCP	**0.01**	0.03	0.03	0.03	0.1	361.21	0.25	**0.01**	0.02	0.03	0.04	0.02
	100%	92.7%	80.5%	80.3%	98.7%	50.6%	82.2%	99.6%	79.4%	79.2%	78.4%	80.8%
	0.03	0.24	0.62	0.45	0.15	4103.27	159.67	0.06	0.45	3872.08	5646.39	4568.59
SW-GCP	**0.03**	0.09	0.12	0.08	0.12	N/A	40.96	**0.03**	0.07	N/A	N/A	N/A
	100%	**100%**	**100%**	**100%**	**100%**	30.5%	98.9%	**100%**	**100%**	33.2%	5%	22.1%
	1.51	10.93	2.37	3.3	14.14	5.32	9.53	12.4	2.38	22.81	14.81	2.4
R3SAT	0.14	0.14	0.14	0.16	0.32	0.13	0.75	0.21	0.13	1.80	2.13	**0.12**
	100%	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
	0.03	52.87	139.33	138.84	156.96	110.02	177.9	147.53	107.4	48.31	38.51	73.27
HGEN	**0.02**	0.73	0.57	0.61	0.95	0.61	3.23	0.76	0.49	3.00	2.44	0.96
	100%	99.4%	98%	97.8%	97.7%	98.4%	97.5%	97.6%	98.4%	99.5%	99.7%	99.2%
	12.22	5912.99	3607.4	1456.4	943.26	5944.6	3411.93	3258.66	1989.91	17.39	19.39	26.51
FAC	**8.03**	N/A	N/A	237.50	155.77	N/A	N/A	N/A	315.48	11.60	12.88	12.62
	100%	0%	30.2%	84.2%	92.1%	0%	31.7%	37.2%	72.5%	**100%**	**100%**	99.9%
	5.59	2238.7	2170.67	2161.59	1231.01	2150.31	385.73	2081.94	2282.37	613.15	794.93	1717.79
CBME(SE)	**0.02**	0.75	0.67	0.77	0.66	0.68	0.27	5.81	3.18	0.04	0.03	19.99
	100%	61.13%	61.13%	61.13%	79.73%	64.12%	92.69%	61.79%	61.13%	90.03%	85.38%	68.77%

Table 8.5: Performance summary of `SATenstein-LS` and its 11 challengers, reproduced from the paper by KhudaBukhsh et al. (2009), with permission from the authors. Every algorithm was run 10 times on each instance with a cutoff of 600 CPU seconds per run. Each row summarizes the performance of `SATenstein-LS[D]` and 11 challengers on the test set for a particular instance distribution D as $a/b/c$, where a (top) is the penalized average runtime (counting timeouts after 600 seconds as $6\,000$ seconds); b (middle) is the median of the median runtimes over all instances (not defined if fewer than half of the median runs failed to terminate); c (bottom) is the percentage of instances solved (i.e., for which median runtime $<$ cutoff).

3D-HP models.[7] Even though their algorithm has only four parameters (two categorical and two continuous, discretized to a total of $9\,000$ configurations), BASICILS achieved substantial performance improvements. While the manually-selected configurations were biased in favour of either short or long protein sequences, BASICILS found a configuration which consistently yielded good mean runtimes for all types of sequences. On average, the speedup factor achieved was approximately 1.5, and for certain classes of protein sequences up to 3. For each manually-selected configuration there were instances for which it performed worse than the previous state-of-the-art algorithm. In contrast, the robust parameter configurations selected by BASICILS yielded uniformly better performance than that algorithm.

8.3.3 Configuration of a Local Search Algorithm for Time-Tabling

In very recent work, Fawcett et al. (2009) used several variants of PARAMILS (including a version that has been slightly extended beyond those presented here) to design a modular stochastic local search algorithm for the post-enrollment course timetabling problem. They followed a design approach that used automated algorithm configuration in order to explore a large design space of modular and highly parameterized stochastic local search algorithms. This quickly led to a solver that placed third in Track 2 of the second International Timetabling Competition (ITC2007) and subsequently produced an improved solver that achieves consistently better performance than the top-ranked solver from the competition.

[7] BASICILS was used, because FOCUSEDILS had not yet been developed when that study was conducted.

8.4 Chapter Summary

In this chapter, we presented a number of successful applications of PARAMILS to the configuration of algorithms for various hard combinatorial problems: SAT, MIP, protein folding, scheduling, and algorithm configuration itself. In self-configuration, PARAMILS found a parameter configuration marginally better than the manually-engineered PARAMILS default. In all other applications, algorithm configuration led to substantial performance improvements, often of several orders of magnitude, with a trend towards larger improvements for harder instances. This confirms the trend in scaling behaviour we already observed in the formal verification domain in Chapter 6.

Most noteworthy are our results for optimizing the most widely-used commercial optimization tool ILOG CPLEX. Even though ILOG has expended substantial efforts to establish a well-performing CPLEX default configuration, we achieved between 2-fold and 23-fold speedups in various configuration scenarios. This result has the potential to substantially facilitate the use of CPLEX in both industry and academia.

PARAMILS was also used in three applications by researchers other than the author of this thesis; this serves as an independent validation of the approach. Together, the applications in this chapter underline the generality and the maturity of algorithm configuration in general, and PARAMILS in particular.

Part IV

Model-based Search for Algorithm Configuration

—in which we improve sequential model-based optimization approaches and substantially extend their scope to include general algorithm configuration

Chapter 9

Sequential Model-Based Optimization: The State of The Art

The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work.
—John von Neumann, Hungarian American mathematician

At this point of the thesis, we have hopefully convinced the reader that automated algorithm configuration can lead to substantial reductions in the development time of complex algorithms, and that it has the potential to improve algorithm performance by orders of magnitude. We now switch focus and consider an alternative, rather different, framework for algorithm configuration, which will turn out to yield the most effective configuration procedures for some types of configuration scenarios. This framework is based on so-called *response surface models*, regression models that aim to predict the cost of parameter configurations. In the context of multiple instances, these models can also be used to predict the cost distribution for a combination of a parameter configuration and an instance.

Response surface models can be useful in a variety of contexts. They can be used to interpolate performance between parameter configurations, and to extrapolate to previously-unseen regions of the configuration space. They can also be used to quantify the importance of each parameter, as well as interactions between parameters, and—in the context of multiple instances—between parameters and instance characteristics. Thereby, they can provide intuition about the parameter response that cannot be gathered from model-free methods. This can be a substantial help for algorithm designers seeking to understand and improve aspects of their algorithm.

Here, we use response surface models in order to guide the search for new and promising parameter configurations. The resulting model-based framework thus provides another possible answer to the first dimension of algorithm configuration: which sequential search strategy

should be used to select the parameter configurations $\vec{\boldsymbol{\Theta}}$ to be evaluated?

In this part of the thesis, we make the connection between algorithm configuration and (stochastic) blackbox optimization (see Section 1.2.1) more explicit than before, starting out from methods for the latter problem. While a multitude of model-free approaches exist for blackbox optimization, in this part of the thesis we focus on model-based procedures. We first study existing state-of-the-art sequential model-based optimization procedures (developed for blackbox optimization), improve them in various ways, and extend them piece by piece to handle increasingly-general algorithm configuration problems. In particular, we focus on the following problems.

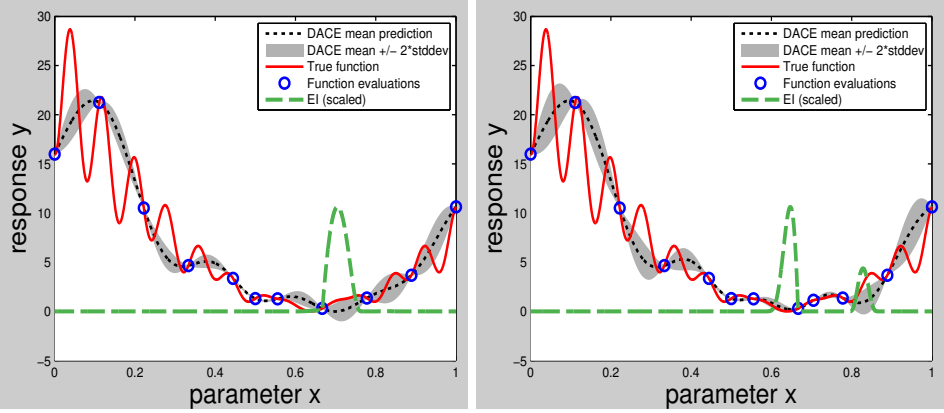
- Optimization of continuous parameters of a randomized algorithm on a single problem instance, given a budget on the number of target algorithm runs (this chapter and Chapter 10). This can be formulated as a canonical stochastic blackbox optimization problem.
- Optimization of continuous parameters of a randomized algorithm on a single problem instance, given a *time* budget (Section 10.4 and Chapter 11). The presence of such a time budget amounts to a change of the termination criterion in the associated stochastic blackbox optimization problem.
- Optimization of *categorical* parameters of a randomized algorithm on a single problem instance (Chapter 12). This can be formulated as a *discrete* stochastic blackbox optimization problem.
- General configuration of algorithms with categorical parameters *across multiple instances* (Chapter 13). As discussed in Section 1.2.2, this can, in principle, be formulated as a discrete stochastic blackbox optimization problem. However, this formulation sacrifices the possibility of selecting which instances (and seeds) to use for each run of the target algorithm.

In this chapter[1], we review the concepts behind sequential model-based optimization (SMBO), discuss two prominent existing SMBO approaches and empirically compare their performance. Since the procedures we study originate in the statistical literature on sequential experimental design, we borrow some terminology from that field to describe them. In particular, to be consistent with that literature, we refer to parameter configurations $\boldsymbol{\theta}_i$ as *design points*, and to the outcome (cost) of a single target algorithm run, o_i, as the *response value* at design point $\boldsymbol{\theta}_i$.

9.1 A Gentle Introduction to Sequential Model-Based Optimization (SMBO)

Model-based optimization methods fit a so-called response surface model and use this model for optimization. This regression model is fitted using a set of training data points $\{(\boldsymbol{\theta}_i, o_i)\}_{i=1}^{n}$, where $\boldsymbol{\theta}_i = (\boldsymbol{\theta}_{i1}, \ldots, \boldsymbol{\theta}_{id})^{\intercal}$ is a design point (a parameter configuration) and o_i is the response value at that design point (the cost of the target algorithm run with configuration $\boldsymbol{\theta}_i$). In

[1] This chapter is based on published joint work with Holger Hoos, Kevin Leyton-Brown, Kevin Murphy, and Thomas Bartz-Beielstein (Hutter et al., 2009e,a).



(a) SMBO, step 1 (b) SMBO, step 2

Figure 9.1: Two steps of SMBO for the optimization of a 1-d function. The true function is shown in red, the blue circles denote our observations. The dotted black line denotes the mean prediction of the DACE model, with the grey area denoting its uncertainty. Expected improvement (scaled for visualization) is shown as a dashed green line. See the text for details.

sequential model-based optimization, the selection of new design points (new parameter configurations to be evaluated) can depend on the response values observed at previous design points.

As a simple example, consider the global minimization of a one-dimensional noise-free function. This simple setting—used here for illustrative purposes only—is demonstrated in Figure 9.1(a). The red line in that figure depicts the true function to be optimized; we use the Levy function[2] with input range $[-15, 10]$ normalized to $[0, 1]$. Note that this function has multiple local minima. Thus, local optimization approaches, such as quasi-Newton methods, are unlikely to work well.

A typical sequential model-based optimization approach would start by sampling the function at a number of inputs, usually defined by a Latin hypercube design (LHD) (Santner et al., 2003) in the region of interest (the Cartesian product of the intervals considered for each parameter; in algorithm configuration, this corresponds to the configuration space, $\mathbf{\Theta}$). In our one-dimensional example in Figure 9.1, blue circles denote samples we took of the function: 10 samples equally spread out across the region of interest (an LHD in one dimension), plus one point in the middle.[3]

In model-based optimization, we fit a response surface model to these samples in order

[2]This function is defined as $f(x) = s + (z-1)^2 \cdot (1 + (\sin(2 \cdot \pi \cdot z))^2)$, where $z = 1 + (x-1)/4$ and $s = \sin(\pi \cdot z)^2$. (See Hedar, 2009)

[3]That additional point in the middle is due to our implementation of configuration procedures, which we used to generate these plots. This implementation always start by evaluating the default, which, here, we chose as the middle of the parameter interval.

to approximate the function. In most recent work on sequential model-based optimization, this model takes the form of a Gaussian stochastic process (GP) (Rasmussen and Williams, 2006). In particular, assuming deterministic responses, Jones et al. (1998) and Santner et al. (2003) use a popular noise-free GP model, which has come to be known as the *DACE* model (an acronym for "Design and Analysis of Computer Experiments", the title of the paper by Sacks et al. (1989) that popularized the model). Our example in Figure 9.1(a) also shows a fit by a DACE model. Note that such a model provides not only a point estimate of the function, but for any given query point, it provides a predictive distribution. In GP models this predictive distribution takes the form of a Gaussian; in fact, for N query points, the predictive distribution is a N-variate Gaussian. In Figure 9.1(a), we plot the mean $\pm$ two standard deviations of the predictive distribution. Note that this does not perfectly fit the true function everywhere—especially in the regions where the function oscillates more strongly—but that it captures the overall trend fairly well.

Now that we have an approximation of the true function, we use it to sequentially select the next design point $\boldsymbol{\theta}$ to query. (In the context of algorithm configuration, we have a predictive model of the cost of parameter configurations, and, based on that, we select the next configuration to run the target algorithm with.) In this decision, we have to trade off learning about new, unknown parts of the parameter space and intensifying the search locally in the best known region (a so-called exploration/exploitation tradeoff). The most popular criterion for selecting the next design point $\boldsymbol{\theta}$ is the expectation of positive improvement over the incumbent solution $\boldsymbol{\theta}_{inc}$ at $\boldsymbol{\theta}$ (where the expectation is taken with respect to the predictive distribution that the current model attributes to $\boldsymbol{\theta}$). This *expected improvement* criterion (EIC) goes back to the work of Mockus et al. (1978) and continues to be the most widely used criterion today. In our example in Figure 9.1(a), the dashed green line denotes this EIC (normalized to an interval $[0, max]$ for visualization purposes), evaluated throughout the region of interest. Note that EIC is high in regions of low predictive mean and high predictive variance; these regions are most likely to contain configurations $\boldsymbol{\theta}$ with cost lower than that of the incumbent, $\boldsymbol{\theta}_{inc}$.

In the next step, we select the design point $\boldsymbol{\theta}_i$ with maximal EIC, query it, and update the model based on the observed response value, o_i (in algorithm configuration, the performance of the single new target algorithm run with configuration $\boldsymbol{\theta}_i$). We demonstrate this in our example in Figure 9.1(b). Note the additional data point at an x-value around 0.7. This data point improves the model, in this case mostly locally.[4] In particular, the uncertainty around the new data point is greatly reduced and the promising region is split in two. In the next step, the left-most of these regions would be explored, since it has higher EIC.

The combination of a noise-free GP model and the EIC described above makes up the *efficient global optimization (EGO)* algorithm by Jones et al. (1998), a popular method for blackbox function optimization. We formalize GP models in the next section. Then, we introduce two independent extensions of EGO that deal with the global optimization of *noisy* functions. (In the context of algorithm configuration, this extension enables the procedures to configure randomized algorithms.)

[4] However, we note that at times a single additional data point can have dramatic effects on the global structure of the model.

9.2 Gaussian Process Regression

To construct a Gaussian process (GP) regression model, first we need to select a parameterized kernel function $k_\lambda : \boldsymbol{\Theta} \times \boldsymbol{\Theta} \rightarrow \mathbb{R}^+$, specifying the similarity between two parameter configurations. We also need to set the variance σ^2 of Gaussian-distributed measurement noise (also known as observation noise; in algorithm configuration, this corresponds to the variance of the target algorithm's runtime distribution). The predictive distribution of a zero-mean Gaussian stochastic process for response o_{n+1} at input $\boldsymbol{\theta}_{n+1}$ given training data $\mathcal{D} = \{(\boldsymbol{\theta}_1, o_1), \ldots, (\boldsymbol{\theta}_n, o_n)\}$, measurement noise of variance σ^2 and kernel function k is then

$$p(o_{n+1}|\boldsymbol{\theta}_{n+1}, \boldsymbol{\theta}_{1:n}, \mathbf{o}_{1:n}) = \mathcal{N}(o_{n+1}|\boldsymbol{k}_*{}^\intercal[\boldsymbol{K} + \sigma^2\boldsymbol{I}]^{-1}\mathbf{o}_{1:n}, k_{**} - \boldsymbol{k}_*{}^\intercal[\boldsymbol{K} + \sigma^2\mathbf{I}]^{-1}),$$

where

$$\begin{aligned} \boldsymbol{K} &= \begin{pmatrix} k(\boldsymbol{\theta}_1, \boldsymbol{\theta}_1) & \ldots & k(\boldsymbol{\theta}_1, \boldsymbol{\theta}_n) \\ & \ddots & \\ k(\boldsymbol{\theta}_n, \boldsymbol{\theta}_1) & \ldots & k(\boldsymbol{\theta}_n, \boldsymbol{\theta}_n) \end{pmatrix} \\ \boldsymbol{k}_* &= (k(\boldsymbol{\theta}_1, \boldsymbol{\theta}_{n+1}), \ldots, k(\boldsymbol{\theta}_n, \boldsymbol{\theta}_{n+1})) \\ k_{**} &= k(\boldsymbol{\theta}_{n+1}, \boldsymbol{\theta}_{n+1}) + \sigma^2, \end{aligned}$$

$\mathbf{I}$ is the n-dimensional identity matrix, and $p(a|b) = \mathcal{N}(a|\mu, \Sigma)$ denotes that the conditional distribution of a given b is a Gaussian with mean μ and covariance matrix Σ. See, *e.g.*, the book by Rasmussen and Williams (2006) for a derivation. A variety of kernel functions are possible, but the most common is a kernel of the form

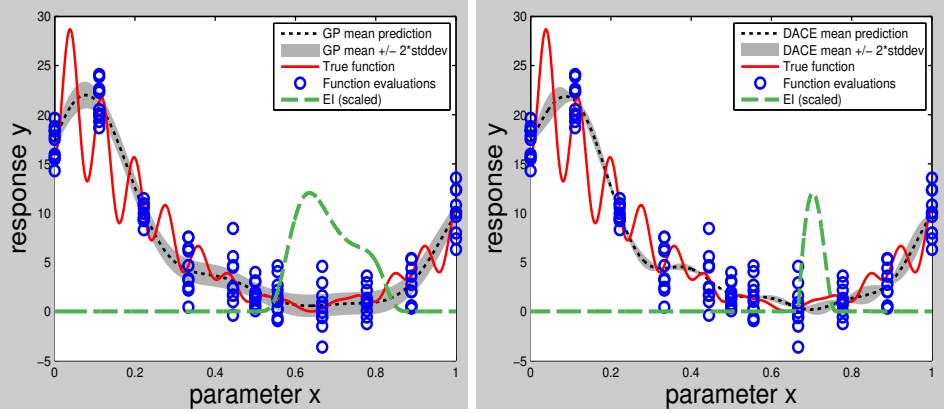
$$K(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) = \exp\left[\sum_{l=1}^{d}(-\lambda_l(\boldsymbol{\theta}_{il} - \boldsymbol{\theta}_{jl})^2)\right],$$

where $\lambda_1, \ldots, \lambda_d$ are the kernel parameters. This kernel is most appropriate if the response is expected to vary smoothly in the input parameters $\boldsymbol{\theta}$. The kernel parameters and the observation noise variance σ^2 constitute the *hyper-parameters* $\boldsymbol{\phi}$, which are typically set by maximizing the *marginal likelihood* $p(\boldsymbol{o_{1:N}})$ with a gradient-based optimizer. Using the chain rule, the gradient is

$$\frac{\partial \log p(\boldsymbol{o_{1:N}})}{\partial \phi_j} = \frac{\partial \log p(\boldsymbol{o_{1:N}})}{\partial(\boldsymbol{K} + \sigma^2\boldsymbol{I})}\frac{\partial(\boldsymbol{K} + \sigma^2\boldsymbol{I})}{\partial \phi_j}.$$

In *noise-free* GP models, such as the popular DACE model used in EGO, the observation noise variance is fixed to $\sigma^2 = 0$.

Learning a GP model from data can be computationally expensive. Inverting the n by n matrix $[\boldsymbol{K} + \sigma^2\mathbf{I}]$ takes time $O(n^3)$, and has to be done in every step of the hyper-parameter optimization. (We discuss approximations later in Section 11.3.2.) We refer to the process of optimizing hyper-parameters and computing the inverse as *fitting* the model. *Training*, *learning*, or *building* the model are used as synonyms of "fitting". Once the fitting has been



(a) Standard GP fit

(b) Noise-free GP fit on cost statistics

Figure 9.2: Alternative ways of fitting a response surface to noisy observations. Here, the additive noise is sampled from $\mathcal{N}(0, 4)$. Note that we plot mean $\pm$ two standard deviations of the *predictive mean*; the predicted observation noise would need to be added to this (but is zero for the DACE model in (b)).

done, subsequent predictions are relatively cheap, only requiring matrix-vector multiplications and thus time $O(n^2)$.

9.3 Two Methods for Handling Noisy Data

In this section, we discuss two different methods for modelling functions with observation noise. In the context of algorithm configuration, such observation noise is due to randomization in the algorithm or to extraneous nuisance variables, such as differences in the runtime environment caused by other processes executed in parallel.

9.3.1 Standard Gaussian Processes

The standard approach to handle observation noise in Gaussian stochastic process (GP) models is to simply assume the noise to be Gaussian-distributed with mean zero and unknown variance σ^2, and to carry on as described in Section 9.2, treating σ^2 as a hyper-parameter to be optimized.

In Figure 9.2(a), we visualize this approach. In that figure, the red line is the same true function as in the noise-free example in Figure 9.1(a). We now have ten noisy observations for each of the design points considered in the previous example, depicted by blue circles. Note that we plot the predictive distribution for the posterior *mean* at each data point. With additional data, the uncertainty in this prediction for the mean will shrink, while the predicted observation noise (not illustrated) will not.

9.3.2 Noise-free Gaussian Processes Trained on Empirical Statistics

Another possibility for handling noisy response values in sequential optimization is to first compute a (user-defined) *empirical cost statistic* $\hat{c}(\boldsymbol{\theta})$ across the observations for each configuration $\boldsymbol{\theta}$, and then to fit a noise-free GP model to these empirical cost statistics. We illustrate this approach in Figure 9.2(b), using the sample mean as the empirical cost statistic. We note that—in contrast to the standard GP—this version yields uncertainty estimates that are exactly zero for the training data points. The two sequential optimization methods we compare in Section 9.5 employ these two approaches for handling noisy data; we discuss the ramifications of these choices in detail in Section 9.5.3.

9.4 Log Transformations of Raw Data and Cost Statistics

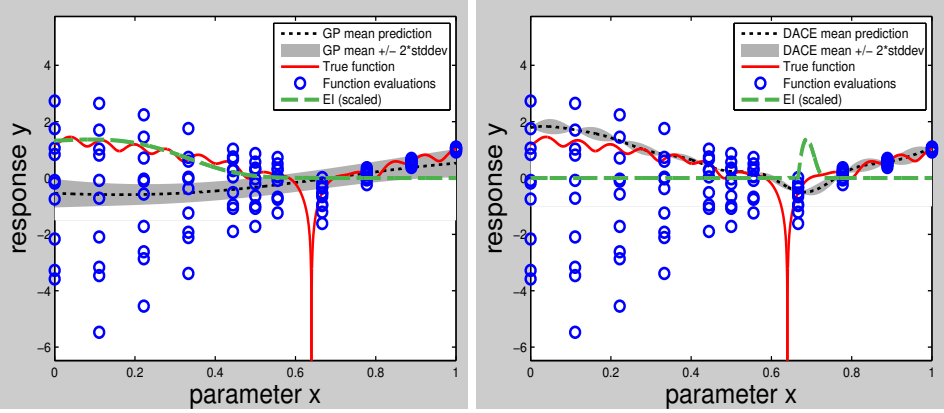
In some cases, the GP model fit can be improved by a transformation of the response data. Transformations that have been suggested in the literature include log transformations (Jones et al., 1998; Williams et al., 2000; Huang et al., 2006; Leyton-Brown et al., 2002), negative inverse transformations (Jones et al., 1998), and square root or cube root transformations (Leyton-Brown, 2003). In all these references, transformations were applied to the raw data (as opposed to transformations of cost statistics, which we apply at the end of this section).

Here, we focus on the log transformation. This is because our main interest is in minimizing *positive* functions with spreads of several orders of magnitude that arise in the optimization of runtimes. In this chapter, we study a number of scenarios where the objective is to minimize positive solution cost; cost metrics that can be negative can be transformed to positive functions by subtracting a lower bound.

One issue that seems to have been overlooked in previous work is that in combination with noisy response values, such transformations in fact change the objective function modelled by the response surface. Take the example of minimizing arithmetic mean runtime in combination with a log transform. The GP simply fits the mean of the response data used to train it; if those response data are log-transformed the GP will fit the *mean of the logs*. Note that this mean of the logs is in fact closely related to the geometric mean:

$$\begin{aligned}\text{geometric mean} &= \sqrt[n]{\prod_{i=1}^{n} x_i} = \left[\exp\left(\sum_{i=1}^{n} \log(x_i)\right)\right]^{(1/n)} \\ &= \exp\left[\frac{1}{n}\sum_{i=1}^{n} \log(x_i)\right] = \exp\left(\text{mean of logs}\right).\end{aligned}$$

The $\exp(\cdot)$ function is monotonic; thus, comparing the mean of the logs of two parameter configurations is equivalent to comparing their geometric means. Consider an example where this causes problems. Let parameter configuration $\boldsymbol{\theta}_1$ yield a deterministic algorithm with (mean) runtime of one second, and let configuration $\boldsymbol{\theta}_2$ yield a randomized algorithm, which terminates in 0.001 seconds in 60% of cases but takes 1000 seconds in the remaining 40%. Clearly, when optimizing *arithmetic mean* runtime, $\boldsymbol{\theta}_2$ is a very poor choice. However, it



(a) Standard GP fit to log-transformed raw data (b) DACE fit to log-transformed cost statistics

Figure 9.3: Alternative GP fits to log-transformed data. The noise in this example is non-stationary and multiplicative. Note that the y-axis now shows log-transformed responses. The standard GP fits the mean of the logs, *i.e.*, the geometric mean. The noise-free DACE model trained on log-transformed cost statistics fits the log of the arithmetic mean.

would be judged preferable to $\boldsymbol{\theta}_1$ by a GP model fitted on log responses:

$$0.6 \cdot \log_{10}(0.001) + 0.4 \cdot \log_{10}(1000) = -0.6 < 0 = \log_{10}(1).$$

We illustrate this problem on the Levy function shown in Figure 9.3. Here, we chose non-stationary multiplicative noise. That is, the noise at parameter value x depends on the function value $f(x)$ and also on the value of x; here, the lower x, the larger the noise.[5] In particular, we drew each response value from a Weibull distribution with mean $f(x)$ and standard deviation $f(x) \cdot \exp(5 \cdot (1 - x))/10$. Figure 9.3(a) demonstrates that the standard GP model fits the mean of the logs, which here largely differs from the log of the mean. This leads both to poor choices for the location of the next design point and to a poor choice of incumbent.

For the second approach, we simply choose the log of the mean as our cost statistic, and then fit a DACE model to that cost statistic. Figure 9.3(b) shows the result of this approach for the same data as used above. The prediction closely follows the (log of the) true function. Note that this approach works for any combination of user-defined cost statistic and transformation. We are not aware of any previous discussion of an instance of this approach.

[5]Note that multiplicative noise is very common in randomized algorithms: the standard deviation is often roughly proportional to the mean of the runtime distribution (in fact, runtime distributions can often be approximated well by exponential distributions (Hoos, 1999a), for which mean and standard deviation are identical). The variance of an algorithm's runtime distribution also often varies with the value of some parameter. For example, up to a point additional randomization tends to improve an algorithm's robustness and thereby decrease the variance of its runtime distribution.

Symbol	Meaning
$\boldsymbol{\Theta}$	Space of allowable parameter configurations (region of interest)
$\boldsymbol{\theta}$	Parameter configuration, element of $\boldsymbol{\Theta}$
$\boldsymbol{\theta}_{i:j}$	Vector of parameter configurations, $[\boldsymbol{\theta}_i, \boldsymbol{\theta}_{i+1}, \ldots, \boldsymbol{\theta}_j]$
d	dimensionality of $\boldsymbol{\Theta}$ (number of parameters to be tuned)
y	Response variable (performance of target algorithm)
$\mathbf{R}$	Sequence of runs, with elements $(\boldsymbol{\theta}_i, \pi_i, s_i, \kappa_i, o_i)$ as defined in Section 1.
$\mathbf{R}[i]$	ith element of $\mathbf{R}$
$\mathbf{R}_{\boldsymbol{\theta}}$	Subsequence of runs using configuration $\boldsymbol{\theta}$ (*i.e.*, those runs with $\boldsymbol{\theta}_i = \boldsymbol{\theta}$)
$N(\boldsymbol{\theta})$	Length of $\mathbf{R}_{\boldsymbol{\theta}}$
$\hat{c}(\boldsymbol{\theta})$	Empirical cost statistic over the $N(\boldsymbol{\theta})$ runs with $\boldsymbol{\theta}$: $\hat{\tau}(\{\mathbf{R}[i].o \mid \boldsymbol{\theta}_i = \boldsymbol{\theta}\})$
$\mathcal{M}$	Predictive model
r	Number of repeats in SPO (increases over time). Initial value: SPO parameter
$\boldsymbol{\Theta}_{hist}$	Set of configurations in SPO that have been incumbents in the past (changes over time)

Table 9.1: Summary of notation used in pseudocode.

Parameter	Meaning
$initR$	Initial number of repeats in SPO
$maxR$	Maximal number of repeats in SPO
D	Size of initial design in SPO
m	Number of configurations determined with EIC to evaluate in each iteration in SPO
p	Number of previous configurations to evaluate in each iteration of SPO$^+$

Table 9.2: SPO and SPO$^+$ algorithm parameters. We used $m = 1$ and $p = 5$ throughout. For the comparison to SKO in this chapter, we changed the code to use SKO's initial design, setting $initR = 1$, $maxR = \infty$, and $D = 40$. Elsewhere, we used $initR = 2$, $D = 250$, and $maxR = 2\,000$.

9.5 Two Existing SMBO Approaches for Noisy Functions: SKO and SPO

In this section, we review two existing model-based optimization methods for noisy responses: the sequential kriging optimization (SKO) algorithm by Huang et al. (2006), and the sequential parameter optimization (SPO) procedure by Bartz-Beielstein et al. (2005; 2006). SKO uses a standard GP model, whereas SPO uses a DACE model trained on cost statistics for the parameter configurations tried so far.

9.5.1 A Framework for Sequential Model-Based Optimization

In this section, we describe SKO and SPO in a unified framework given as pseudocode in Algorithm Framework 9.1. We will use this framework to describe all approaches developed in this part of the thesis.

Table 9.1 summarizes our notation. Note in particular $N(\boldsymbol{\theta})$ and $\hat{c}(\boldsymbol{\theta})$. $N(\boldsymbol{\theta})$ denotes the number of runs we have so far executed with a parameter configuration $\boldsymbol{\theta}$; $\hat{c}(\boldsymbol{\theta})$ denotes the empirical performance across the $N(\boldsymbol{\theta})$ runs that have been performed for $\boldsymbol{\theta}$. These can be

Algorithm Framework 9.1: Sequential Model-Based Optimization.

Input : Target algorithm A
Output: Incumbent parameter configuration $\boldsymbol{\theta}_{inc}$

```
[R, θ_inc] ← Initialize()
M ← NULL
repeat
    [M, θ_inc] ← FitModel(M, R, θ_inc)
    Θ⃗_new ← SelectNewConfigurations(M, θ_inc, R)
    [R, θ_inc] ← Intensify(Θ⃗_new, θ_inc, M, R)
until total budget for configuration exhausted
return θ_inc
```

thought of as macros that respectively count the runs of $\boldsymbol{\theta}$ performed so far and compute their empirical cost statistic.

9.5.2 Initialization

We outline the initialization of SKO and SPO in Procedures 9.2 and 9.4, respectively. Procedure Initialize is called as

$$[\mathbf{R}, \boldsymbol{\theta}_{inc}] = \text{Initialize}().$$

SKO starts with a Latin hypercube design (LHD) of $10 \cdot d$ parameter configurations, where d is the number of parameters to be optimized. It executes the target algorithm at these configurations and then performs an additional run for the d configurations with the lowest response. (The execution of target algorithm runs is a common building block used in configurators; we describe it in Procedure 9.3, ExecuteRuns.) The incumbent $\boldsymbol{\theta}_{inc}$ is chosen as the configuration with the lowest empirical cost $\hat{c}(\boldsymbol{\theta})$ amongst these d configurations. In SPO, D parameter configurations are chosen according to a LHD and the target algorithm is executed r times for each of them; D and r are parameters of SPO. The incumbent $\boldsymbol{\theta}_{inc}$ is chosen as the parameter configuration with the lowest empirical cost $\hat{c}(\boldsymbol{\theta})$.

9.5.3 Fit of Response Surface Model

Both SKO and SPO base their predictions on a combination of a linear model and a GP model fitted on the residual errors of the linear model. However, both of them default to using only a single constant basis function in the linear model, thus reducing the linear model component to a constant offset term, the mean response value. Throughout this chapter, we use these defaults; the model we use is thus an offset term plus a zero-mean Gaussian stochastic process. SPO uses the DACE Matlab toolbox to construct this predictive model, while SKO implements the respective equations itself. The exact equations used in both SKO and the DACE toolbox implement methods to deal with ill conditioning; we refer the reader to the original publications for details (Huang et al., 2006; Bartz-Beielstein, 2006; Lophaven et al., 2002).

Procedure 9.2: Initialize() in SKO

Input : none (in order to avoid bloated interfaces, the parameter configuration space, $\boldsymbol{\Theta}$, and its dimension, d, are global parameters)
Output: Sequence of target algorithm runs, $\mathbf{R}$; incumbent parameter configuration $\boldsymbol{\theta}_{inc}$

$\mathbf{R} \leftarrow []$
$k \leftarrow 10 \cdot d$
$\boldsymbol{\theta}_{1:k} \leftarrow$ LatinHypercubeDesign($\boldsymbol{\Theta}$, k)
for $i = 1, \ldots, k$ **do**
    $\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}_i$, 1)
$\boldsymbol{\theta}_{k+1:k+d} \leftarrow$ the d configurations $\boldsymbol{\theta}_j$ out of $\boldsymbol{\theta}_{1:k}$ with smallest $\hat{c}(\boldsymbol{\theta}_j)$
for $i = 1, \ldots, d$ **do**
    $\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}_{k+i}$, 1)
$\boldsymbol{\theta}_{inc} \leftarrow$ random element of $\text{argmin}_{\boldsymbol{\theta} \in \{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_k\}}(\hat{c}(\boldsymbol{\theta}))$
return [$\mathbf{R}$, $\boldsymbol{\theta}_{inc}$]

Procedure 9.3: ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}$, numRuns)

Input : Sequence of target algorithm runs, $\mathbf{R}$; parameter configuration, $\boldsymbol{\theta}$; number of runs to perform, numRuns
Output: (Extended) sequence of target algorithm runs, $\mathbf{R}$

for $i = 1, \ldots,$ *numRuns* **do**
    Let s denote a previously-unused seed
    Let π denote the problem instance under consideration in the configuration scenario
    Execute $A(\boldsymbol{\theta})$ with seed s and captime κ_{max} on instance π, store response in o
    Append $(\boldsymbol{\theta}, \pi, s, \kappa, o)$ to $\mathbf{R}$
return $\mathbf{R}$

Procedure FitModel is called as

$$[\mathcal{M}, \boldsymbol{\theta}_{inc}] = \text{FitModel}(\mathcal{M}, \mathbf{R}, \boldsymbol{\theta}_{inc}).$$

When it is first called, $\mathcal{M}$ is undefined (NULL). Note that this interface allows the procedure to

Procedure 9.4: Initialize() in SPO

Input : none (parameter configuration space, $\boldsymbol{\Theta}$, is a global parameter; the size of the initial design, D, and the number of initial repetitions, $initR$, are global parameters of SPO.)
Output : Sequence of target algorithm runs, $\mathbf{R}$; incumbent parameter configuration $\boldsymbol{\theta}_{inc}$
Side Effect: Changes SPO's global parameters r (number of repetitions), and $\boldsymbol{\Theta}_{hist}$ (set of configurations that ever were incumbents)

$\mathbf{R} \leftarrow []$
$\boldsymbol{\theta}_{1:D} \leftarrow$ LatinHypercubeDesign($\boldsymbol{\Theta}$, D)
for $i = 1, \ldots, D$ **do**
    $\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}_i$, $initR$)
$\boldsymbol{\theta}_{inc} \leftarrow$ random element of $\text{argmin}_{\boldsymbol{\theta} \in \{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_D\}}(\hat{c}(\boldsymbol{\theta}))$
$\boldsymbol{\Theta}_{hist} \leftarrow \{\boldsymbol{\theta}_{inc}\}$
$r \leftarrow initR$
return [$\mathbf{R}$, $\boldsymbol{\theta}_{inc}$]

Procedure 9.5: FitModel($\mathcal{M}$, $\mathbf{R}$, $\boldsymbol{\theta}_{inc}$) in SKO

SKO fits a standard GP model to the raw response values $\mathbf{R}[i].o$ and sets its incumbent $\boldsymbol{\theta}_{inc}$ based on the learned model; n denotes the length of $\mathbf{R}$

Input : GP model, $\mathcal{M}$ (NULL at first invocation); sequence of target algorithm runs, $\mathbf{R}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$
Output: GP model, $\mathcal{M}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$

if *($\mathcal{M}$ is* NULL*) or ($n/\mathcal{M}.lastUpdate > 1.1$)* **then**
  Fit standard GP model $\mathcal{M}$ and hyper-parameters for data $\bigcup_{i=1}^{n}\{(\mathbf{R}[i].\boldsymbol{\theta}, \mathbf{R}[i].o)\}$
  $\mathcal{M}$.lastUpdate $\leftarrow n$
else
  Fit standard GP model $\mathcal{M}$ for data $\bigcup_{i=1}^{n}\{(\mathbf{R}[i].\boldsymbol{\theta}, \mathbf{R}[i].o)\}$, reusing hyper-parameters saved in previous $\mathcal{M}$
$\boldsymbol{\Theta}_{seen} \leftarrow \bigcup_{i=1}^{n}\{\mathbf{R}[i].\boldsymbol{\theta}\}$
for all $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{seen}$ **do**
  $[\mu_{\boldsymbol{\theta}}, \sigma^2_{\boldsymbol{\theta}}] \leftarrow$ Predict($\mathcal{M}$, $\boldsymbol{\theta}$)
$\boldsymbol{\theta}_{inc} \leftarrow$ random element of $\text{argmin}_{\boldsymbol{\theta}\in\boldsymbol{\Theta}}(\mu_{\boldsymbol{\theta}} + \sigma_{\boldsymbol{\theta}})$
return $[\mathcal{M}, \boldsymbol{\theta}_{inc}]$

Procedure 9.6: FitModel($\mathcal{M}$, $\mathbf{R}$, $\boldsymbol{\theta}_{inc}$) in SPO

SPO fits a DACE model to cost statistics $\hat{c}(\boldsymbol{\theta})$ (aggregates across all observed response values for $\boldsymbol{\theta}$). It does not update its incumbent $\boldsymbol{\theta}_{inc}$ in this procedure; n denotes the total number of target algorithm runs so far.

Input : DACE model, $\mathcal{M}$; sequence of target algorithm runs, $\mathbf{R}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$
Output: DACE model, $\mathcal{M}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$

$\boldsymbol{\Theta}_{seen} \leftarrow \bigcup_{i=1}^{n}\{\mathbf{R}[i].\boldsymbol{\theta}\}$
Fit DACE model $\mathcal{M}$ and hyper-parameters for data $\bigcup_{\boldsymbol{\theta}\in\boldsymbol{\Theta}_{seen}}\{\boldsymbol{\theta}, \hat{c}(\boldsymbol{\theta})\}$, with fixed $\sigma^2 = 0$
return $[\mathcal{M}, \boldsymbol{\theta}_{inc}]$

update the incumbent configuration, $\boldsymbol{\theta}_{inc}$. SKO uses this option, while SPO does not. (Instead, it updates $\boldsymbol{\theta}_{inc}$ in Procedure Intensify.)

SKO uses Gaussian stochastic process models in the conventional manner to fit noisy response data directly; we describe this in Procedure 9.5. Note that when a GP model is fitted directly on noisy response data, measurement noise is assumed to be Gaussian-distributed—an assumption that is violated in many applications of parameter optimization. While distributions of solution qualities across multiple runs of a randomized heuristic algorithm can often be approximated quite well with a Gaussian distribution, it is well known that the distributions of runtimes of randomized heuristic algorithms for solving hard combinatorial problems tend to exhibit substantially heavier tails.

As also described in Procedure 9.9, after fitting its model, SKO selects a new incumbent, $\boldsymbol{\theta}_{inc}$, based on the new model. The parameter configuration that minimizes a GP model's mean prediction is not necessarily the best choice of incumbent, because it may be based on dramatically fewer runs of the target algorithm than other, similar-scoring configurations. Recognizing this fact, SKO implements a risk-averse strategy: it picks the previously-evaluated parameter configuration that minimizes predicted mean plus one predicted standard deviation.

SPO uses GP models in the non-standard manner discussed in Section 9.3.2. It first computes the user-defined empirical cost statistic $\hat{c}(\boldsymbol{\theta})$ for each parameter configuration $\boldsymbol{\theta}$

Procedure 9.7: SelectNewConfigurations($\mathcal{M}, \boldsymbol{\theta}_{inc}, \mathbf{R}$) in SKO

Input : Model, $\mathcal{M}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$; sequence of target algorithm runs, $\mathbf{R}$
Output: Sequence of parameter configurations to evaluate, here with one element, $[\boldsymbol{\theta}_{new}]$

$\boldsymbol{\theta}_{new} \leftarrow$ the single parameter configuration found optimizing the augmented expected improvement criterion from (Huang et al., 2006) with the Nelder-Mead simplex method.
return $[\boldsymbol{\theta}_{new}]$

Procedure 9.8: SelectNewConfigurations($\mathcal{M}, \boldsymbol{\theta}_{inc}, \mathbf{R}$) in SPO
Note that m is a parameter of SPO.

Input : Model, $\mathcal{M}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$; sequence of target algorithm runs, $\mathbf{R}$
Output: Sequence of parameter configurations to evaluate, $\vec{\boldsymbol{\Theta}}_{new}$

// ===== Select m parameter configurations with expected improvement
$\boldsymbol{\Theta}_{rand} \leftarrow$ set of 10 000 elements drawn uniformly at random from $\boldsymbol{\Theta}$
for all $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{rand}$ **do**
    $[\mu_{\boldsymbol{\theta}}, \sigma^2_{\boldsymbol{\theta}}] \leftarrow$ Predict($\mathcal{M}$, $\boldsymbol{\theta}$)
    $EI(\boldsymbol{\theta}) \leftarrow$ Compute expected improvement criterion $E[I^2(\boldsymbol{\theta})]$ (see Section 10.3.2) given $\mu_{\boldsymbol{\theta}}$ and $\sigma^2_{\boldsymbol{\theta}}$
$\vec{\boldsymbol{\Theta}}_{new} \leftarrow$ list of all $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{rand}$, sorted by decreasing $EI(\boldsymbol{\theta})$
$\vec{\boldsymbol{\Theta}}_{new} \leftarrow \vec{\boldsymbol{\Theta}}_{new}[1, \ldots, m]$
return $\vec{\boldsymbol{\Theta}}_{new}$

evaluated so far, and then fits a DACE model (a noise-free GP model) to learn a mapping from parameter configurations to the cost statistic. This approach has a number of benefits and drawbacks. In addition to those discussed in Section 9.4, fitting the GP model on the cost statistic directly enables SPO to optimize almost arbitrary user-defined cost statistics, which could not be done with standard Gaussian stochastic processes. Examples include median performance, empirical variance across runs, and tradeoffs between empirical mean and variance. To our best knowledge, SPO is the only existing model-based method with such flexibility in the objective function being optimized. Another benefit is that the assumption of Gaussian-distributed response values is dropped. The final benefit of collapsing the data to a single point per parameter configuration is reduced computational complexity. While SKO has cubic scaling behaviour in the number of target algorithm runs performed, SPO only takes time cubic in the number of *distinct* parameter configurations evaluated.

9.5.4 Selection of new Parameter Settings to Evaluate

Following Jones et al. (1998), both SKO and SPO use an expected improvement criterion (EIC) to determine which parameter configurations to investigate next, thereby drawing on both the mean and variance predictions of the GP model. This criterion trades off learning about new, unknown parts of the parameter space and intensifying the search locally in the best known region.

Procedure SelectNewConfigurations is called as

$$\vec{\boldsymbol{\Theta}}_{new} = \text{SelectNewConfigurations}(\mathcal{M}, \boldsymbol{\theta}_{inc}, \mathbf{R}).$$

SKO selects a single new parameter configuration by maximizing an augmented expected improvement criterion using the Nelder-Mead simplex method (Nelder and Mead, 1965). The augmentation adds a bias away from parameter configurations for which predictive variance is low; see Huang et al. (2006). SPO, on the other hand, evaluates the $E[I^2]$ expected improvement criterion (Schonlau et al., 1998, see also Section 10.3.2) at $10\,000$ randomly selected parameter configurations and picks the m ones with highest expected improvement. Here, we use the default $m = 1$. For completeness, we give the simple pseudocode for these methods in Procedures 9.7 and 9.8.

9.5.5 Intensification

Any parameter-optimization method must make decisions about which parameter configuration $\boldsymbol{\theta}_{inc}$ to return to the user as its incumbent solution, both if interrupted during the search progress and (especially) upon completion. Candidate parameter configurations are suggested by Procedure SelectNewConfigurations, and in order to decide whether they, the current incumbent, or even another parameter configuration should become the new incumbent, it is advisable to perform additional runs of the target algorithm. Which parameter configurations to use, how many runs to execute with each of them, and how to determine the new incumbent based on those runs is specified in Procedure Intensify, which is called as

$$[\mathbf{R}, \boldsymbol{\theta}_{inc}] = \text{Intensify}(\vec{\boldsymbol{\Theta}}_{new}, \boldsymbol{\theta}_{inc}, \mathcal{M}, \mathbf{R}). \tag{9.1}$$

Note that this procedure allows an update of the incumbent, $\boldsymbol{\theta}_{inc}$. SPO makes use of this option, while SKO updates its incumbent in Procedure FitModel (see Procedure 9.5).

In order to provide more confident estimates for its incumbent, SPO implements an explicit intensification strategy. In SPO, predictive uncertainty cannot be used to decide which parameter configuration to select as incumbent. That is because its DACE model (a noise-free GP model) predicts exactly zero uncertainty at all previously-evaluated parameter configurations. An alternative measure of confidence is gained by the number of evaluations performed for a parameter configuration. SPO performs additional runs for its incumbent parameter configuration $\boldsymbol{\theta}_{inc}$ in order to make sure $\boldsymbol{\theta}_{inc}$ is a truly good parameter configuration (as opposed to having been selected since it simply happened to yield low response values in the limited number of target algorithm runs previously performed with $\boldsymbol{\theta}_{inc}$). How many evaluations are used exactly differs between SPO versions 0.3 and 0.4; Procedures 9.10 and 9.11 detail these two versions. In contrast, SKO does not implement an explicit intensification strategy; in each iteration, it only performs a single run with the selected parameter configuration.

9.6 An Experimental Comparison of SKO and SPO

To the best of our knowledge, SPO is the only sequential model-based optimization procedure that has been applied to optimizing algorithm performance. Even for SPO we are not aware of any applications for the minimization of runtime to solve decision problems. Rather, the focus in SPO's applications has been the optimization of the solution quality that a target

Procedure 9.9: Intensify($\vec{\boldsymbol{\Theta}}_{new}$, $\boldsymbol{\theta}_{inc}$, $\mathcal{M}$, $\mathbf{R}$) in SKO
SKO only performs a single target algorithm run and does not update its incumbent in this procedure.

Input : Sequence of parameter configurations to evaluate, $\vec{\boldsymbol{\Theta}}_{new}$, here with one element; incumbent configuration, $\boldsymbol{\theta}_{inc}$; model, $\mathcal{M}$; sequence of target algorithm runs, $\mathbf{R}$
Output: Sequence of target algorithm runs, $\mathbf{R}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$
1 $\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\vec{\boldsymbol{\Theta}}_{new}[1]$, 1)
2 **return** $[\mathbf{R}, \boldsymbol{\theta}_{inc}]$

Procedure 9.10: Intensify($\vec{\boldsymbol{\Theta}}_{new}$, $\boldsymbol{\theta}_{inc}$, $\mathcal{M}$, $\mathbf{R}$) in SPO 0.3
After performing runs for the new parameter configurations and for the incumbent, SPO updates its incumbent. The maximal number of runs to perform with a configuration, maxR, is a parameter of configurators using this procedure; in all our experiments, we set it to 2 000. Side effect: changes SPO's global parameters r and $\boldsymbol{\Theta}_{hist}$

Input : Sequence of parameter configurations to evaluate, $\vec{\boldsymbol{\Theta}}_{new}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$; model, $\mathcal{M}$; sequence of target algorithm runs, $\mathbf{R}$
Output: Sequence of target algorithm runs, $\mathbf{R}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$
1 **for** $i = 1, \ldots,$ *length*($\vec{\boldsymbol{\Theta}}_{new}$) **do**
2 $\quad \mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\vec{\boldsymbol{\Theta}}_{new}[i]$, r)
3 $\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}_{inc}$, $r/2$)
4 $\boldsymbol{\Theta}_{seen} \leftarrow \bigcup_{i=1}^{n} \{\mathbf{R}[i].\boldsymbol{\theta}\}$
5 $\boldsymbol{\theta}_{inc} \leftarrow$ random element of $\text{argmin}_{\boldsymbol{\theta} \in \boldsymbol{\Theta}_{seen}}(\hat{c}(\boldsymbol{\theta}))$
6 **if** $\boldsymbol{\theta}_{inc} \in \boldsymbol{\Theta}_{hist}$ **then** $r \leftarrow \min(2 \cdot r, \text{maxR})$
7 $\boldsymbol{\Theta}_{hist} \leftarrow \boldsymbol{\Theta}_{hist} \cup \{\boldsymbol{\theta}_{inc}\}$
8 **return** $[\mathbf{R}, \boldsymbol{\theta}_{inc}]$

algorithm can achieve within a given time budget. In particular, it has been applied to optimize the solution quality of CMA-ES (Hansen and Ostermeier, 1996; Hansen and Kern, 2004), a prominent gradient-free global optimization algorithm for continuous functions (Bartz-Beielstein et al., 2008). We thus used CMA-ES for the experimental comparison of SKO and SPO. (For details on CMA-ES, see Section 3.2.3.)

9.6.1 Experimental Setup

We empirically compared SKO and SPO on the four CMA-ES configuration scenarios in set BLACKBOXOPT defined in Section 3.5.1: CMAES-SPHERE, CMAES-ACKLEY, CMAES-GRIEWANGK, and CMAES-RASTRIGIN. Briefly, in these scenarios the aim is to minimize the mean solution cost across a set of CMA-ES runs on the respective function being used, where the solution cost in one run is the minimal function value CMA-ES found in the run. Each CMA-ES run is allowed a fixed number of function evaluations. For the experiments in this chapter, we fixed the budget available for configuration to 200 runs of the target algorithm, CMA-ES.

We chose this low limit of 200 target algorithm runs because the original SKO implementation was very slow: even limited to as few as 200 target algorithm runs, each SKO run took about one hour.[6] Most of SKO's time was spent in the numerical optimization of that EIC.

[6] SKO was run on a 3GHz Pentium 4 with 4 GB RAM running Windows XP Professional, Service Pack 3. We

Procedure 9.11: Intensify($\vec{\boldsymbol{\Theta}}_{new}$, $\boldsymbol{\theta}_{inc}$, $\mathcal{M}$, $\mathbf{R}$) in SPO 0.4
After performing runs for the new parameter configurations and for the incumbent, SPO updates its incumbent. The maximal number of runs to perform with a configuration, maxR, is a parameter of configurators using this procedure; in all our experiments, we set it to 2 000. Side effect: changes SPO's global parameters r and $\boldsymbol{\Theta}_{hist}$

Input : Sequence of parameter configurations to evaluate, $\vec{\boldsymbol{\Theta}}_{new}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$; model, $\mathcal{M}$; sequence of target algorithm runs, $\mathbf{R}$
Output: Sequence of target algorithm runs, $\mathbf{R}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$

for $i = 1, \ldots,$ *length*$(\vec{\boldsymbol{\Theta}}_{new})$ **do**
  $\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\vec{\boldsymbol{\Theta}}_{new}[i]$, r)
$\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}_{inc}$, $r - N(\boldsymbol{\theta}_{inc})$)
$\boldsymbol{\Theta}_{seen} \leftarrow \bigcup_{i=1}^{n} \{\mathbf{R}[i].\boldsymbol{\theta}\}$
$\boldsymbol{\theta}_{inc} \leftarrow$ random element of $\operatorname{argmin}_{\boldsymbol{\theta} \in \boldsymbol{\Theta}_{seen}} (\hat{c}(\boldsymbol{\theta}))$
if $\boldsymbol{\theta}_{inc} \in \boldsymbol{\Theta}_{hist}$ **then** $r \leftarrow \min(r + 1, \text{maxR})$
$\boldsymbol{\Theta}_{hist} \leftarrow \boldsymbol{\Theta}_{hist} \cup \{\boldsymbol{\theta}_{inc}\}$
return $[\mathbf{R}, \boldsymbol{\theta}_{inc}]$

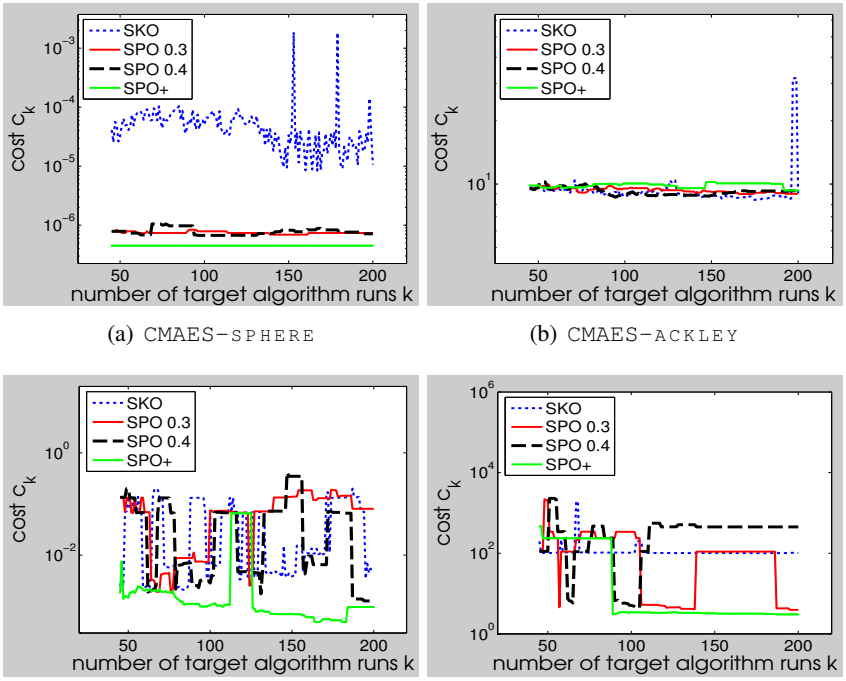
Some SKO runs actually failed due to problems in this numerical optimization. We repeated those runs until they completed. (Repetitions were non-deterministic due to measurement noise in the objective function.)

We reimplemented SPO 0.3 and 0.4 (discussed in more detail in Section 10.3.1) and verified that performance of SPO 0.4 matched that of the original SPO 0.4 implementation (Bartz-Beielstein et al., 2008). The SPO runs were substantially faster than those of SKO; they took about 2 minutes per repetition, 85% of which was spent running the target algorithm.

9.6.2 Experimental Results

Our first set of experiments for comparing SKO and SPO used original, untransformed CMA-ES solution quality as the objective function to be minimized. In order to remove one confounding factor from the study, we modified SPO to use SKO's 44-point initial design. In Figure 9.4, we show the performance of SKO and three different SPO variants: SPO 0.3 and SPO 0.4 as discussed in Section 9.5, and, for reference, SPO$^+$, which we introduce in the next chapter. Here, the only difference between these methods is in their intensification mechanism (in the next chapter, we also change additional components in SPO$^+$). On the Sphere function, the LHD already contained very good parameter configurations, and the challenge was mostly to select the best of these as its incumbent. From the figure, we observe that SPO largely succeeded in doing this, while SKO did not. On the Ackley function, SKO's performance was quite good, except for a brief interval of very poor performance after about 195 runs of the target algorithm. On the Griewangk and Rastrigin functions, the variation of performance

report wall clock time on an otherwise idle system. (We did not use Unix machines for these experiments since SKO only compiled for Windows.) In order to ascertain that the target algorithm has exactly the same behaviour it does for other configuration procedures (which run under Unix), we gathered the results of the target algorithm runs SKO requested by means of a wrapper script that connected to the same type of machine the SPO experiments were carried out on, performed the requested run of the target algorithm there and returned the result of the run. This incurred very little overhead.



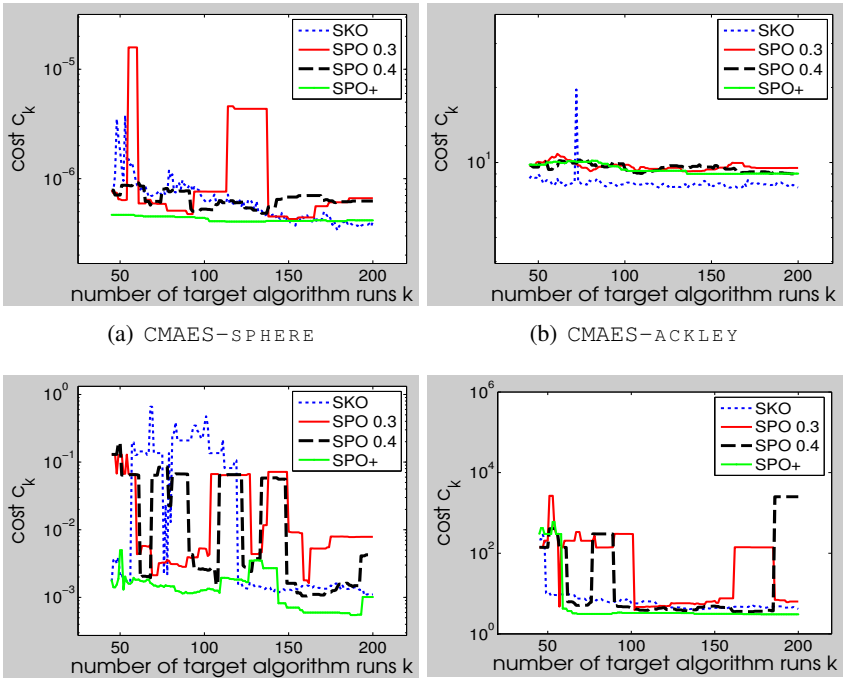
(a) CMAES-SPHERE (b) CMAES-ACKLEY

(c) CMAES-GRIEWANGK (d) CMAES-RASTRIGIN

Figure 9.4: Comparison of SKO and three variants of SPO for optimizing CMA-ES. We plot the performance p_k of each method (mean solution quality CMA-ES achieved in 100 test runs on each of the 4 test functions using the method's chosen parameter configurations) as a function of the number of algorithm runs, k, it was allowed to perform; these values are averaged across 25 runs of each method. All models were based on SKO's initial design and original untransformed data.

across multiple runs of CMA-ES was very high. Correspondingly, all approaches showed large variation in the quality of parameter configurations selected over time. (Intuitively, the configurator detects a new region, which seems promising based on a few runs of the target algorithm. Then, after additional target algorithm runs, that region is discovered to be worse than initially thought. During the period it takes to discover the true, poor nature of the region, the search returns a poor incumbent.)

Secondly, we experimented with log transformations as described in Section 9.4. As discussed there, in the case of SKO a log transform of the noisy data leads to a fit of the geometric instead of the arithmetic mean. Despite this fact, the log transformation improved SKO performance quite substantially. Figure 9.5 shows that for the SPO variants a log transformation—in this case of cost statistics—did not improve performance as much. We attribute this to the fact that the quality of the model is less important in SPO than in SKO. While SPO "only" uses the model in order to make decisions about the next parameter configuration to evaluate, SKO also uses it in order to select its incumbent. After the log



(a) CMAES-SPHERE (b) CMAES-ACKLEY

(c) CMAES-GRIEWANGK (d) CMAES-RASTRIGIN

Figure 9.5: Comparison of SKO and three variants of SPO for optimizing CMA-ES, with log-transformations. We plot the performance p_k of each method (mean solution quality CMA-ES achieved in 100 test runs on each of the 4 test functions using the method's chosen parameter configurations) as a function of the number of algorithm runs, k, it was allowed to perform; these values are averaged across 25 runs of each method. All models were based on SKO's initial design.

transformation SKO and SPO$^+$ performed comparably.

9.7 Chapter Summary

In this chapter, we introduced a general framework for Sequential Model-Based Optimization (SMBO) that we will use throughout this part of the thesis. We described and experimentally compared two existing instantiations of this general framework from the literature, Sequential Kriging Optimization (SKO) by Huang et al. (2006) and Sequential Parameter Optimization (SPO) by Bartz-Beielstein et al. (2005).

In this comparison, SPO performed much more robustly "out-of-the-box", whereas SKO became very competitive when using a log transformation of the response values.

Chapter 10

Improvements I: An Experimental Investigation of SPO Components

There is no higher or lower knowledge, but one only, flowing out of experimentation.
—Leonardo da Vinci, Italian inventor and artist

In this chapter[1], we investigate the sequential parameter optimization (SPO, see Section 9.5) framework in more depth. Three main reasons motivated us to follow this path—rather than that of the alternative sequential kriging optimization (SKO, also described in Section 9.5)—in order to build effective model-based algorithm configuration procedures. Firstly, our main interest is in complex scenarios, in which the predictive model might not actually perform very well. In such scenarios, we believe it is important to employ an explicit intensification criterion (as, *e.g.*, implemented by SPO) instead of relying on the model alone to select incumbents. Secondly, SPO has the advantage of being able to optimize a variety of user-defined objective functions, while the standard Gaussian process model underlying SKO can only fit the mean of the data. In practice, users might be more interested in statistics other than mean performance, such as the best out of ten algorithm runs. SPO supports such a criterion and also allows for many other options. Finally, the SKO implementation we used was simply too slow to be applied to the algorithm configuration scenarios we are interested in, some of which require the execution of tens of thousands of target algorithm runs. (Recall from Section 9.6.1 that SKO's complexity is cubic in that number of algorithm runs, n_{runs}, and that it already required an hour for $n_{runs} = 200$.) In the next chapter, we will again consider models similar to those used by SKO but will employ approximation techniques in order to reduce their time complexity.

In what follows, we thoroughly investigate the components of SPO. First, we investigate the quality of the initial predictive models, depending on (1) the initial design and (2) whether or not we use a log transformation. Then, we study components of the sequential optimization

[1]This chapter is based on published joint work with Holger Hoos, Kevin Leyton-Brown, Kevin Murphy, and Thomas Bartz-Beielstein (Hutter et al., 2009e,a).

process: (3) the intensification mechanism and (4) various expected improvement criteria (EIC). Based on this study of components, we introduce a novel variant of SPO—dubbed *SPO*$^+$—which differs from SPO in its use of a log transformation and, most importantly, in its intensification mechanism. We demonstrate that on the type of simple configuration scenario SPO$^+$ is applicable for—optimizing numerical algorithm parameters on a single problem instance—it is competitive with state-of-the-art algorithm configuration procedures (including BasicILS and FocusedILS). Finally, we bound the computational overhead due to learning the model and optimizing EIC, leading to another new version of SPO—dubbed SPO*—which finds good parameter configurations substantially faster than SPO$^+$.

10.1 Experimental Setup

In this chapter we use the BLACKBOXOPT set of configuration scenarios. In particular, we continue to use the four CMA-ES configuration scenarios considered in the previous chapter. We also study configuration scenario SAPS-QWH, in which the objective is to minimize median SAPS runtime (measured in local search steps) on a single problem instance (see Section 3.5.1 for details). We chose this configuration scenario to facilitate a direct comparison of the performance achieved by the algorithm configuration procedures considered here and in the work previously described in Section 5.4.

10.2 Model Quality

It is not obvious that a model-based algorithm configuration procedure needs models that accurately predict target algorithm performance across all parameter configurations, including very bad ones. Nevertheless, all else being equal, models with good overall accuracy can generally be expected to be helpful to such methods, and are furthermore essential to more general tasks such as performance robustness analysis. In this section, we investigate the effect of two key model-design choices on the accuracy of the GP models used by SPO: whether or not we use a transformation of cost statistics, and how we select the configurations in the initial design. First, we describe our measures of model quality.

10.2.1 Measures of Model Quality

Straightforward candidates for quantitative model performance measures include root mean squared error (RMSE) of the predictions, log-likelihood of the observed data under the predictive distribution, and the Pearson correlation coefficient between predictions and the actual response. However, none of these well reflects the context in which we use model predictions. In particular, we solely use predictive distributions of a parameter configuration's cost measure—we never use predictions for performance in single runs. We use the model for two main purposes: (1) predicting which out of two parameter configurations will have better mean performance, and (2) selecting the next parameter configuration(s) to evaluate based on an expected improvement criterion (EIC). We capture performance in task (1) by computing the quality of predictive ranks, defined as follows.

Definition 11 (Spearman correlation coefficient)**.** *Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two vectors of the same length, n. Let $\boldsymbol{r}$ and $\boldsymbol{s}$ be the corresponding vectors of ranks, where ties lead to average ranks. Then, the* Spearman correlation coefficient *between vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is the Pearson correlation coefficient ρ between vectors $\boldsymbol{r}$ and $\boldsymbol{s}$:*

$$\rho = \frac{n(\sum_{i=1}^{n} r_i s_i) - (\sum_{i=1}^{n} r_i)(\sum_{i=1}^{n} s_i)}{\sqrt{n(\sum_{i=1}^{n} r_i) - (\sum_{i=1}^{n} r_i)^2}\sqrt{n(\sum_{i=1}^{n} s_i) - (\sum_{i=1}^{n} s_i)^2}}. \tag{10.1}$$

Definition 12 (Quality of predictive ranks)**.** *The* quality of predictive ranks *of a model $\mathcal{M}$ on a test set of parameter configurations $\{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_n\}$ is the Spearman correlation coefficient between validation costs $c_{valid}(\boldsymbol{\theta}_1), \ldots, c_{valid}(\boldsymbol{\theta}_n)$ and model predictions $\hat{c}(\boldsymbol{\theta}_1), \ldots, \hat{c}(\boldsymbol{\theta}_n)$.*

The quality of predictive ranks lies in the interval $[-1, 1]$, with 1 indicating perfect correlation of the predicted and the true ranks, 0 indicating no correlation and -1 perfect anti-correlation.

The second purpose for which we use models is to compute an expected improvement criterion (EIC) to select new parameter configurations. Optimally, we would like to select configurations of the highest possible quality. Thus, we assess performance in this task by the correlation coefficient of ranks between EIC and test set performance on a set of test configurations. This quantity depends both on the EIC used and on the model.

Definition 13 (EIC quality)**.** *The* EIC quality *of a combination of model $\mathcal{M}$ and an EIC on a set of parameter configurations $\{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_n\}$ is the Spearman correlation coefficient between validation costs $c_{valid}(\boldsymbol{\theta}_1), \ldots, c_{valid}(\boldsymbol{\theta}_n)$ and expected improvements, $EI(\boldsymbol{\theta}_1), \ldots, EI(\boldsymbol{\theta}_n)$.*

In order to also quantify overall model quality, we use the standard measure of root mean squared error.

Definition 14 (RMSE)**.** *The* root mean squared error (RMSE) *of a model $\mathcal{M}$ on a set of parameter configurations $\{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_n\}$ is the square root of average squared prediction error $\sqrt{\frac{1}{n}\sum_{i=1}^{n}[c_{valid}(\boldsymbol{\theta}_i) - \hat{c}(\boldsymbol{\theta}_i)]^2}$.*

We use these quantitative measures throughout this part of the thesis to evaluate model quality. In order to compute them, we require a set of parameter configurations $\{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_n\}$ and their test set performance, whose computation is expensive and does not happen during the configuration process, but rather offline for validation purposes. Note, however, that for each set of parameter configurations, we only need to compute this test performance once and can then use it to evaluate arbitrary models. We used two different sets: *Random*, a set of 100 parameter configurations sampled uniformly at random from the parameter configuration space; and *Good*, a set of well-performing parameter configurations. We determined this second set of configurations with a subsidiary algorithm configuration procedure, P, that performed well on the respective configuration scenario. We specify the procedure P used and its budget on a case-by-case basis. In each case, we executed 25 runs of P, keeping track of the set $\mathcal{C}$ of all configurations it ever labeled as incumbents during the configuration process. We selected set *Good* as a random subset of $\mathcal{C}$ of size 100, or, when $\mathcal{C}$ contained less than 100 configurations, used *Good*=$\mathcal{C}$. We compute our quantitative measures on set *Good* since we

are primarily interested in distinguishing excellent parameter configurations from merely good ones.

10.2.2 Transforming Performance Data

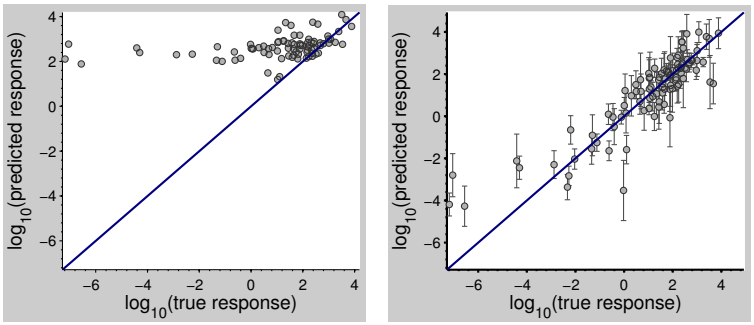
We first qualitatively study model performance along the lines of the approach used by Jones et al. (1998) and then employ our quantitative measures. Jones et al. suggested the use of model validation techniques to ensure that the models are sufficiently accurate in order to guide the search for the global optimum. They employed leave-one-out cross-validation. That is, given a training set with n data points, they learned n models, each of them on subsets of $n - 1$ data points. Each model was then used to predict the response value for the data point not used to train that model. We apply a similar 10-fold cross-validation technique: we learn 10 models, each of them on 90% of the training data, and used each model to predict the response values for the 10% of the data not used to train that model.

Here, we use this approach to study whether the log transformation of cost statistics we used for SPO in the previous chapter did indeed yield more accurate models than could be obtained by using the original, untransformed cost statistics. We show diagnostic plots for a single model for configuration scenario CMAES-SPHERE and summarize data across many models built on independent training data and other configuration scenarios with our quantitative measures. All models in this section are learned on 500 training data points. Each such point represents target algorithm performance for $d = 250$ parameter configurations determined with a random Latin hypercube, performing $r = 2$ repetitions each. Here, we focus on models fitted to data from this initial design (instead of data gathered throughout the sequential optimization process) in order to study model performance in isolation, without taking into account interactions between the model and the sequential optimization process.

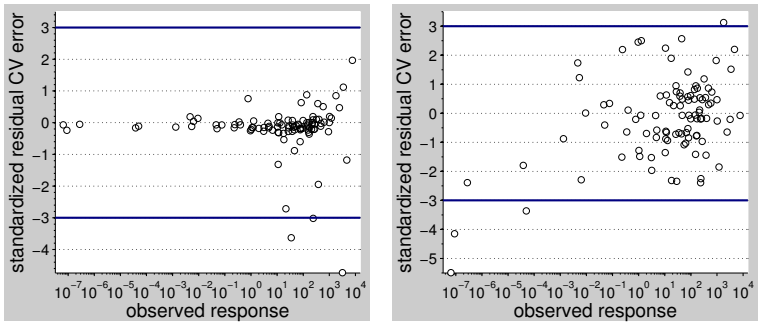
Figure 10.1 shows the diagnostic plots of Jones et al. (1998) for scenario CMAES-SPHERE, for DACE models trained on untransformed and log-transformed cost statistics. The first type of diagnostic plot (Figures 10.1(a) and 10.1(b)) simply shows actual response *vs* predicted response. We also plot standard deviations of the predictions for the model trained on log statistics; we omit these for the model trained on untransformed statistics since the predictive distribution of this model is a Gaussian distribution in untransformed space (which cannot be plotted properly on the log-log axis since mean minus standard deviation is often negative). We notice that the model trained on log statistics clearly fitted the data better, especially in the sparsely-populated region with very good parameter configurations.

The second type of diagnostic plot (Figures 10.1(c) and 10.1(d)) shows the standardized residual error for each data point i. This is the error of the mean prediction at the data point, $\mu_i - o_i$, normalized by the predicted standard deviation at i, σ_i. Optimally, these residual errors would be close to Gaussian-distributed. If this were the case, around 97% of them should lie within the interval $[-3, 3]$. Almost all data points indeed lie in this interval. For both models we see trends for data points with larger response values: for the model trained on the untransformed data, uncertainty estimates get too low, whereas for the model trained on log-transformed data they get too high.

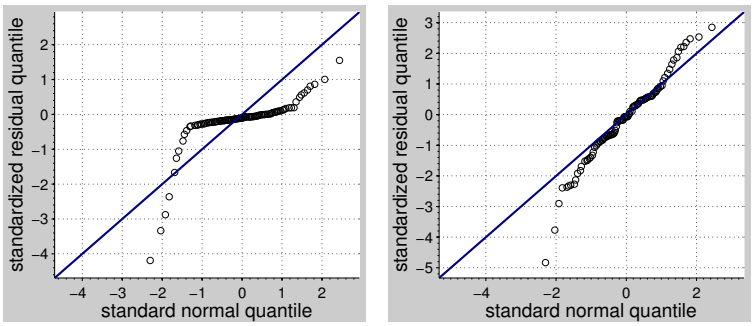
Finally, the third type of plot (Figures 10.1(e) and 10.1(f)) assesses how closely the distribution of these standardized residuals matches a Gaussian distribution. The closer the



(a) Predicted *vs* actual response, orig (b) Predicted *vs* actual runtime, log



(c) Standardized residual errors, orig (d) Standardized residual errors, log



(e) Quantile-quantile plot, orig (f) Quantile-quantile plot, log

Figure 10.1: Diagnostic plots of Jones et al. (1998) for scenario CMAES-SPHERE. The left column fits a DACE model to the original, untransformed cost statistics, the right column fits a DACE model to log-transformed cost statistics. These plots can be generated online (at runtime of the configuration procedure, without extra target algorithm runs) using cross-validation.

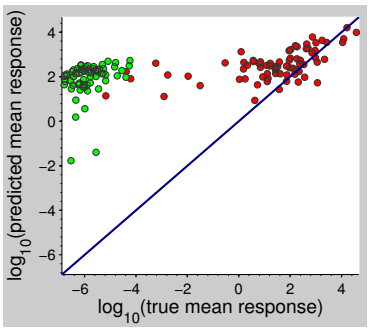
points in this quantile-quantile plot are to the diagonal, the better the match. Here, the standardized residuals for the model trained on log statistics were quite close to Gaussian-distributed. For the model trained on untransformed data, the fit was quite poor with too much of the distribution's mass close to zero.

In Figure 10.2, we evaluate the quality of the same two models for predicting the cost (mean response across many algorithm runs) of our validation sets of configurations, *Good* and *Random*. Since we estimate the true costs of these configurations in an offline evaluation, we refer to these plots as *offline diagnostic plots* as opposed to the (online) diagnostic plots suggested by Jones et al. (1998). The results are quite similar to those in Figure 10.1, with the important distinction that the offline plots contain many more data points with low mean response. This is of course due to the fact that in our offline evaluation we not only evaluate model quality on randomly sampled configurations (test set *Random*, plotted in red) but also on a set of good parameter configurations (test set *Good*, plotted in green). In this case, test set *Good* was gathered with the SPO$^+$ configuration procedure (which we introduce in Section 10.3) with a budget of $1\,000$ target algorithms. We used the same procedure to create the respective test sets *Good* for the other CMA-ES scenarios we consider in this section; for SAPS-QWH, we allowed a budget of $20\,000$ target algorithms for this process.

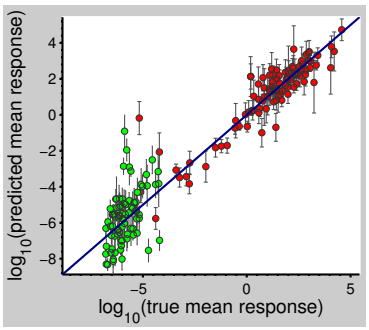
The fit in the first row of figures is slightly better in the offline evaluation, which can be explained by the reduced noise. Test set costs are mean solution costs over 100 repetitions of CMA-ES, whose variance is a factor of $\sqrt{100} = 10$ lower than for single responses. For the model fit on untransformed data (see Figure 10.2(d)), the plot of standardized residual errors is very similar to the online crossvalidation plot, with the exception of an outlier for the parameter configuration with the largest mean response. For the model fit on log-transformed data, however, the many configurations with low mean response reveals a new pattern: uncertainty estimates degrade for good parameter configurations (see Figure 10.2(d)). In fact, in order to avoid a degenerate plot, we had to omit two data points with true mean response $< 10^{-6}$ and standardized residual error of around 500 and 2200, respectively.

The third type of plot also looks quite similar to the one for the online cross-validation plots. For the model trained on untransformed data (Figure 10.2(e)), the outlier is the main difference. For the model trained on log-transformed data (Figure 10.2(f)), the many good parameter configurations—for which predictive uncertainty was worse—led to a distribution of standardized residual error much more different from a Gaussian.

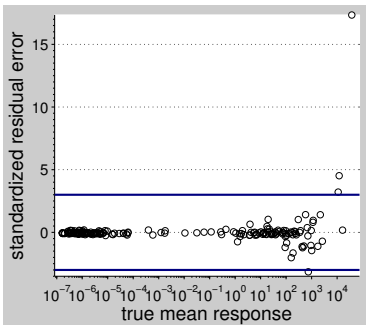
We now study quantitative model performance. While the figures above are an important tool to aid our qualitative understanding of model performance, model performance often differs substantially across independent repetitions of the same procedure with different training data, as well as across domains. We capture this in our quantitative measures. First, we give our quantitative measures for the data shown in Figure 10.2. Here, the quality of predictive ranks was much better for the model trained on log-transformed cost statistics: 0.68 *vs* 0.39 for the model trained on the original data. Likewise, the EIC quality (here, $E[I^2]$, described in Section 10.3.2) was higher: 0.72 *vs* 0.24. Finally, RMSE was also much better (lower) when training on log-transformed cost statistics: 1.26 *vs* 589. In Figure 10.3, we show boxplots of these performance measures for 25 independent repetitions for each of the configuration scenarios considered in this chapter. This figure shows that the



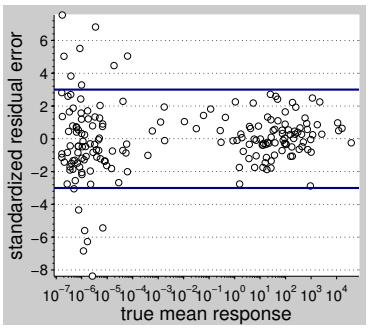
(a) Predicted *vs* actual response, orig. Test sets: *Good*, green; *Random*, red



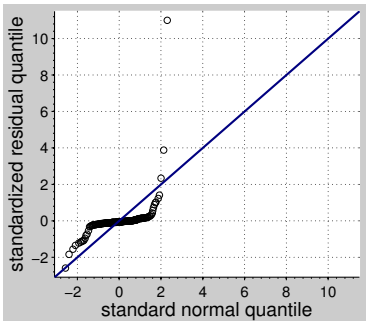
(b) Predicted *vs* actual runtime, log. Test sets: *Good*, green; *Random*, red



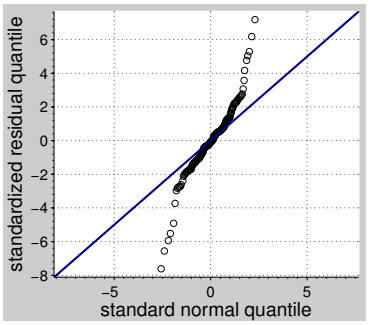
(c) Standardized residual errors, orig



(d) Standardized residual errors, log

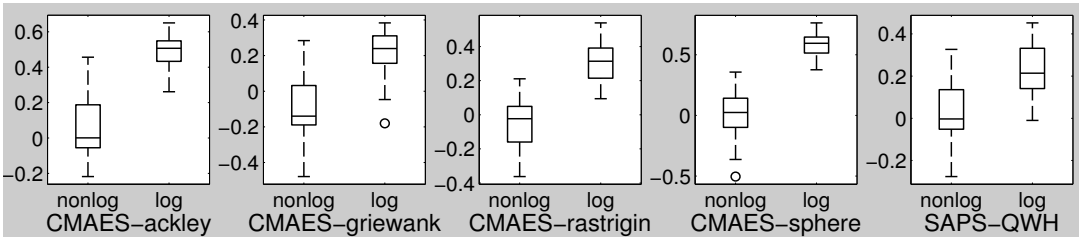


(e) Quantile-quantile plot, orig

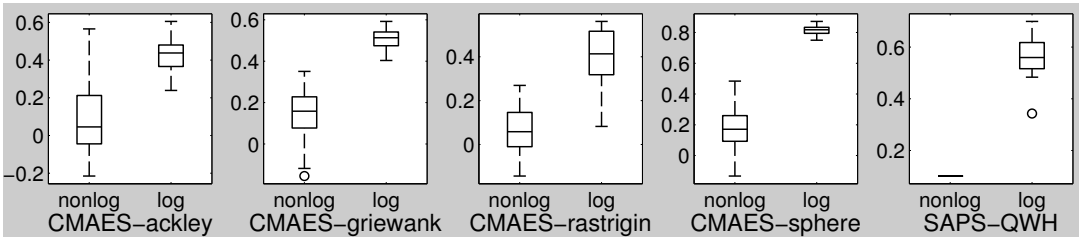


(f) Quantile-quantile plot, log

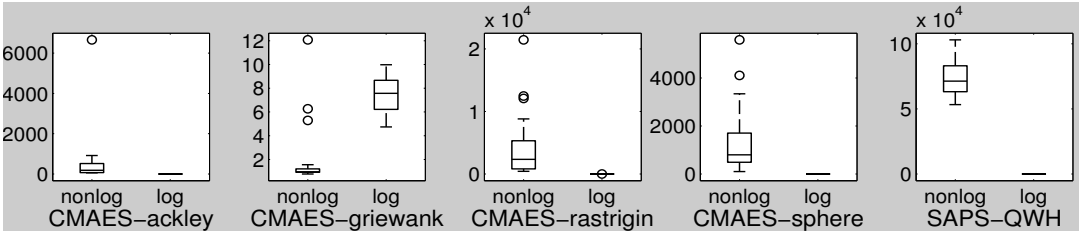
Figure 10.2: Offline diagnostic plots for scenario `CMAES-SPHERE`. The left column fits a DACE model to the original, untransformed cost statistics, the right column fits a DACE model to log-transformed cost statistics. These plots require knowledge of the true mean response values for each test configuration—which need to be computed offline. Not shown: two data points with true mean response $< 10^{-6}$ and standardized residual error around 500 and 2200, respectively.



(a) Quality of predictive ranks (high is good, 1 is optimal)



(b) EIC quality (high is good, 1 optimal. For scenario SAPS-QWH, in all 25 runs EIC was zero (up to numerical precision) for all but one configuration, yielding identical (low) EIC quality



(c) Root mean squared error (RMSE; low is good, 0 is optimal)

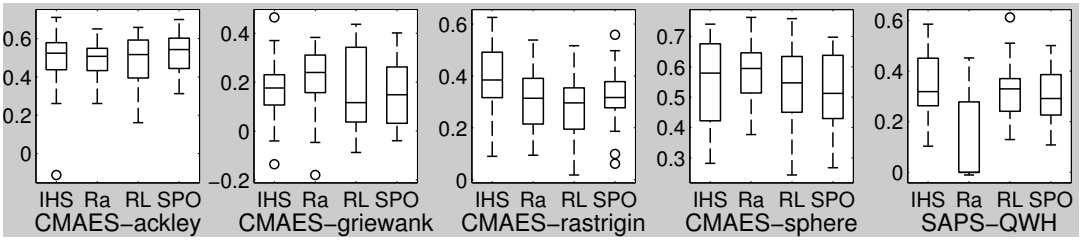
Figure 10.3: Comparison of models based on log-transformed and original data. We plot boxplots across 25 repetitions with different training but identical test data.

log transformation consistently improved both the quality of predictive ranks and the EIC quality across the five configuration scenarios. It also decreased RMSE (by many orders of magnitude) for four scenarios, but increased it for scenario CMAES-GRIEWANGK.

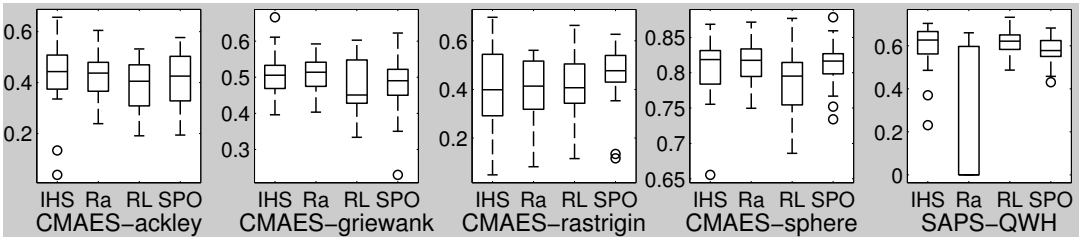
10.2.3 Choosing the Initial Design

In the previous section, we used an initial design of $d = 250$ data points with $r = 2$ repetitions each. The effects of d and r were studied before by Bartz-Beielstein and Preuss (2006), and we thus fixed these variables in this study.[2] Specifically, we used $r = 2$ and $d = 250$, such that when methods were allowed $1\,000$ runs of the target algorithm, half of them were chosen

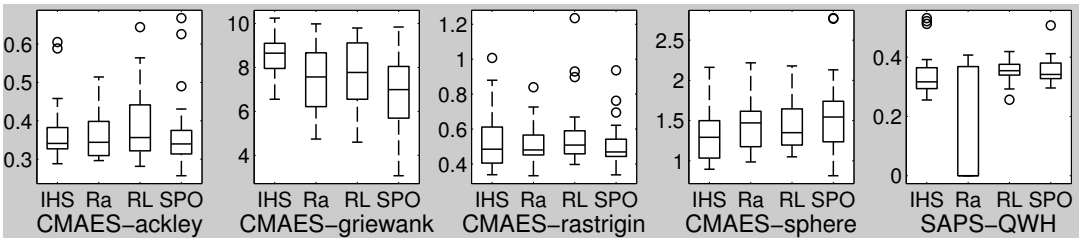
[2]However, we note that the first author of that study has—in personal communication—expressed that he still considers it unclear how to best choose the initial design size.



(a) Quality of predictive ranks (high is good, 1 is optimal)



(b) EIC quality (high is good, 1 is optimal)



(c) Root mean squared error (RMSE; low is good, 0 is optimal)

Figure 10.4: Comparison of models based on different initial designs. We show boxplots across 25 repetitions with different training but identical test data. In each plot, "IHS" denotes iterated hypercube sampling, "Ra" denotes randomly sampled design points, "RL" denotes a random LHD, and "SPO" denotes the LHD SPO uses.

by the initial design.

Here, we study the effect of the method for choosing *which* 250 parameter configurations to include in the initial design, considering four methods: (1) a uniform random sample from the region of interest; (2) a random Latin hypercube design (LHD); (3) the LHD used in SPO; and (4) a more complex LHD based on iterated distributed hypercube sampling (IHS) (Beachkofski and Grandhi, 2002).

We summarize the results of this analysis in Figure 10.4. From this figure, we see that all initial designs led to similar performance. The only striking difference was for scenario SAPS-QWH, where the pure random initialization strategy sometimes resulted in very poor models. For the other scenarios, even pure random initialization resulted in models of comparable performance. Note that this does not contradict results from the literature; for

example, Santner et al. (2003) state on page 149: "It has not been demonstrated that LHDs are superior to any designs other than simple random sampling (and they are only superior to simple random sampling in some cases)." Breaking the tie by conceptual simplicity, we thus use simple random LHDs in the remainder of this thesis.

10.3 Sequential Experimental Design

Having studied the initial design, we now turn our attention to the sequential search for performance-optimizing parameters. Since log transformations consistently led to improved performance and random LHDs yielded comparable performance to more complex designs, we fixed these two design choices.

10.3.1 Intensification Mechanism

In order to achieve good results when optimizing parameters based on a noisy cost statistic (such as runtime or solution quality achieved by a randomized algorithm), it is important to perform a sufficient number of runs for the parameter configurations considered. However, runs of a given target algorithm on interesting problem instances are typically computationally expensive. Thus, we face Question 5 discussed in Chapter 4: how should we trade off the number of configurations evaluated against the number of runs used in these evaluations?

Realizing the importance of this tradeoff, SPO implements an intensification mechanism: a procedure for gradually increasing the number of runs performed for each parameter configuration. In particular, SPO increases the number of runs to be performed for each subsequent parameter configuration whenever the incumbent $\boldsymbol{\theta}_{inc}$ selected in an iteration was already the incumbent in a previous iteration. SPO 0.3 (Bartz-Beielstein et al., 2005; Bartz-Beielstein, 2006; Bartz-Beielstein and Preuss, 2006) doubles the number of runs for subsequent function evaluations whenever this happens, SPO 0.4 only increments the number of runs by one each time. (We gave pseudocode for this in Section 9.5.5; see Procedures 9.10 and 9.11 on pages 142 and 143.) Both versions perform additional runs for the current incumbent, $\boldsymbol{\theta}_{inc}$, to make sure it gets as many function evaluations as new parameter configurations. [3]

While the intensification mechanisms of SPO 0.3 and SPO 0.4 work in most cases, we have encountered runs of SPO in high-noise scenarios in which there is a large number of parameter configurations with a few runs and "lucky" function evaluations, making these configurations likely to become incumbents. In those runs, a new incumbent was picked in almost every iteration, because the previous incumbent had been found to be poor after performing additional runs on it. This situation continued throughout the entire algorithm configuration trajectory, leading to a final choice of parameter configurations that had only been evaluated using very few ("lucky") runs and that performed poorly subsequently.

This observation motivated us to introduce a different intensification mechanism that

[3] Another approach for allocating an appropriate number of function evaluations to each parameter configuration is *optimal computational budget allocation (OCBA)* by Chen et al. (2000). Lasarczyk (2007) implemented OCBA as an explicit intensification method to improve SPO's selection procedure, especially in high-noise scenarios, but that implementation is not available yet. (It will be available in the next release of SPO, and it would be interesting to experiment with it then.)

Procedure 10.1: Intensify($\vec{\boldsymbol{\Theta}}_{new}$, $\boldsymbol{\theta}_{inc}$, $\mathcal{M}$, $\mathbf{R}$) in SPO$^+$
Recall that $N(\boldsymbol{\theta})$ denotes the number of algorithm runs which have been performed for $\boldsymbol{\theta}$, *i.e.*, the length of $\mathbf{R}_{\boldsymbol{\theta}}$. The maximal number of runs to perform with a configuration, maxR, is a parameter of configurators using this procedure; in all our experiments, we set it to 2 000.

Input : Sequence of parameter configurations to evaluate, $\vec{\boldsymbol{\Theta}}_{new}$, here with one element; incumbent configuration, $\boldsymbol{\theta}_{inc}$; model, $\mathcal{M}$; sequence of target algorithm runs, $\mathbf{R}$
Output: Sequence of target algorithm runs, $\mathbf{R}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$

for $i = 1, \ldots,$ *length*$(\vec{\boldsymbol{\Theta}}_{new})$ **do**
  $\boldsymbol{\theta} \leftarrow \vec{\boldsymbol{\Theta}}_{new}[i]$
  $r \leftarrow 1$
  $\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}$, 1)
  numBonus $\leftarrow 1$
  if $N(\boldsymbol{\theta}) > N(\boldsymbol{\theta}_{inc})$ **then**
    $\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}_{inc}$, 1)
    numBonus $\leftarrow 0$
  while *true* **do**
    if $\hat{c}(\boldsymbol{\theta}) > \hat{c}(\boldsymbol{\theta}_{inc})$ **then**
      // ===== Reject challenger, perform bonus runs for $\boldsymbol{\theta}_{inc}$
      $\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}_{inc}$, $\min($numBonus, maxR $- N(\boldsymbol{\theta}_{inc}))$)
      break
    if $N(\boldsymbol{\theta}) \geq N(\boldsymbol{\theta}_{inc})$ **then**
      // ===== Challenger becomes incumbent
      $\boldsymbol{\theta}_{inc} \leftarrow \boldsymbol{\theta}$
      break
    if *total budget for configuration exhausted* **then**
      return $[\boldsymbol{\theta}_{inc}, \mathbf{R}]$
    $r \leftarrow \min(r \cdot 2, N(\boldsymbol{\theta}_{inc}) - N(\boldsymbol{\theta}))$
    $\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}$, r)
    numBonus $\leftarrow$ numBonus $+r$
return $[\mathbf{R}, \boldsymbol{\theta}_{inc}]$

guarantees increasing confidence in the performance of the parameter configurations we select as incumbents. In particular, inspired by the mechanism used in FocusedILS (Hutter et al., 2007b), we maintain the invariant that we never choose an incumbent unless it is the parameter configuration with the most function evaluations. Promising parameter configurations receive additional function evaluations until they either cease to appear promising or receive enough function evaluations to become the new incumbent. We provide pseudocode for this new intensification mechanism in Procedure 10.1.

In detail, our new intensification mechanism works as follows. In the first iteration, the incumbent is chosen exactly as in SPO, because at this point, all parameter configurations receive the same number of function evaluations. From then on, in each iteration we select a set of parameter configurations and compare them to the incumbent $\boldsymbol{\theta}_{inc}$. Denote the number of runs that have so far been executed with parameter configuration $\boldsymbol{\theta}$ as $N(\boldsymbol{\theta})$, and the corresponding empirical performance as $\hat{c}_N(\boldsymbol{\theta})$. For each selected configuration $\boldsymbol{\theta}$,

Procedure 10.2: SelectNewParameterConfigurations($\mathcal{M}, \boldsymbol{\theta}_{inc}, \mathbf{R}$) in SPO$^+$
Recall that p and m are parameters of SPO$^+$. We used $m = 1$ and $p = 5$ in our experiments.

Input : Model, $\mathcal{M}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$; sequence of target algorithm runs, $\mathbf{R}$
Output: Sequence of parameter configurations to evaluate, here with one element, $[\boldsymbol{\theta}_{new}]$

// ===== Select p previously-used parameter configurations
$\boldsymbol{\Theta}_{seen} \leftarrow \bigcup\{\mathbf{R}[1].\boldsymbol{\theta}, \ldots, \mathbf{R}[n].\boldsymbol{\theta}\}$
$\vec{\boldsymbol{\Theta}}_{new} \leftarrow p$ elements $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{seen}$, drawn with probability $\propto 1/\hat{c}(\boldsymbol{\theta})$ and without repetitions
// ===== Select m parameter configurations with expected improvement
$\boldsymbol{\Theta}_{rand} \leftarrow$ set of 10 000 elements drawn uniformly at random from $\boldsymbol{\Theta}$
for $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{rand}$ **do**
    $[\mu_{\boldsymbol{\theta}}, \sigma^2_{\boldsymbol{\theta}}] \leftarrow$ Predict($\mathcal{M}$, $\boldsymbol{\theta}$)
    $EI(\boldsymbol{\theta}) \leftarrow$ Compute expected improvement criterion $E[I^2(\boldsymbol{\theta})]$ given $\mu_{\boldsymbol{\theta}}$ and $\sigma^2_{\boldsymbol{\theta}}$
Append to $\vec{\boldsymbol{\Theta}}_{new}$ all $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{rand}$, sorted by decreasing $EI(\boldsymbol{\theta})$
$\vec{\boldsymbol{\Theta}}_{new} \leftarrow \vec{\boldsymbol{\Theta}}_{new}[1, \ldots, p + m]$
return $\vec{\boldsymbol{\Theta}}_{new}$

we iteratively perform runs until $N(\boldsymbol{\theta}) \geq N(\boldsymbol{\theta}_{inc})$ and/or $\hat{c}_N(\boldsymbol{\theta}) > \hat{c}_N(\boldsymbol{\theta}_{inc})$.[4] Whenever we reach a point where $N(\boldsymbol{\theta}) \geq N(\boldsymbol{\theta}_{inc})$ and $\hat{c}_N(\boldsymbol{\theta}) \leq \hat{c}_N(\boldsymbol{\theta}_{inc})$, we select $\boldsymbol{\theta}$ as the new incumbent. On the other hand, if we ever observe $\hat{c}_N(\boldsymbol{\theta}) > \hat{c}_N(\boldsymbol{\theta}_{inc})$, we reject $\boldsymbol{\theta}$. Note that this criterion for rejection is very aggressive. Indeed, rejection frequently occurs after a single run, at a point where a statistical test would not be able to conclude that $\boldsymbol{\theta}$ is worse than $\boldsymbol{\theta}_{inc}$. Upon rejecting a configuration $\boldsymbol{\theta}$, we also perform as many additional runs for $\boldsymbol{\theta}_{inc}$ as were just performed for evaluating $\boldsymbol{\theta}$. This ensures that we use a comparable number of runs for intensification as for exploration of new parameter configurations.

The parameter configurations we evaluate against $\boldsymbol{\theta}_{inc}$ at each iteration include one new parameter configuration selected based on an expected improvement criterion (here $E[I^2]$, see Section 10.3.2). They also include p previously-evaluated parameter configurations $\boldsymbol{\theta}_{1:p}$, where p is an algorithm parameter and in this work always set to 5. This set is constructed by selecting p previously-evaluated configurations $\boldsymbol{\theta}$ with probability proportional to $1/\hat{c}(\boldsymbol{\theta})$, without replacement. Procedure 10.2 provides pseudocode for this selection of parameter configurations. This is called by the general SMBO framework, and the resulting parameter configurations are evaluated against $\boldsymbol{\theta}_{inc}$ in Procedure 10.1.

This mechanism of re-evaluating previous configurations guarantees that at each step there will be a positive probability of revisiting a potentially-optimal parameter configuration after it has been rejected. It allows us to be aggressive in rejecting new candidates, since we can always revisit promising ones. Note that if the other SPO variants (0.3 and 0.4) discover the true optimal parameter configuration but observe one or more very "unlucky" runs on it, they will not revisit that configuration again. This occurs because for visited parameter configurations with poor empirical performance across their performed runs the noise-free Gaussian process model attributes a high mean and zero uncertainty. Thus no expected improvement criterion will pick it again in later iterations.

[4] We batch runs to reduce overhead, starting with a single new run for each $\boldsymbol{\theta}$ and doubling the number of new runs iteratively up to a maximum of $N(\boldsymbol{\theta}_{inc}) - N(\boldsymbol{\theta})$ runs.

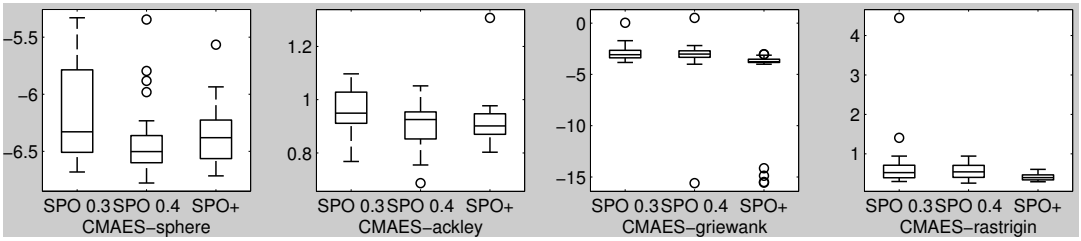


Figure 10.5: Comparison of the intensification mechanisms in SPO 0.3, SPO 0.4, and SPO$^+$. We show boxplots of performance $p_{1\,000}$ (mean solution quality CMA-ES achieved in 100 test runs using the procedure's chosen parameter configurations) achieved in the 25 runs of each tuner for each test case.

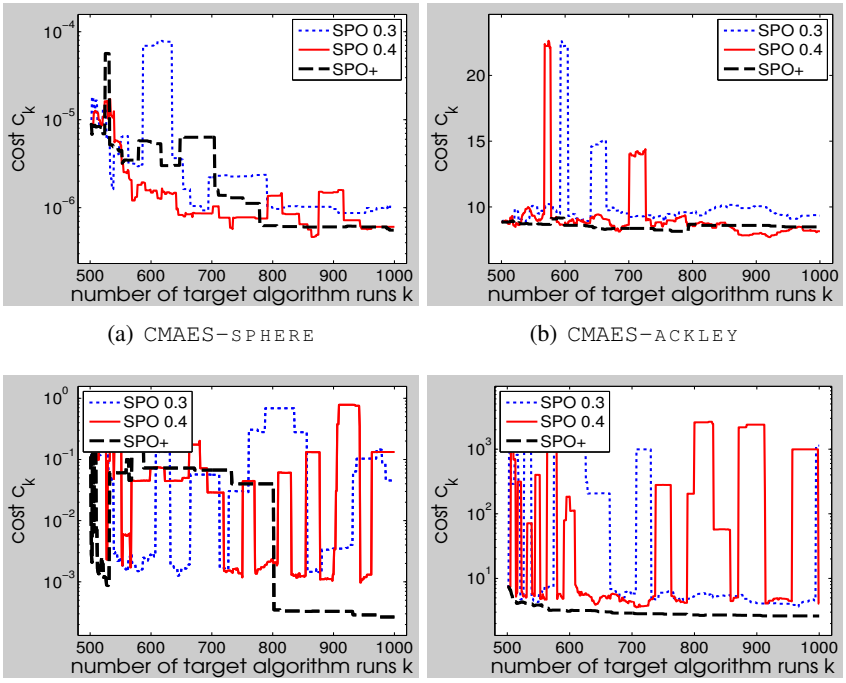
	Sphere	Ackley	Griewangk	Rastrigin
SPO 0.3 *vs* SPO 0.4	0.07	**0.015**	0.95	1
SPO 0.3 *vs* SPO$^+$	0.20	**0.020**	**0.00006**	**0.0005**
SPO 0.4 *vs* SPO$^+$	0.56	0.97	**0.00009**	**0.0014**

Table 10.1: p-values for pairwise comparison of different intensification mechanisms for optimizing CMA-ES performance on a number of instances. These p-values correspond to the data in Figure 10.5.

We denote as SPO$^+$ the variant of SPO that uses a random LHD, log-transformed cost statistics, expected improvement criterion $E[I^2]$, and the new intensification criterion just described. We compared SPO 0.3, SPO 0.4, and SPO$^+$—all based on a random LHD and log-transformed data—for our CMA-ES configuration scenarios and summarize the results in Figure 10.5 and Table 10.1. For configuration scenario `CMAES-ACKLEY`, SPO 0.4 performed best on average, but only insignificantly better than SPO$^+$, one of whose runs performed quite poorly. For scenario `CMAES-SPHERE`, on average SPO$^+$ performed insignificantly better than the other SPO variants, with better median performance and no poor outliers among its 25 runs. For the other two configuration scenarios, SPO$^+$ performed both significantly and substantially better than either SPO 0.3 or SPO 0.4, finding parameter configurations that led to CMA-ES performance orders of magnitude better than those obtained from SPO 0.3.

More importantly, as can be seen in Figure 10.6, over the course of the optimization process, SPO$^+$ showed much less variation in the quality of the incumbent parameter configuration than did the other SPO variants. This was the case even for scenario `CMAES-ACKLEY`, where SPO$^+$ did not perform best on average at the very end of its trajectory, and can also be seen for `CMAES-GRIEWANGK` and `CMAES-RASTRIGIN` functions, where SPO$^+$ clearly produced the best results.

In Hutter et al. (2009a), we performed a more in-depth evaluation of these results. There, we found that the poor performance of some of the parameter configurations found by SPO variants 0.3 and 0.4 was due to a small percentage of target runs with extremely poor performance. The solution cost distributions of CMA-ES were often multimodal, with orders of



(a) CMAES-SPHERE (b) CMAES-ACKLEY

(c) CMAES-GRIEWANGK (d) CMAES-RASTRIGIN

Figure 10.6: Comparison of the intensification mechanisms in SPO 0.3, SPO 0.4, and SPO$^+$. We show performance p_k (mean solution quality CMA-ES achieved in 100 test runs using the procedure's chosen parameter configurations) as a function of the number of target algorithm runs, k, the method is allowed. We plot means of p_k across 25 repetitions of each algorithm configuration procedure.

	Sphere	Ackley	Griewangk	Rastrigin
$E[I]$ *vs* $E[I^2]$	0.29	0.55	**0.016**	0.90
$E[I]$ *vs* $E[I_{\text{exp}}]$	0.63	0.25	0.11	**0.030**
$E[I^2]$ *vs* $E[I_{\text{exp}}]$	0.54	0.32	0.77	0.38

Table 10.2: p-values for pairwise comparison of different expected improvement criteria for optimizing CMA-ES performance on a number of instances. These p-values correspond to the data in Figure 10.7.

magnitude differences between the qualities of modes. Intuitively, in such noisy domains, SPO$^+$'s mechanism of only changing incumbents when enough evidence is gathered for the new candidate is particularly important.

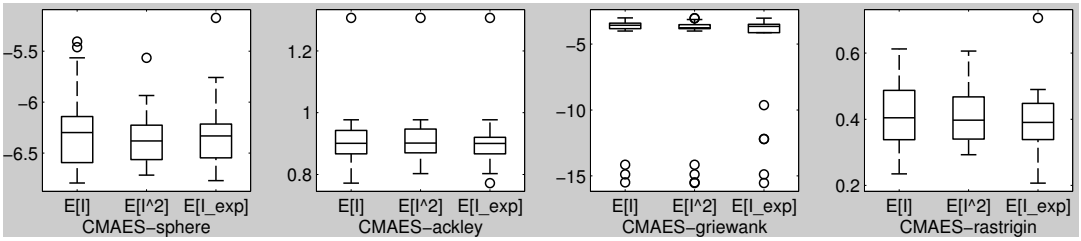


Figure 10.7: Comparison of three expected improvement criteria in SPO$^+$. We show boxplots of performance p_{1000} (mean solution quality CMA-ES achieved in 100 test runs using SPO$^+$'s chosen parameter configurations) achieved in the 25 runs of each tuner for each test case.

10.3.2 Expected Improvement Criterion

In sequential model-based optimization, parameter configurations to be investigated are selected based on an expected improvement criterion (EIC). This aims to address the exploration/exploitation tradeoff faced when deciding whether to learn more about new, unknown parts of the parameter space, or to intensify the search locally in the best known region. We briefly summarize two common versions of the EIC, and then describe a novel variation that we also investigated.

The classic expected improvement criterion (see, *e.g.*, Jones et al., 1998) is defined as follows. Given a deterministic function f and the minimal value $f_{\min}$ seen so far, the improvement at a new parameter configuration $\boldsymbol{\theta}$ is defined as

$$I(\boldsymbol{\theta}) := \max\{0, f_{\min} - f(\boldsymbol{\theta})\}. \tag{10.2}$$

Of course, this quantity cannot be computed, since $f(\boldsymbol{\theta})$ is unknown. We therefore compute the expected improvement, $E[I(\boldsymbol{\theta})]$. To do so, we require a probabilistic model of f, in our case the Gaussian process model. Let $\mu_{\boldsymbol{\theta}} = E[f(\boldsymbol{\theta})]$ be the mean and $\sigma^2_{\boldsymbol{\theta}}$ be the variance predicted by our model, and define $u = \frac{f_{\min} - \mu_{\boldsymbol{\theta}}}{\sigma_{\boldsymbol{\theta}}}$. Then one can show that $E[I(\boldsymbol{\theta})]$ has the following closed-form expression:

$$E[I(\boldsymbol{\theta})] = \sigma_{\boldsymbol{\theta}} \cdot [u \cdot \Phi(u) + \varphi(u)], \tag{10.3}$$

where φ and Φ denote the probability density function and cumulative distribution function of a standard normal distribution, respectively.

A generalized expected improvement criterion was introduced by Schonlau et al. (1998), who considered the quantity

$$I^g(\boldsymbol{\theta}) := \max\{0, [f_{\min} - f(\boldsymbol{\theta})]^g\} \tag{10.4}$$

for $g \in \{0, 1, 2, 3, \ldots\}$, with larger g encouraging more exploration. The value $g = 1$ corresponds to the classic EIC. SPO uses $g = 2$, which takes into account the uncertainty in our estimate of $I(\boldsymbol{\theta})$ since $E[I^2(\boldsymbol{\theta})] = (E[I(\boldsymbol{\theta})])^2 + \text{Var}(I(\boldsymbol{\theta}))$ and can be computed by the

closed-form formula

$$E[I^2(\boldsymbol{\theta})] = \sigma^2_{\boldsymbol{\theta}} \cdot [(u^2+1) \cdot \Phi(u) + u \cdot \varphi(u)]. \tag{10.5}$$

One issue that seems to have been overlooked in previous work is the interaction between log transformations of the data and the EIC. When we use a log transformation, we do so in order to increase predictive accuracy, yet our loss function continues to be defined in terms of the untransformed data (*e.g.*, actual runtimes). Hence we should optimize the criterion

$$I_{\text{exp}}(\boldsymbol{\theta}) := \max\{0, f_{\min} - e^{h(\boldsymbol{\theta})}\}, \tag{10.6}$$

where $h(\cdot)$ predicts log performance and $f_{\min}$ is the untransformed best known function value.

Let $v := \frac{\ln(f_{\min}) - \mu_{\boldsymbol{\theta}}}{\sigma_{\boldsymbol{\theta}}}$. Then, we have the following closed-form expression:

$$E[I_{\text{exp}}(\boldsymbol{\theta})] \quad = \quad f_{\min}\Phi(v) - e^{\frac{1}{2}\sigma^2_{\boldsymbol{\theta}} + \mu_{\boldsymbol{\theta}}} \cdot \Phi(v - \sigma_{\boldsymbol{\theta}}). \tag{10.7}$$

This closed-form expression can be derived as follows. Let X denote a random variable distributed according to a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$. Then,

$$\begin{aligned}
& E[\max(0, f_{\min} - \exp(X))] \\
= & \int_{-\infty}^{\infty} \max(0, f_{\min} - \exp(x)) p(x) dx \\
= & \int_{-\infty}^{\ln(f_{\min})} (f_{\min} - \exp(x)) \frac{1}{\sigma} \varphi(\frac{x-\mu}{\sigma}) dx \\
= & f_{\min}\Phi(\frac{\ln(f_{\min}) - \mu}{\sigma}) - \int_{-\infty}^{\frac{\ln(f_{\min})-\mu}{\sigma}} \exp[x\sigma + \mu] \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{1}{2}x^2\right] dx \\
= & f_{\min}\Phi(\frac{\ln(f_{\min}) - \mu}{\sigma}) - \int_{-\infty}^{\frac{\ln(f_{\min})-\mu}{\sigma}} \exp[\frac{1}{2}\sigma^2 + \mu] \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{1}{2}(x-\sigma)^2\right] dx \\
= & f_{\min}\Phi(\frac{\ln(f_{\min}) - \mu}{\sigma}) - \exp[\frac{1}{2}\sigma^2 + \mu]\Phi(\frac{\ln(f_{\min}) - \mu}{\sigma} - \sigma).
\end{aligned}$$

In Figure 10.7 and Table 10.2, we experimentally compare variants of SPO$^+$ using each of these three EI criteria on the four CMA-ES configuration scenarios, based (as before) on a random LHD and log-transformed data. Overall, the differences are small. On average, $E[I^2]$ yielded the best results for test case CMAES-sphere and our new criterion $E[I_{\text{exp}}]$ performed best in the remaining cases. Though not visually obvious from the boxplots, two of the 12 pairwise differences were statistically significant as judged by a Max-Wilcoxon test.

10.3.3 Overall Evaluation

In Sections 10.3.1 and 10.3.2, we fixed the design choices of using log transformations and initial designs based on random LHDs. Now, we revisit these choices: using our new SPO$^+$ intensification criterion and expected improvement criterion $E[I^2]$, we studied how much the

Procedure	SAPS median runtime [search steps]
SAPS default from Hutter et al. (2002)	$85.5 \cdot 10^3$
CALIBRA(100) from Hutter et al. (2007b)	$10.7 \cdot 10^3 \pm 1.1 \cdot 10^3$
BasicILS(100) from Hutter et al. (2007b)	$10.9 \cdot 10^3 \pm 0.6 \cdot 10^3$
FocusedILS from Hutter et al. (2007b)	$10.6 \cdot 10^3 \pm 0.5 \cdot 10^3$
SPO 0.3	$18.3 \cdot 10^3 \pm 13.7 \cdot 10^3$
SPO 0.4	$10.4 \cdot 10^3 \pm 0.7 \cdot 10^3$
SPO$^+$	$\mathbf{10.0 \cdot 10^3 \pm 0.4 \cdot 10^3}$

Table 10.3: Performance comparison of various configuration procedures for optimizing SAPS on instance QWH. We report mean $\pm$ standard deviation of performance $p_{20\,000}$ (median search steps SAPS required on QWH in 1 000 test runs using the parameter configurations the method chose after 20 000 algorithm runs), across 25 runs of each method. Based on a Mann-Whitney U test, SPO$^+$ performed significantly better than CALIBRA, BasicILS, FocusedILS, and SPO 0.3 with p-values 0.015, 0.0002, 0.0009, and $4 \cdot 10^{-9}$, respectively; the p-value for a comparison against SPO 0.4 was 0.06.

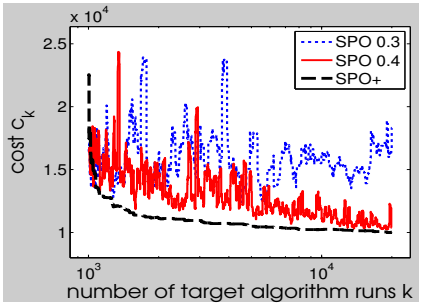


Figure 10.8: Comparison of SPO variants (all based on a random LHD and log-transformed data) for minimizing SAPS median runtime on instance QWH. We plot the performance p_k of each method (median search steps SAPS required on instance QWH in 1 000 test runs using the parameter configurations the method chose after k algorithm runs), as a function of the number of algorithm runs, k, it was allowed to perform. These values are averaged across 25 runs of each method.

final performance of SPO$^+$ changed when not using a log transform and when using different methods to create the initial design. Not surprisingly, no initial design led to significantly better final performance than any of the others. The result for the log transform was more surprising: although we saw in Section 10.2 that the log transform consistently improved predictive model performance, based on a Mann-Whitney U test it turned out to significantly improve *final* algorithm configuration performance only for CMA-ES-sphere.

Finally, using the single configuration scenario SAPS-QWH, we compared the performance of SPO 0.3, 0.4, and SPO$^+$ (all based on random LHDs and using log-transformed data) to the algorithm configuration procedures studied by Hutter et al. (2007b). We summarize the results in Table 10.3. SPO 0.3 performed worse than those methods, SPO 0.4 performed comparably,

and SPO$^+$ outperformed all methods with the exception of SPO 0.4 significantly albeit only by a small margin. Figure 10.8 illustrates the difference between SPO 0.3, SPO 0.4, and SPO$^+$ for this SAPS benchmark. Similar to what we observed for CMA-ES (Figure 10.6), SPO 0.3 and 0.4 changed their incumbents very frequently, with SPO 0.4 showing more robust behaviour than SPO 0.3, and SPO$^+$ in turn much more robust behaviour than SPO 0.4.

10.4 Reducing Overheads: SPO* and RANDOM*

In this and in the previous chapter, we measured the budget allocated to a configuration procedure as the number of target algorithms it is allowed to execute. This is the standard measure in blackbox function optimization. It is meaningful if all target algorithm runs take the same amount of time and clearly dominate any computational costs incurred by the configuration procedure. However, these conditions do not apply for the automated configuration of algorithms, where (1) some target algorithm runs can be much faster than others, and (2) in some configuration scenarios target algorithm runs are fast compared to the overhead's of the configuration procedure.

In fact, the experiment of running SPO$^+$ on SAPS-QWH described in Section 10.3.3 required about 10 CPU hours to perform its allowed 20 000 target algorithms runs. Only about 10 minutes of this time was spent actually running SAPS, for an average runtime of 30 milliseconds per target algorithm run. (Poor parameter configurations led to much larger runtimes, but in scenario SAPS-QWH, runs were cut off after one second.) Thus, in that case the configuration procedure's overhead was a factor of 60.

As discussed in Section 3.6.4, there are two types of overhead: implementation-specific overhead that could be reduced with better-engineered code, and overhead due to the computational complexity of building models and optimizing EIC. We do not count the first type of overhead since it could be substantially reduced in a straightforward manner. The second type of overhead is more fundamental and primarily occurs in model-based optimization, which is computationally more expensive than the model-free approaches discussed in Part III of this thesis. We thus *do* count that overhead.

We now discuss a modification of SPO$^+$ to limit that computational overhead. For this purpose, we modify the general SMBO framework to keep track of the overhead time spent learning the model, t_{model}, and optimizing EIC, t_{eic}—see Algorithm Framework 10.3. The new mechanism for selecting configurations is detailed in Procedure 10.4. Note that this procedure interleaves random configurations with configurations selected by optimizing EIC. Since each iteration is now time-bounded rather than bounded by the number of configurations, we eliminate the previous bound on parameter configurations, m (compare Procedure 9.8 on page 140). In Procedure 10.5, we introduce a modified intensification procedure that executes runs of the target algorithm for at least time $t_{overhead} = t_{model} + t_{ei}$. Specifically, the outer **for**-loop now runs until the time spent executing the target algorithm in this iteration exceeds $t_{overhead}$, but at least for $p + 2$ iterations—the p old configurations, one EIC configuration and one random configuration. Due to the extra random configurations evaluated in each iteration, we no longer rely on the Latin hypercube design for initialization. Instead, we initialize the model based on a single run using the algorithm's default configuration. These changes result

Algorithm Framework 10.3: Sequential Model-Based Optimization With a Time Budget.

Input : Target algorithm A
Output: Incumbent parameter configuration $\boldsymbol{\theta}_{inc}$

$[\mathbf{R}, \boldsymbol{\theta}_{inc}] \leftarrow$ Initialize()
$\mathcal{M} \leftarrow \emptyset$
repeat
  $[\mathcal{M}, \boldsymbol{\theta}_{inc}, t_{model}] \leftarrow$ FitModel($\mathcal{M}$, $\mathbf{R}$, $\boldsymbol{\theta}_{inc}$)
  $[\vec{\boldsymbol{\Theta}}_{new}, t_{ei}] \leftarrow$ SelectNewConfigurations($\mathcal{M}$, $\boldsymbol{\theta}_{inc}$, $\mathbf{R}$)
  $[\mathbf{R}, \boldsymbol{\theta}_{inc}] \leftarrow$ Intensify($\vec{\boldsymbol{\Theta}}_{new}$, $\boldsymbol{\theta}_{inc}$, $\mathcal{M}$, $\mathbf{R}$, $t_{model} + t_{ei}$)
until *total time budget for configuration exhausted*
return $\boldsymbol{\theta}_{inc}$

Procedure 10.4: SelectNewParameterConfigurations($\mathcal{M}, \boldsymbol{\theta}_{inc}, \mathbf{R}$) in SPO*
p is a parameter of configurators using this method; we set it to $p = 5$ in SPO*.

Input : Model, $\mathcal{M}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$; sequence of target algorithm runs, $\mathbf{R}$
Output: Sequence of parameter configurations to evaluate, $\vec{\boldsymbol{\Theta}}_{new}$

// ===== Select p previously-used parameter configurations
$\boldsymbol{\Theta}_{seen} \leftarrow \bigcup\{\mathbf{R}[1].\boldsymbol{\theta}, \ldots, \mathbf{R}[n].\boldsymbol{\theta}\}$
$\vec{\boldsymbol{\Theta}}_{new} \leftarrow p$ elements $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{seen}$, drawn with probability $\propto 1/\hat{c}(\boldsymbol{\theta})$ and without repetitions
// ===== Select parameter configurations with expected improvement
$\boldsymbol{\Theta}_{rand} \leftarrow$ set of 10 000 elements drawn uniformly at random from $\boldsymbol{\Theta}$
for $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{rand}$ **do**
  $[\mu_{\boldsymbol{\theta}}, \sigma^2_{\boldsymbol{\theta}}] \leftarrow$ Predict($\mathcal{M}$, $\boldsymbol{\theta}$)
  $EI(\boldsymbol{\theta}) \leftarrow$ Compute EIC $E[I_{exp}(\boldsymbol{\theta})]$ (see Section 10.3.2) given $\mu_{\boldsymbol{\theta}}$ and $\sigma^2_{\boldsymbol{\theta}}$
Let t_{ei} denote the total time spent computing expected improvement
Let $\vec{\boldsymbol{\Theta}}_{ei}$ be a list of all $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{rand}$, sorted by decreasing $EI(\boldsymbol{\theta})$
// ===== Interleave configurations with high EI, and random configurations
for $i = 1, \ldots, 10\,000$ **do**
  Append $\vec{\boldsymbol{\Theta}}_{ei}[i]$ to $\vec{\boldsymbol{\Theta}}_{new}$
  Draw a parameter configuration $\boldsymbol{\theta}$ uniformly at random from $\boldsymbol{\Theta}$ and append it to $\vec{\boldsymbol{\Theta}}_{new}$
return $[\vec{\boldsymbol{\Theta}}_{new}, t_{ei}]$

in a procedure we call SPO*. One final difference between SPO$^+$ and SPO* is the expected improvement criterion (EIC) used. While in SPO$^+$, we use the same EIC previous SPO variants use, $E[I^2]$, in SPO*, we use our new EIC $E[I_{exp}(\boldsymbol{\theta})]$ (see Section 10.3.2). This did not lead to significant differences in our experiments, but $E[I_{exp}(\boldsymbol{\theta})]$ is theoretically better justified for the log transformations we use.

In Figure 10.9, we show that in configuration scenario SAPS-QWH this new version, SPO*, finds good configurations substantially faster than SPO$^+$. We show the performance of SPO$^+$ (exactly as used in Section 10.3.3: with an LHD of 500 points and 2 repetitions each), and two versions of SPO*: our normal variant discussed above (using a single run of the

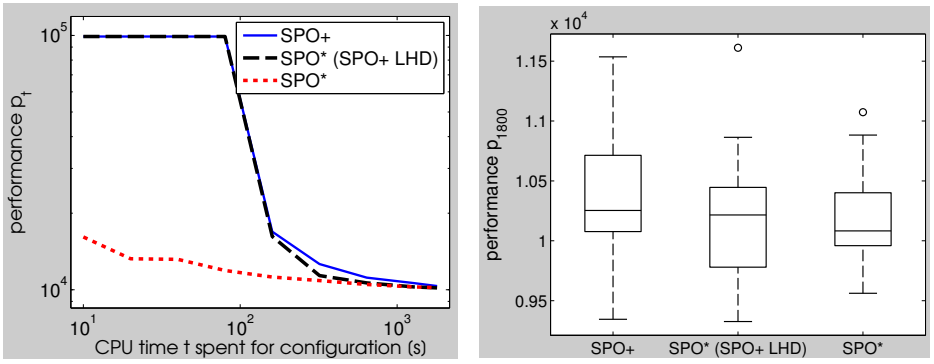
Procedure 10.5: Intensify($\vec{\boldsymbol{\Theta}}_{new}$, $\boldsymbol{\theta}_{inc}$, $\mathcal{M}$, $\mathbf{R}$, p, $t_{overhead}$) in SPO*
p is a parameter of configurators using this method; we set it to $p = 5$ in SPO*.
Recall that $N(\boldsymbol{\theta})$ denotes the number of algorithm runs which have been performed for $\boldsymbol{\theta}$, *i.e.*, the length of $\mathbf{R}_{\boldsymbol{\theta}}$. The maximal number of runs to perform with a configuration, maxR, is a parameter of configurators using this procedure; in all our experiments, we set it to 2 000.

Input : Sequence of parameter configurations to evaluate, $\vec{\boldsymbol{\Theta}}_{new}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$; model, $\mathcal{M}$; sequence of target algorithm runs, $\mathbf{R}$
Output: Sequence of target algorithm runs, $\mathbf{R}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$

for $i = 1, \ldots,$ *length*($\vec{\boldsymbol{\Theta}}_{new}$) **do**
  if $i \geq p + 2$ **and** *time spent in this call to this procedure exceeds* $t_{overhead}$ **then**
    break
  $\boldsymbol{\theta} \leftarrow \vec{\boldsymbol{\Theta}}_{new}[i]$
  $r \leftarrow 1$
  $\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}$, 1)
  numBonus $\leftarrow 1$
  if $N(\boldsymbol{\theta}) > N(\boldsymbol{\theta}_{inc})$ **then**
    $\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}_{inc}$, 1)
    numBonus $\leftarrow 0$
  while *true* **do**
    if $\hat{c}(\boldsymbol{\theta}) > \hat{c}(\boldsymbol{\theta}_{inc})$ **then**
      // ===== Reject challenger, perform bonus runs for $\boldsymbol{\theta}_{inc}$
      $\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}_{inc}$, min(numBonus, maxR $- N(\boldsymbol{\theta}_{inc})$))
      break
    if $N(\boldsymbol{\theta}) \geq N(\boldsymbol{\theta}_{inc})$ **then**
      // ===== Challenger becomes incumbent
      $\boldsymbol{\theta}_{inc} \leftarrow \boldsymbol{\theta}$
      break
    if *total time budget for configuration exhausted* **then**
      return $[\boldsymbol{\theta}_{inc}, \mathbf{R}]$
    $r \leftarrow \min(r \cdot 2, N(\boldsymbol{\theta}_{inc}) - N(\boldsymbol{\theta}))$
    $\mathbf{R} \leftarrow$ ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}$, r)
    numBonus $\leftarrow$ numBonus $+r$
return $[\mathbf{R}, \boldsymbol{\theta}_{inc}]$

default for initialization), and a variant using the same LHD as SPO$^+$. Comparing the two configuration procedures with the same LHD, after the runs for the initial design are finished, SPO* performed better than SPO$^+$, leading to significantly improved performance (judged by a Mann Whitney U test). Without the overhead time of the initial design, our regular SPO* version started finding good configurations much faster. After one hour, it was, however, only insignificantly better than the SPO* version which used the larger LHD.

Interestingly, the new intensification criterion in SPO* did not only help to improve the model-based optimization approach, it also transformed pure random search into a competitive configuration procedure. Specifically, we studied a configuration procedure—dubbed RANDOM*—that only differed from SPO* in Procedures FitModel and SelectNewConfigurations: the former procedure is empty in RANDOM* and the latter returns a list of 10 000 parameter



(a) Performance over time

(b) End result after one hour, p_{3600}

Figure 10.9: Comparison of SPO$^+$ and SPO*. We carried out 25 runs of each configurator and show performance p_t (median number of SAPS search steps using the procedure's chosen parameter configurations). 'SPO*(SPO$^+$ LHD)' denotes a version of SPO* that uses the same LHD as the SPO$^+$ variant shown ($500 \cdot 2$ data points). Subplot (a): p_t as a function of the time, t, the method is allowed for configuration (mean over 25 runs). Subplot (b): boxplot of performance values p_{3600} achieved in the 25 runs.

configurations chosen uniformly at random. This very simple configuration procedure turned out to yield somewhat *better* performance than SPO* on scenario SAPS-QWH, albeit not significantly better (we give detailed experimental results in the next chapter, in Table 11.1 on page 190). Our conclusion is therefore that the performance improvements we observed are largely due to our better intensification criteria as opposed to the model underlying the various SPO variants. In the next chapter, we will study whether this underlying model can be improved to construct a more effective configuration procedure.

10.5 Chapter Summary

In this chapter we evaluated and improved important components of the sequential parameter optimization (SPO) framework. In our evaluation of the initial model, we found that log transformations of the empirical cost statistics substantially improved predictive quality, while various methods for constructing the initial design only led to insignificantly different results. We evaluated two components of the sequential experimental design, introducing a new expected improvement criterion (EIC) to deal with log transformations, and a new intensification mechanism. Though theoretically more principled, our new EIC did not consistently improve performance in practice. In contrast, our new intensification mechanism substantially improved robustness and led us to a new configuration procedure, SPO$^+$ (which also uses a log-transformation of cost statistics). On configuration scenario SAPS-QWH, SPO$^+$ was competitive with state-of-the-art configuration procedures. In particular, it significantly—but not substantially—outperformed FOCUSEDILS, PARAMILS, and CALIBRA.

Finally, we bounded SPO$^+$'s computational overhead to arrive at configuration procedure SPO* which found good configurations much faster than SPO$^+$. SPO* interleaves randomly-sampled parameter configurations with the ones selected based on the response surface model and employs a further-improved intensification mechanism. Using *only* randomly-sampled configurations with this new intensification criterion led us to a new variant of RANDOM-SEARCH—dubbed RANDOM*—that turned out to perform competitively with SPO*. Thus, we concluded that our performance improvements were largely due to the improved intensification criteria, and not to the usefulness of the model per se. In the next chapter, we will study whether better models can improve this performance.

Chapter 11

Improvements II: Different Models

Prediction is very difficult, especially about the future.
—Niels Bohr, Danish atomic physicist

In this chapter, we study the use of alternative response surface models in the sequential model-based optimization (SMBO) framework. While the SPO variants we studied in the previous chapters employ a "vanilla" noise-free Gaussian process (GP) model, called the DACE model, we use a modified version of random forests (RFs) and the existing projected process (PP) approximation of Gaussian processes (GPs). We study the performance of these and the DACE model, analyzing model performance both qualitatively and quantitatively. Since the configuration procedures we discuss in this chapter are still limited to optimizing continuous algorithm parameters for single problem instances, we study the seven SINGLEINSTCONT configuration scenarios defined in Section 3.5.2, in which the objective is to minimize SAPS runtime on single instances.

For these configuration scenarios, in which the objective is to tune numerical parameters on single instances, we demonstrate that both RF and approximate GP models yield substantially better predictions than the DACE model, and take orders of magnitude less time to do so. We introduce novel configuration procedures based on our new models and demonstrate that the version based on approximate GP models significantly outperforms SPO* and RANDOM* (both defined in Section 10.4 on page 163). It also clearly outperforms FOCUSEDILS, which is handicapped in this continuous optimization task by searching a restricted discretized subspace.

11.1 Random Forests

Random forests (Breiman, 2001) are a flexible tool for regression and classification. We chose to study them (in addition to Gaussian process models) since we are interested in building models for algorithms with categorical parameters. Random forest (RF) models are well known to handle such data well. For the time being, however, we only consider continuous inputs, deferring categorical variables to the next chapter. Here, we describe the standard RF

model framework for regression and also introduce some novel elements. We largely follow the presentation of Hastie et al. (2009).

11.1.1 Regression Trees

Regression trees are simple tree-based predictors of continuous response variables. They partition the input space (in our case, the parameter configuration space), $\boldsymbol{\Theta}$, into disjoint regions $R_1, \ldots, R_M$ and use a simple model—such as a constant—for prediction in each region. Here, we use a constant c_m in region R_m. This leads to the following prediction for an input point $\boldsymbol{\theta} \in \boldsymbol{\Theta}$:

$$\hat{\mu}(\boldsymbol{\theta}) = \sum_{m=1}^{M} c_m \delta(\boldsymbol{\theta} \in R_m),$$

where δ is the Kronecker delta function. Note that since the regions R_m partition the input space, this sum will always only have one nonzero element.

We denote the set of training data points as $\mathcal{D} = \{(\boldsymbol{\theta}_i, o_i)\}_{i=1}^{n}$, and the subset of data points in region R_m as $\mathcal{D}_m$. Under the standard squared error loss function $\sum_{i=1}^{n} (o_i - \hat{\mu}(\boldsymbol{\theta}_i))^2$, the optimal choice of constant c_m in region R_m is the sample mean of the data points that fall into the region,

$$c_m = \frac{1}{|\mathcal{D}_m|} \sum_{\boldsymbol{\theta}_i \in R_m} o_i. \tag{11.1}$$

Finding the optimal regression tree (*i.e.*, the optimal partition of input space) is typically computationally infeasible. Thus, we resort to a greedy recursive tree construction mechanism. This process starts at the root of the tree with all available data $\langle (\boldsymbol{\theta}_1, o_1), \ldots, (\boldsymbol{\theta}_n, o_n) \rangle$. At each node, we consider binary partitionings of the node's data along a *split variable* j and *split points* s: data point $\boldsymbol{\theta}_i$ is put into region $R_1(j, s)$ if $\theta_{i,j} < s$ and into region $R_2(j, s)$ otherwise, where $\theta_{i,j}$ denotes the value of the j-th element of input $\boldsymbol{\theta}_i$. At each node, split variable j and split point s are chosen to minimize some loss function $l(j, s)$, in our case the sum of squared differences to the regions' means,

$$l(j, s) = \left[\sum_{\boldsymbol{\theta}_i \in R_1(j,s)} (o_i - c_1)^2 + \sum_{\boldsymbol{\theta}_i \in R_2(j,s)} (o_i - c_2)^2 \right], \tag{11.2}$$

where c_1 and c_2 are chosen according to Equation (11.1) as the sample mean in regions $R_1(j, s)$ and $R_2(j, s)$, respectively.

Starting with the complete training data at the root, the tree building mechanism finds the best split at each node and partitions the data points into the two resulting regions of the node's two children. It then recursively applies the same process to the children. The process terminates if no more splits are possible (all training data points in a node share the same $\boldsymbol{\theta}$ values) or if there are fewer than n_{min} data points left in a node, where n_{min} is an algorithm parameter that aims to prevent over-fitting. There also exist methods for pruning regression trees by removing useless branches (which we do not use). For these, we refer the interested

Procedure 11.1: Fit Random Forest($\mathcal{D}$, B, *perc*, n_{min})

Input : Data $\mathcal{D} = \{(\boldsymbol{\theta}_i, o_i)\}_{i=1}^n$ (where each $\boldsymbol{\theta}_i$ has d elements); number of trees, B; percentage of variables to consider at each split, *perc*; minimal size for split, n_{min}

Output : A random forest (set of trees), $\{T_1, \ldots, T_B\}$

for $b = 1, \ldots, B$ **do**
- (a) Draw a bootstrap sample s of n elements from $\{1, \ldots, n\}$, with repetitions, and let $\mathcal{D}_b = \{(\boldsymbol{\theta}_{s(i)}, o_{s(i)})\}_{i=1}^n$
- (b) Grow a regression tree T_b to data $\mathcal{D}_b$, by recursively splitting nodes as long as they contain at least n_{min} data points:
- **begin**
  - Draw random subset, $\mathcal{V} \subseteq \{1, \ldots, d\}$, of cardinality $\max(1, \lfloor \textit{perc} \cdot d \rfloor)$
  - Select split variable/point combination (j, s) with $j \in \mathcal{V}$ to min. Eq. (11.2)
  - Split node into 2 child nodes with data $R_1(j, s)$ and $R_2(j, s)$, respectively
  - At each leaf node m with associated region R_m, store the response values $\mathcal{O}_m$ of the training data points in that region, $\mathcal{O}_m = \{o_{s(i)} | \boldsymbol{\theta}_{s(i)} \in R_m\}$
- **end**

RF $\leftarrow \{T_1, \ldots, T_B\}$

return RF

reader to the book by Hastie et al. (2009).

In order to predict the response value at a new input, $\boldsymbol{\theta}_i$, we *propagate* $\boldsymbol{\theta}$ *down the tree*, that is, at each node with split variable j and split point s, we continue to the left child node if $\theta_{i,j} < s$ and to the right child node otherwise. The prediction for $\boldsymbol{\theta}_i$ is the constant c_m in the leaf this process leads to.

11.1.2 Standard Random Forest Framework

If grown to sufficient depths, regression trees are very flexible predictors. They are able to capture very complex interactions and thus typically have low bias. However, this great flexibility also leads to large variance: small changes in the data can lead to a dramatically different tree. One common technique to reduce variance of low-bias predictors is *bagging* (Hastie et al., 2009). This stands for *b*ootstrap *agg*regation and works by fitting multiple so-called base learners to B random bootstrap samples of the data. The predictions of these B learners are then combined to form an overall prediction.

Procedure 11.1, adapted from Hastie et al. (2009), shows how bagging is used to build a random forest. Note that there are two tuning parameters: the percentage of variables to consider at each split point, *perc*, and the parameter determining when to stop splitting a node, n_{min}. The number of trees is of a slightly different nature: the more trees, the better the prediction, but the more costly the procedure. Breiman (2001) suggests to use small random subsets, $\mathcal{V}$, of variables for classification. However, for regression he states—confirming our own experience—that $\mathcal{V}$ should be much larger. Throughout, we use the default percentage of *perc* $= 5/6$. Later on in this chapter (in Section 11.4.2), we compare versions with two values of n_{min}, 1 and 10, and eventually choose $n_{min} = 10$.

Prediction for a new input $\boldsymbol{\theta}$ in the standard random forest framework is trivial: predict

the response for $\boldsymbol{\theta}$ with each tree and average the predictions.

11.1.3 Uncertainty Estimates Across Trees

We introduce a simple method for quantifying predictive uncertainty in random forests. Even though straightforward, we are not aware of any previous publication of this idea.[1]

Intuitively, if the individual tree predictions $y_1, \ldots, y_B$ tend to agree for a new data point, we are more certain about the prediction than if they disagree vastly. Following this intuition, we compute the uncertainty estimate for a new data point $\boldsymbol{\theta}$ as the empirical variance of the individual tree predictions $(y_1, \ldots, y_B)$.

Our implementation of random forests also chooses split points in a non-standard way in order to improve uncertainty estimates. We still select a split point that minimizes the loss in Equation (11.2). However, note that if this loss is minimized by splitting on variable j between the values of $\boldsymbol{\theta}_{kj}$ and $\boldsymbol{\theta}_{lj}$, then we are still free to choose the exact split value anywhere in the interval $(\boldsymbol{\theta}_{kj}, \boldsymbol{\theta}_{lj})$ (assuming without loss of generality that $\boldsymbol{\theta}_{kj} < \boldsymbol{\theta}_{lj}$). In typical implementations, the split value is chosen as the midpoint between $\boldsymbol{\theta}_{kj}$ and $\boldsymbol{\theta}_{lj}$. Here, instead, we draw it uniformly at random from the interval $(\boldsymbol{\theta}_{kj}, \boldsymbol{\theta}_{lj})$. In the limit of an infinite number of trees, this leads to a linear interpolation of the training data instead of a partition into regions of constant prediction—see Figure 11.1(b) for an example. Furthermore, it causes variance to vary smoothly and grow further away from data points.

11.1.4 Transformations of the Response Variable

Transformations often help to improve the model fit. We have already shown an example for this in Section 10.2.2. However, in Section 9.4 we have also shown an example where computing a cost statistic (the arithmetic mean) after a (log-)transformation effectively changes the cost statistic (in that case to the geometric mean). Random forests allow us to use transformations of the response variable without incurring this problem.

There are several ways in which we can integrate transformations of the response variable into random forests. Firstly, when we wish to model a cost based on a statistic, τ, and use a response transformation, $t(\cdot)$, we can make predictions for a region R_m as either

$$c_m = \hat{\tau}(\{t(o_i) | \boldsymbol{\theta}_i \in R_m\}), \tag{11.3}$$

or

$$c_m = t(\hat{\tau}(\{o_i | \boldsymbol{\theta}_i \in R_m\})). \tag{11.4}$$

The traditional approach of applying a transformation outside of the learner leads to the first alternative, which is prone to exactly the same problem we discussed above: this transformation implicitly changes the cost statistic being modelled (*e.g.*, from arithmetic to geometric mean). Here, we apply the second approach, which is not prone to this problem. Optimally, we would like to also substitute Equation (11.1) by Equation (11.4) to reflect this modified

[1] Meinshausen (2006) recently introduced quantile regression trees to predict quantiles of the predictive distribution for single responses. Our goal is slightly different: we are interested in an uncertainty of the *mean* response (or another user-defined cost statistic) at an input.

Algorithm 11.2: Prediction with Random Forest(RF, $\boldsymbol{\theta}$, t)

Input : Random Forest RF=$\{T_1, \ldots, T_B\}$; parameter configuration, $\boldsymbol{\theta}$; transformation t
Output : Predicted mean μ and variance σ^2 of cost measure $c(\boldsymbol{\theta})$.

for $b = 1, \ldots, B$ **do**
    Let R_m be the region of T_b containing $\boldsymbol{\theta}$, with stored set of response values $\mathcal{O}_m$
    $y_b \leftarrow t(\hat{\tau}(\{o_i | t(o_i) \in \mathcal{O}_m\}))$
$\mu \leftarrow \frac{1}{B} \sum_{b=1}^{B} y_b$
$\sigma^2 \leftarrow \frac{1}{B-1} \sum_{b=1}^{B} (y_b - \mu)^2$
return $[\mu, \sigma^2]$

criterion throughout the tree construction process. However, in order to use existing methods to efficiently compute the loss function $l(j, s)$ (see Equation (11.2)) in amortized $O(1)$ time, we computed the transformed response outside the learner and used the standard tree building procedure as described above, just as one would do in the traditional approach. Only at prediction time, we applied Equation (11.4) to form the prediction o_b of each tree T_b. (In future work, it would be interesting to study whether performance improves if Equation (11.1) is replaced by (11.4) throughout, and whether this can be achieved computationally efficiently.) We compute empirical mean and variance across the individual tree predictions; since in our case the predictions are log-transformed, the Gaussian distribution defined by this mean and variance corresponds to a log-Normal distribution in untransformed space. This is a much better choice than, for example, a Gaussian in untransformed space: log-Normal distributions have positive support and a wide spread, much like typical runtime distributions of randomized algorithms.

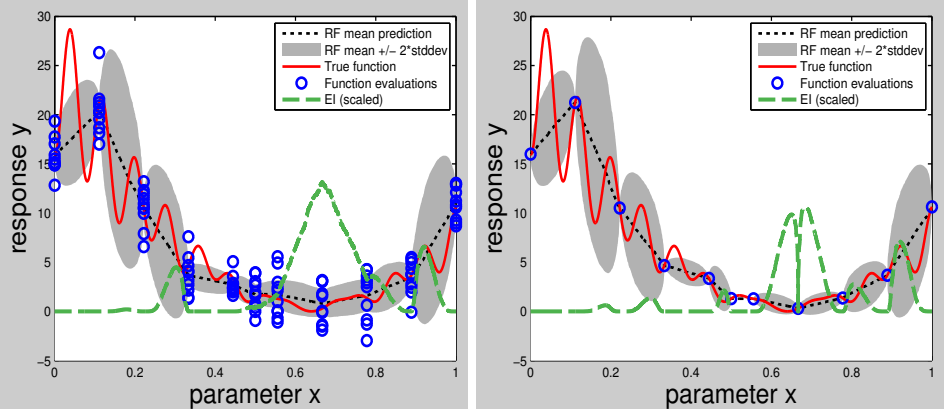
Procedure 11.2 gives pseudocode for prediction in our version of RFs. We use standard RF training as outlined in Procedure 11.1, with input data $\mathcal{D} = \{(\boldsymbol{\theta}_i, t(o_i))\}_{i=1}^{n}$, with t being $\log_{10}$.

11.2 Qualitative Comparison of Models

The models we discussed so far—based on standard Gaussian stochastic processes (GPs), noise-free GPs, and random forests—all have advantages and disadvantages. Here, we qualitatively compare model predictions and how the models integrate into the SMBO approach.

11.2.1 Qualitative Model Quality

Like GP models, our version of random forests (RFs) can be used to model noise-free and noisy functions. Figure 11.1 shows that our standard RF model yields meaningful predictions and uncertainty estimates for noisy functions—in particular, for the 1-d Levy function we already used for illustrations in Section 9.1. If we know that observations are noise-free it is desirable to yield perfect predictions on the training data. For example, the DACE model (a noise-free GP model) has this property: for previously-seen data points its predictive mean equals the true function value and the predictive variance is zero. While our standard RF model does not have this property, we can easily achieve it by skipping the bootstrap sampling



(a) RF fit to noisy data

(b) RF fit to noise-free data

Figure 11.1: Step 1 of SMBO based on RF models. For the noisy data in (a), we use our standard RF models. For the noise-free data in (b), we skip the bootstrap step to perfectly reproduce the training data. Note that in the noise-free case expected improvement is exactly zero at previously-observed data points.

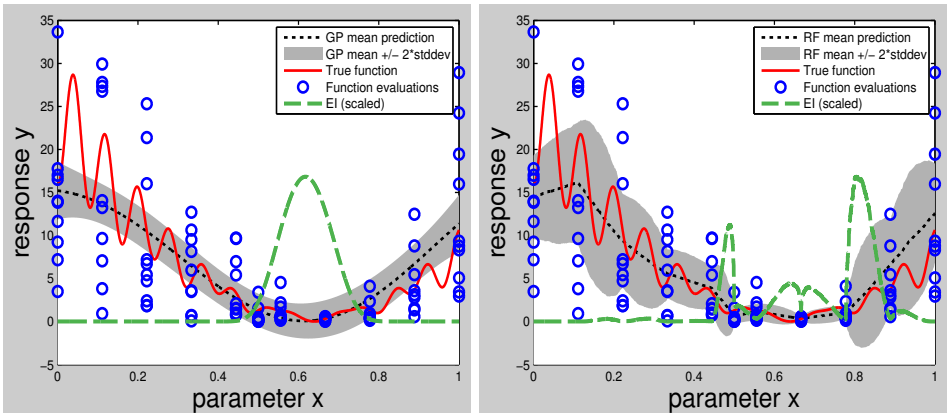
step—using the complete training data to build each tree—and splitting nodes until no more splits are possible ($n_{min} = 1$). Figure 11.1(b) illustrates this "noise-free" RF fit for our noise-free 1-d function. Note that for data points outside the training set each tree can still yield different predictions. This is due to the remaining randomness in the random selection of split variables and split points.

Proportional Noise

In algorithm configuration, the observation noise is rarely additive Gaussian. For example, algorithm runtime—the quantity we are primarily interested in minimizing—is strictly positive. Good approximations of its distribution can often be achieved with exponential distributions or generalizations thereof, such as the Weibull distribution (Hoos, 1999a). Here, we study how well GPs and RFs can fit data under such observation noise.

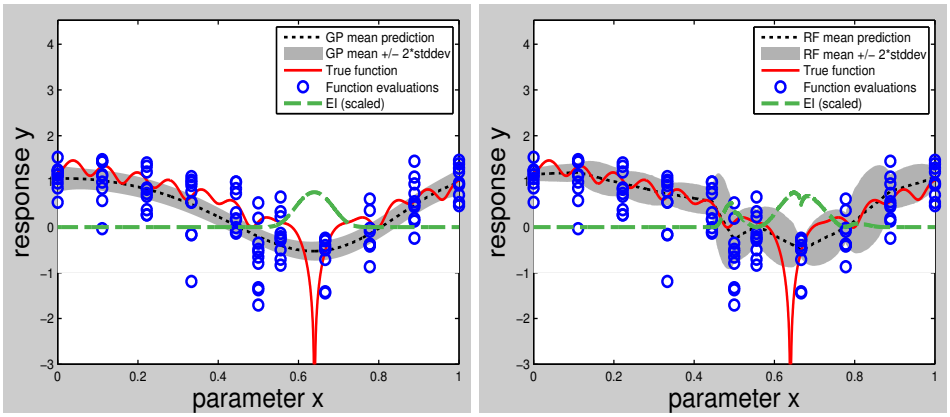
Figure 11.2 shows the drawback of a standard GP model for such non-stationary noise: its uncertainty estimates are too low in the high-noise region and too high in the low noise region.[2] In contrast, the RF (see Figure 11.2(b)) adapts to the data and yields better uncertainty estimates. For brevity, here and in the following we omit the fairly uninteresting plots for the DACE model. They were all qualitatively similar to Figure 9.2(b): its mean predictions perfectly tracked the sample average at the training data, and its predictive variance was low

[2]Non-stationary GP models could in principle be used instead. In particular, also in need of approximate GP models, we experimented with sparse online Gaussian processes (Csat and Opper, 2003) and sparse pseudo-input Gaussian processes (Snelson and Ghahramani, 2006). However, the former one was orders of magnitude slower than the standard implementation we ended up using (Rasmussen and Williams, 2006), and the latter was tied to a particular kernel function that we could not extend to categorical parameters (discussed in the next chapter).



(a) Standard GP fit (b) RF fit

Figure 11.2: Standard GP and RF model fits for proportional noise. For a true function value y, observations were drawn from an exponential distribution with mean y.

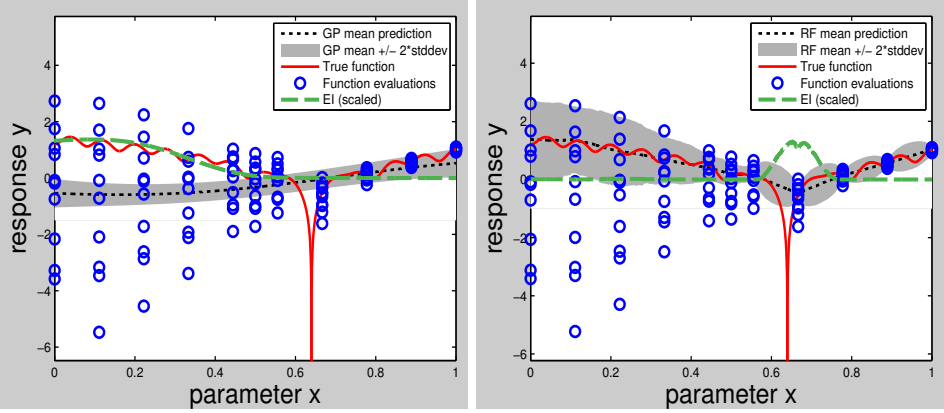


(a) Standard GP fit (b) RF fit

Figure 11.3: Standard GP and RF model fits in log space for proportional noise. The data is identical to that shown in Figure11.2 (but plotted on a logarithmic y-axis).

throughout (zero at the training data points).

A log transformation renders proportional noise stationary. As we show in Figure 11.3, such a transformation strongly improves the uncertainty estimates of the GP model. Note, however, that its mean predictions are somewhat below the true function. This is because when fit to log-transformed data, the standard GP model tracks the *geometric* mean of the observations instead of the arithmetic mean (see Section 9.4).



(a) Standard GP fit on log-transformed raw data

(b) RF fit in log domain

Figure 11.4: Standard GP and RF fits to log-transformed data where noise in this example is non-stationary and multiplicative. This is the same data as in Figure 9.3 on page 135. The standard GP model fits the mean of the logs, *i.e.*, the geometric mean. The RF model fits the true function in log space.

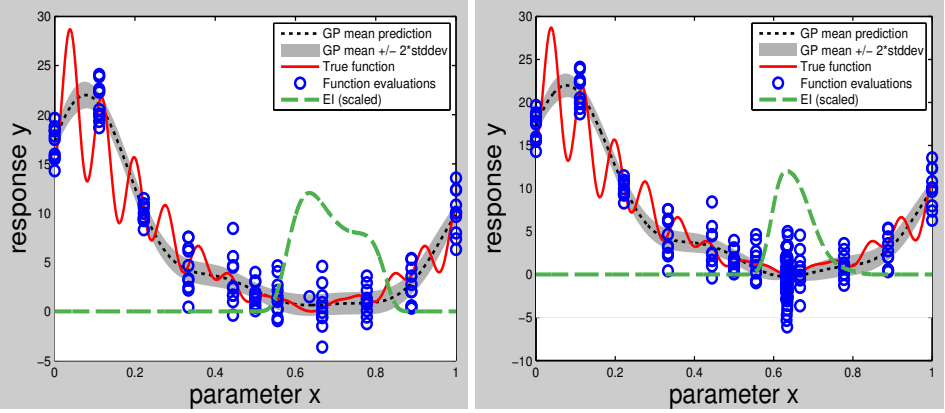
Non-Stationary Proportional Noise

As long as the noise (of the log-transformed response) is stationary fitting the geometric mean is not necessarily a problem. This is because for stationary noise, the minima of the geometric mean and the arithmetic mean coincide (in fact, *all* comparisons between the geometric means at two data points will yield the same result as a comparison of the corresponding arithmetic means).

However, if the noise is proportional and *non-stationary* (even after the log transform), SMBO based on a standard GP model fitted to log-transformed response values will fail to minimize the arithmetic mean. Figure 11.4 demonstrates that the GP model has two problems. First, fitting geometric instead of arithmetic mean, it predicts the function's minimum to be at the far left. Second, having to choose one global noise variance, it is much too confident in its predictions in the high-noise region on the left. In contrast, the RF avoids both of these problems: it tracks the true function and has higher uncertainty in high-noise regions.

11.2.2 Potential Failure Modes of SMBO Using Various Models

All of the models we introduced—standard GPs, noise-free GPs fit on cost statistics, and RFs—have potential qualitative failure modes. Here, we demonstrate these failure modes on our simple 1-d function.



(a) Standard GP fit, step 2

(b) Standard GP fit, step 100

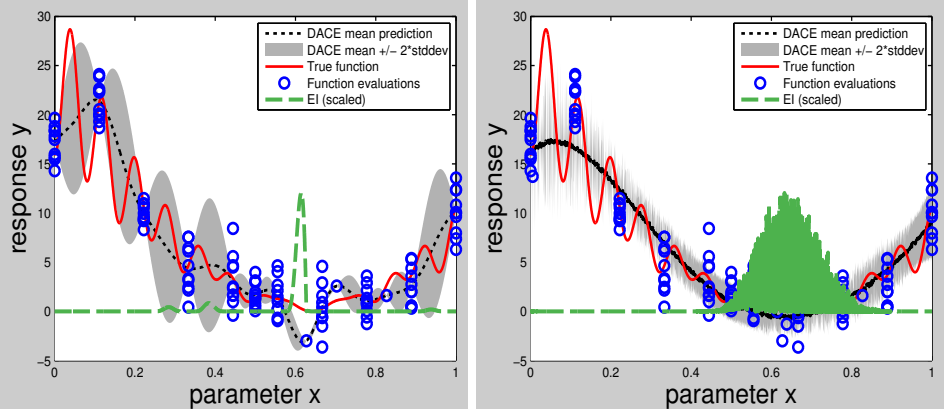
Figure 11.5: One failure mode of SMBO based on the standard GP model. Even for simple additive noise (same as in Figure 9.2(a)), the sequential optimization approach keeps revisiting the same small area. We show steps 2 and 100 of the sequential procedure. All queried points lie between 0.63 and 0.635; the optimum lies at 0.64.

Potential Failure Mode of SMBO based on Standard Gaussian Process

In Figure 11.5, we show a failure mode of SMBO based on a standard GP model. Observe that even in this simple case of additive Gaussian noise, the sequential optimization keeps revisiting the same point (the same very small region) over and over again, for at least 100 sequential steps. The problem here is that the GP model barely changes when adding additional observations for this point. Since the model does not change much it makes similar recommendations in the next step, and so on. Note that in this case, the search stagnates starting in step one of the procedure! The effect is even worse for proportional noise models, such as in Figure 11.2(a). There, the search stagnates as well, but at a worse point: due to the non-stationary noise the predicted minimum is further away from the true minimum.

Potential Failure Mode of SMBO based on the DACE Model

Fits of a noise-free GP model, such as the DACE model, to noisy data can be extremely sensitive; we demonstrate this in Figure 11.6. Since the DACE model cannot attribute the discrepancy in the cost statistics of two nearby data points to noise, it is forced to reduce its length scale λ (compare Section 9.2) in order to perfectly interpolate the cost statistics for all training data. However, note that despite the extremely sensitive model the sequential optimization process can still continue. In this case, the model still captures the overall trend in the data (with very large local fluctuations). Correspondingly, the expected improvement function yields meaningful values and—in contrast to the standard GP model—keeps exploring the space.



(a) Noise-free GP fit, step 4

(b) Noise-free GP fit, step 7

Figure 11.6: Failure model of SMBO based on DACE model. We show additional sequential optimization steps based on the DACE model (a noise-free GP model) for the same noise model as in Figure 9.2. In step 4, the length scale starts getting quite short, and by step 7 (and thereafter) it is extremely short. The sequential optimization process can still continue, albeit with an extremely noise model.

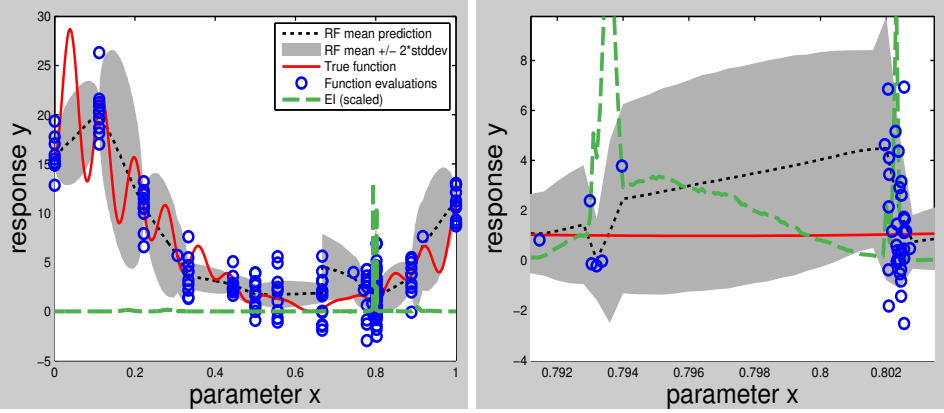
Potential Failure Mode of SMBO based on Random Forest

Finally, random forests have potential failure modes, too. For example, Figure 11.7 demonstrates a problem for the simple additive noise model. Figure 11.7(a) shows the progress of SMBO based on a random forest model at step 50. Note that the search has not explored many regions far away from the initial design points.[3] Figure 11.7(b) provides more detail for the area with two spikes in expected improvement. In both cases, the spike occurs in areas with low predictive mean and comparably high variance. These coincide with regions that contain many training data points, some of which have very low response values. It thus seems that SMBO based on the RF model tends to have problems with exploring new, previously-unseen areas, at least in one-dimensional optimization. This effect is more subtle than the previously-discussed failure modes. It might also be less pronounced in higher dimensions due to the random selection of split variables, as well as for the categorical inputs we consider in the next chapter.

11.3 Scaling To Many Data Points

For target algorithms whose runtime distributions have high variance, algorithm configuration procedures often need to execute thousands or tenthousands of target algorithm runs in order to reliable estimate a good parameter configuration. The time complexity of different models

[3]Thanks to Eric Brochu, who has first found a similar failure mode in his independent experiments using random forests and discussed it with the author of this thesis.



(a) SMBO based on RF fit, step 50 (b) Zoom into small region of same plot

Figure 11.7: One potential failure mode of SMBO based on RF models. The search prefers regions with many data points.

scales differently with the resulting large amount of data. This scaling behaviour is especially important in the context of models within SMBO: in each iteration, we need to fit a model to inform the selection of the next configuration(s).[4]

11.3.1 Computational Complexity of Random Forests

The computational cost for fitting a random forest is quite manageable. At a single node with n data points, it takes time $O(v \cdot n \log n)$ to identify the best combination of split variable and point (where $v = \max(1, \lfloor \mathit{perc} \cdot d \rfloor)$ denotes the number of split variables considered). This is because for each continuous split variable j, we can sort the n values $\boldsymbol{\theta}_{1j}, \ldots, \boldsymbol{\theta}_{nj}$ and only consider up to $n-1$ possible split points between different values. For the squared error loss function we use, the computation of $l(j, s)$ (see Equation 11.2 on page 169) can be performed in amortized $O(1)$ time for each of j's split points s, such that the total time required for determining the best split point of a single continuous variable is $O(n \log n)$. The complexity of building a regression tree depends on how balanced it is. In the worst case, one data point is split off at a time, leading to a tree of depth $n-1$ and a complexity of $O(v \cdot \sum_{i=1}^{n}(n-i)\log(n-i))$, which is $O(v \cdot n^2 \log n)$. In the best case—a balanced tree—we have the recurrence relation $T(n) = v \cdot n \log n + 2 \cdot T(n/2)$, leading to a complexity

[4] One can of course update the previous iteration's model with the new data points acquired since the last model fit. However, it is not clear that this will lead to improvements. While updating a GP model with fixed hyper-parameters can be implemented very efficiently, the optimal hyper-parameters can change dramatically when new training data becomes available, such that simply updating the model without re-optimizing the hyper-parameters can lead to suboptimal models. Likewise, one could locally update random forests with new training data, only occasionally re-learning the overall structure. Indeed, we experimented with this method, but preliminary experiments suggested that the resulting models used new, very informative, training data less effectively than models learned from scratch in each iteration.

of $O(v \cdot n \log^2 n)$. In our experience, trees are certainly not perfectly balanced but much closer to this best case than to the worst case. For example, 10000 data points typically led to tree depths between 25 and 30 (for reference, $\log_2(10000) \approx 13.3$). Building B trees simply takes B times as long as building a single tree. Thus, assuming perfectly balanced trees, the complexity of learning a random forest is $O(B \cdot m \cdot n \log^2 n)$.

Prediction with random forests is quite cheap. One merely needs to propagate new query points $\boldsymbol{\theta}$ down each tree. At each node with split variable j and split point s, one only needs to compare $\boldsymbol{\theta}_j$ to s, an $O(1)$ operation. In the worst case of depth $n-1$, this takes $O(n)$ per tree, in the best (balanced) case $O(\log n)$. Thus, in the best case, the complexity for prediction at a new data point is $O(B \log n)$.

11.3.2 Approximate Gaussian Processes

As described in Section 9.2, fitting a standard Gaussian process can be time-consuming. A gradient optimizer is used to determine hyper-parameters, and in each of h steps of the hyper-parameter optimization we need to invert a n by n matrix, leading to a computational complexity of $O(h \cdot n^3)$. Various approximations exist to reduce this complexity (see, *e.g.*, Quinonero-Candela et al., 2007). Here, we use the projected process (PP) approximation. We only give the final equations for predictive mean and variance; for a derivation, see the book by Rasmussen and Williams (2006). The PP approximation to GPs uses a subset of p of the n training data points, the so-called *active set*. Let v be a vector holding the indices of these p data points. Let $k(\cdot,\cdot)$ denote the GP covariance function and let K_{pp} denote the p by p matrix with $K_{pp}(i,j) = k(\boldsymbol{\theta}_{v(i)}, \boldsymbol{\theta}_{v(j)})$. Similarly, let $\boldsymbol{K_{pn}}$ denote the p by n matrix with $\boldsymbol{K_{pn}}(i,j) = k(\boldsymbol{\theta}_{v(i)}, \boldsymbol{\theta}_j)$; finally, let $\boldsymbol{K_{np}}$ denote the transpose of $\boldsymbol{K_{pn}}$. We then have

$$p(o_{n+1}|\boldsymbol{\theta}_{n+1}, \boldsymbol{\theta}_{1:n}, o_{1:n}) = \mathcal{N}(o_{n+1}|\mu_{n+1}, \text{Var}_{n+1}), \tag{11.5}$$

where

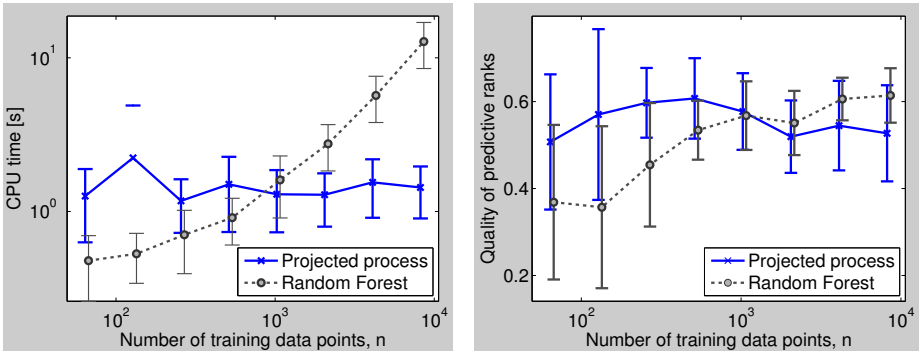
$$\mu_{n+1} = \boldsymbol{k_*}^\intercal(\sigma^2 \boldsymbol{K_{pp}} + \boldsymbol{K_{pn}}\boldsymbol{K_{np}})^{-1}\boldsymbol{K_{pn}}\, o_{1:n} \tag{11.6}$$

$$\text{Var}_{n+1} = k_{**} - \boldsymbol{k_*}^\intercal \boldsymbol{K_{pp}}^{-1}\boldsymbol{k_*} + \sigma^2 \boldsymbol{k_*}^\intercal(\sigma^2 \boldsymbol{K_{pp}} + \boldsymbol{K_{pn}}\boldsymbol{K_{np}})^{-1}\boldsymbol{k_*}, \tag{11.7}$$

and $\boldsymbol{k_*}$ and k_{**} are as defined in Section 9.2.

These equations assume a kernel with fixed hyper-parameters. We perform hyper-parameter optimization using a set of p data points randomly sampled without repetitions from the n input data points. We then sample an independent set of p data points for the subsequent projected process approximation; in both cases, if $p > n$, we only use n data points.

This approximation can greatly reduce the computational complexity. The hyper-parameter optimization based on p data points takes time $O(h \cdot p^3)$. In addition, there is a one-time cost of $O(p^2 \cdot n)$ for the projected process equations. Thus, the complexity for fitting the approximate GP model is $O([h \cdot p + n] \cdot p^2)$. The complexity for predictions with this PP approximation is $O(p)$ for the mean and $O(p^2)$ for the variance of the predictive distribution (Rasmussen and Williams, 2006).



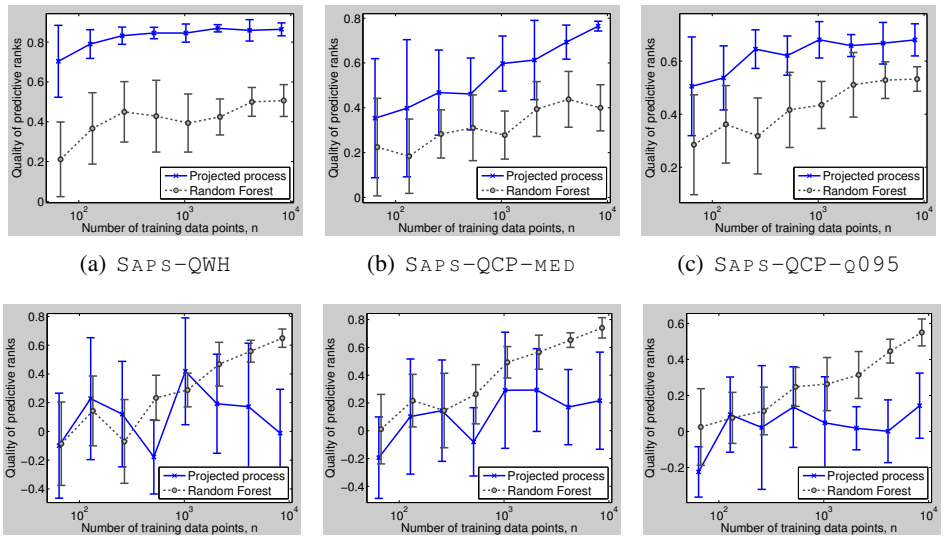
(a) Time complexity (b) Predictive quality of ranks

Figure 11.8: [Scaling of RF and approximate GP model with the number of training data points, n, for scenario SAPS-QCP-Q075. We use $B = 10$ trees and an active set of size $p = 100$. We built models for $8, 16, 32, \ldots, 4\,096$, and $8\,192$ training data points. For each size, we performed 10 repetitions and plot mean and standard deviations. To improve readability, the x-values of the random forest are slightly shifted to the right.

11.3.3 Experiments for Scaling Behaviour

Having discussed the theoretical time complexity of fitting RF and approximate GP models, we now study their empirical scaling behaviour in the number of data points, both in terms of computational time and in terms of model quality. We also study their scaling behaviour as we grow model complexity: the number of trees in random forests and the size of the active set in approximate GP models. In our RF models, here we use $n_{min} = 10$. As in Section 10.2, we measure model quality as the quality of predictive ranks on a set of good parameter configurations for the respective configuration scenarios. (For a discussion of that performance measure, see Section 10.2.1.) The sets of good parameter configurations we used here were determined as described in Section 10.2.1, using the SPO* procedure with a time budget of five hours per run.

For our study of scaling performance with the number of training data points, we used RF models with a fixed number of 10 trees and approximate GP models with 100 data points in the active set. Both types of models were trained using log-transformed data. Figure 11.8 shows their scaling behaviour for configuration scenario SAPS-QCP-Q075 when using up to $8\,192$ data points for training. For the RF models both time complexity and predictive quality improved with more training data points. In contrast, for the approximate GP models, both time complexity and performance were not affected much by the number of training data points. For the time complexity, this matches our expectation. Recall that the projected process approximation takes time $O([h \cdot p + n] \cdot p^2)$, where h is the number of steps in the hyper-parameter optimization and p is the size of the active set. Here, we used $p = 100$ and h was typically around 30, such that $h \cdot p$ was only a factor of 3 below the largest n we considered. We hypothesize that predictive quality did not increase with larger n because we



(a) SAPS-QWH (b) SAPS-QCP-MED (c) SAPS-QCP-Q095

(d) SAPS-SWGCP-MED (e) SAPS-SWGCP-Q075 (f) SAPS-SWGCP-Q095

Figure 11.9: Predictive quality of ranks for RF and approximate GP models with a growing number of training data points, n.

used a constant number of data points in the hyper-parameter optimization.

We now study the same kind of data for our six other configuration scenarios. The time complexity plots look very similar to Figure 11.8(a) for the other scenarios, and we omit them for brevity. The plots for model performance are more interesting. In Figure 11.9, we show that the approximate GP models yielded better performance for SAPS-QWH and the QCP scenarios, while the RF models performed better for the SWGCP scenarios. In all scenarios, the RF models improved with more data points; this was less clear for the approximate GP models.

Now, we study a different type of scaling behaviour: for a fixed-size data set, how do RF and approximate GP models scale when we increase the number of trees and the size of the active set, respectively? In Figure 11.10, we plot predictive performance against computational complexity of the various methods. The performance of RF models improved substantially up to about 64 trees and started to flatten out afterwards. For the approximate GP model, as expected, time complexity increased with the size of the active set. However, unexpectedly, its predictive performance became somewhat *worse* in the example shown. We hypothesize that this is caused by ill-conditioning or problems in the numerical optimization of hyper-parameters.

For the other configuration scenarios, shown in Figure 11.11, results were similar. Overall, we can see that the RF models were very robust: more trees always improved performance at the price of increased computational cost. In contrast, a larger active set sometimes improved the approximate GP models and sometimes worsened it. Since small active sets yielded overall good performance of the approximate GP models at a fairly low computational cost, we fixed

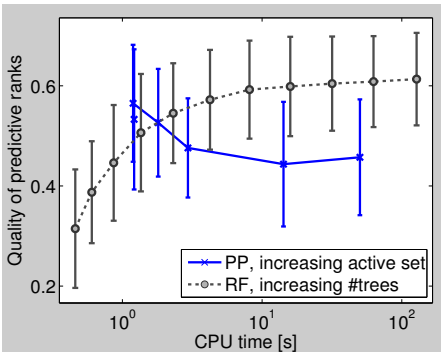


Figure 11.10: Scaling of RF models (with # trees) and of approximate GP models (with size of active set) for configuration scenario SAPS-QCP-Q075. We plot the performance (quality of predictive ranks for test set *Good*) versus the time the method required. For the projected process, we allowed active set sizes of $32, 64, 128, \ldots, 1024$, and for RFs we allowed $1, 2, 4, \ldots, 1024$ trees.

the size of the active set to $p = 100$ for all experiments in the rest of this thesis. Likewise, we fixed the number of trees in RF models to $B = 10$ for the rest of the thesis.

11.4 Model Quality

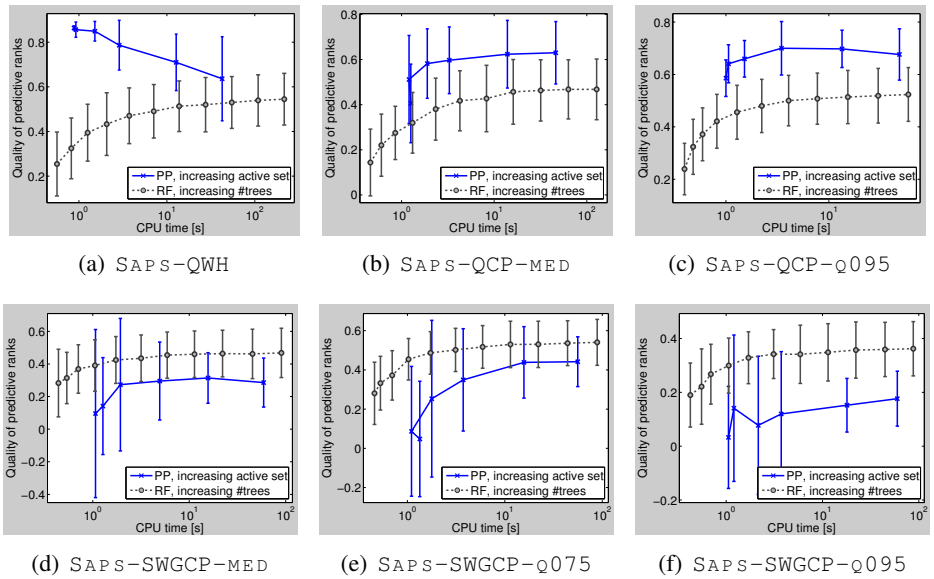
In this section, we compare the quality of approximate GP, RF, and DACE models for our seven configuration scenarios. As training data for each model, we employed 1 001 data points: single runtimes of SAPS for its default and for 1 000 randomly-sampled configurations. Some of these runs timed out after $\kappa_{max} = 5$ seconds and—according to the penalized average runtime (PAR) criterion with penalty constant 10—were counted as having taken 50 seconds (see Section 3.4 for a discussion of PAR). We thus learned models that directly predict PAR. As before, to judge qualitative model performance we used two sets of parameter configurations: *Random*, a set of 100 randomly-sampled configurations, and *Good*, the same set of 100 "good" configurations already used in Section 11.3.3.

11.4.1 Diagnostic Plots

First, we qualitatively assess model performance for one configuration scenario, SAPS-QCP-Q075. Figure 11.12 shows diagnostic plots for our three types of models, both trained on untransformed and log-transformed runtime data.

For the DACE model (in the first row of figures), we notice a qualitatively-similar trend as already observed in Section 10.2.2: the log transformation greatly improves model performance. This is reflected in our quantitative measures: the quality of predictive ranks on test set *Good* (points shown in green in Figure 11.12) was -0.07 (untransformed) *vs* 0.28 (log-transformed), and the EIC quality was 0.43 *vs* 0.62.

For the approximate GP model (second row of Figure 11.12), the log transform also im-



(a) SAPS-QWH (b) SAPS-QCP-MED (c) SAPS-QCP-Q095

(d) SAPS-SWGCP-MED (e) SAPS-SWGCP-Q075 (f) SAPS-SWGCP-Q095

Figure 11.11: Scaling of RF models (with # trees) and of approximate GP models (with size of active set), for other configuration scenarios.

proved performance and especially improved predictions for the best parameter configurations. Quantitatively, quality of predictive ranks was 0.28 (untransformed) *vs* 0.64 (log-transformed), and EIC quality was 0.75 *vs* 0.78. Also note that the approximate GP led to very small uncertainty estimates.

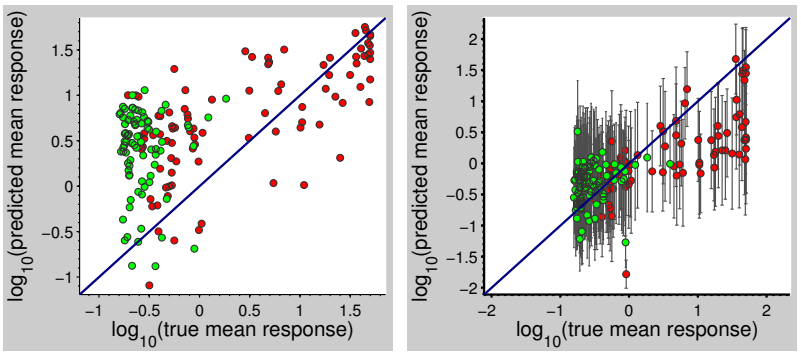
Finally, for the RF model (third row in Figure 11.12), the difference between the log and nonlog models appeared least pronounced. This was confirmed by our first quantitative measure: the predictive quality of ranks was 0.62 (untransformed) *vs* 0.63 (log-transformed). However, the uncertainty estimates for the version using untransformed data appeared not to be informative, leading to poor EIC quality: -0.34 (untransformed) *vs* 0.16 (log-transformed).

11.4.2 Overall Model Performance

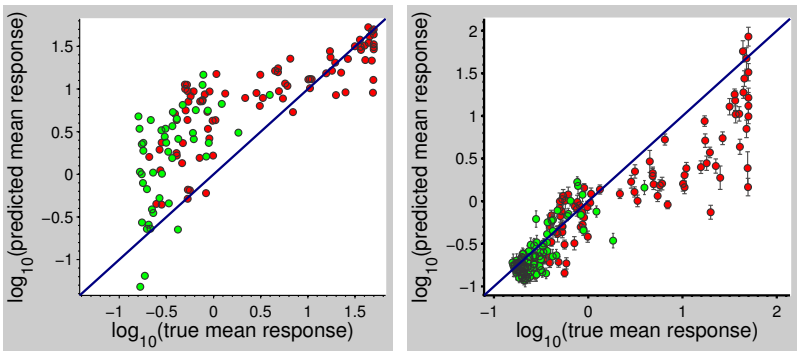
While these quantitative plots highlight interesting characteristics of the various models, it is important to study performance across multiple runs and across different domains to capture noise effects and differences between scenarios.

In Figure 11.12, we compare four types of models: the approximate GP model, the DACE model, and RF models with $n_{min} = 10$ and $n_{min} = 1$. Overall, the DACE model led to poor rank predictions and RMSE, but to quite competitive EIC quality. Model learning took about 1.5 orders of magnitude longer than for the approximate GP model and two orders of magnitude longer than for the RF model with $n_{min} = 10$.

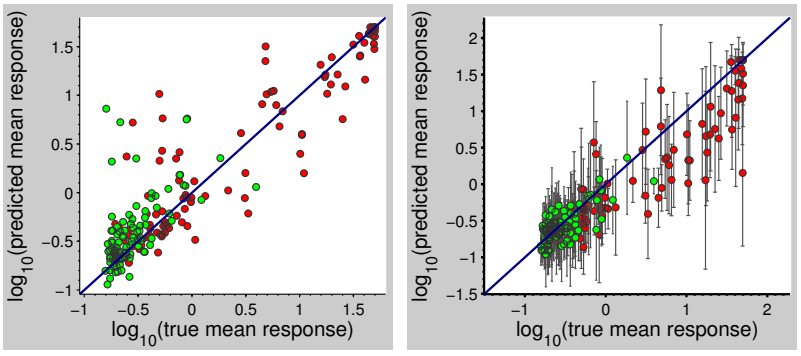
In terms of rank predictions, the approximate GP model performed much better than the RF model for some scenarios but worse for others. However, in terms of EIC quality, it



(a) DACE model, untransformed data (b) DACE model, log-transformed data

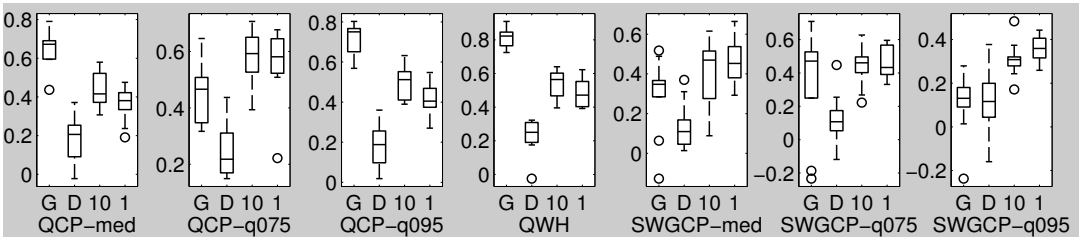


(c) Approximate GP, untransformed data (d) Approximate GP, log-transformed data

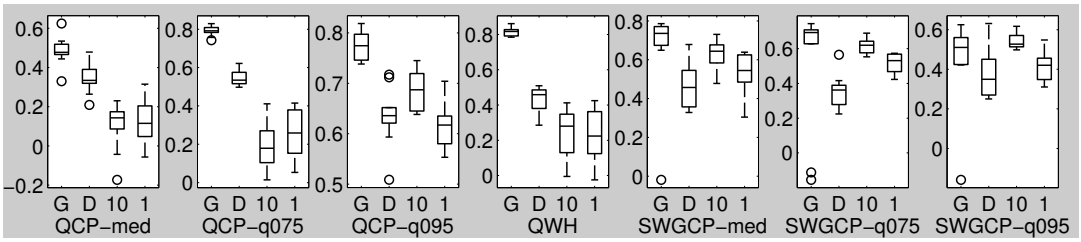


(e) Random forest, untransformed data (f) Random forest, log-transformed data

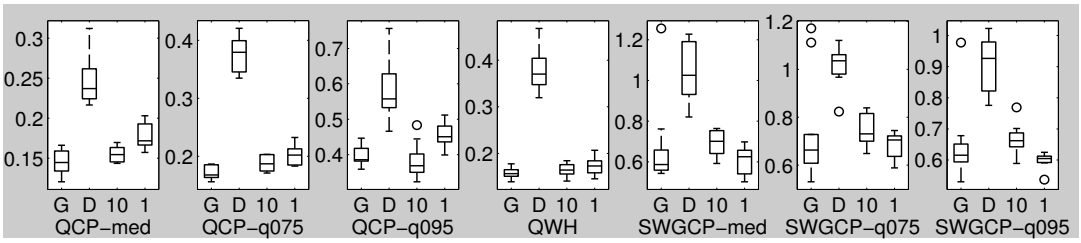
Figure 11.12: Predicted *vs* actual test set cost for approximate GP, RF, and DACE model, on scenario SAPS-QCP-Q075. Left column: untransformed data; right column: log-transformed data. Test sets: *Good* shown in green, *Random* shown in red.



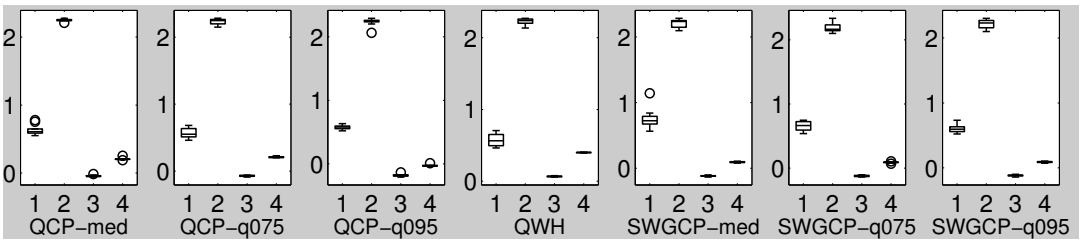
(a) Quality of predictive ranks (high is good, 1 is optimal)



(b) EIC quality (high is good, 1 is optimal)

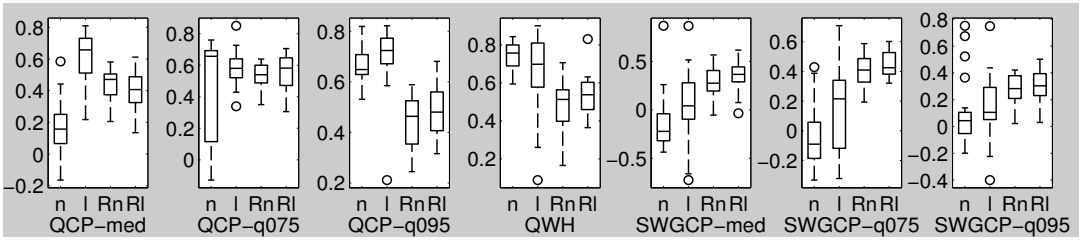


(c) Root mean squared error (RMSE; low is good, 0 is optimal)

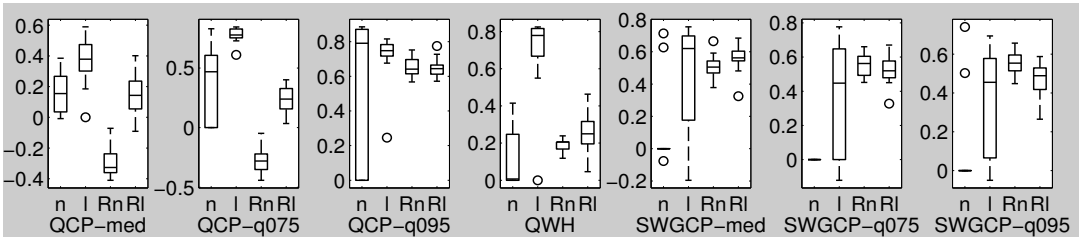


(d) Log_{10} of CPU time (in seconds)

Figure 11.13: Comparison of models. We performed 25 runs for each model with different training but identical test data and show boxplots for the respective quantities across the 25 runs. In each plot, "G" denotes an approximate GP model "D" denotes the DACE model, "10" denotes a RF model with $n_{min} = 10$ and "1" a RF model with $n_{min} = 1$.



(a) Quality of predictive ranks



(b) EIC quality

Figure 11.14: Comparison of models built on original and log-transformed data. We performed 25 runs for each model with different training but identical test data and show boxplots for our performance measures across the 25 runs. In each plot, "n" and "l" denote approximate GP models trained on the original and log-transformed data, respectively; "Rn" and "Rl" denote RF models with $n_{min} = 10$ trained on the original and log-transformed data, respectively.

performed clearly best on six of the scenarios, and amongst the best for the last one. It also tended to have the lowest RMSE of all models.

Finally, comparing the two versions of RF models, the version with $n_{min} = 10$ performed slightly better overall. In terms of predictive quality of ranks, as well as in terms RMSE, this version tended to perform better for the `QCP` scenarios, whereas the version with $n_{min} = 1$ tended to perform better for the `SWGCP` scenarios. However, the version with $n_{min} = 10$ tended to yield higher EIC quality and was also consistently faster. As a consequence, we fixed $n_{min} = 10$ for the experiments in the remainder of this thesis.

11.4.3 Log Transformation for approximate GPs and Random Forests

As a final part of our study of model quality, we quantitatively compared the performance of models based on untransformed and log-transformed data. For the DACE model, we already showed the benefits of the log-transformation in Section 10.2.2; thus, we omit it here.

The data shown in Figure 11.12 suggested that RF and approximate GP models trained on untransformed data were competitive at predicting ranks, but that EIC based on these models did not correlate well with performance. Figure 11.12 is based on a single run for scenario `SAPS-QCP-Q075`. As Figure 11.14 shows that run was quite representative for that scenario: across 25 independent runs, the log transformation did *not* lead to consistent improvements in

predicting ranks, but it *did* improve the quality of EIC for both the RF and the approximate GP model. For the approximate GP model, behaviour was qualitatively similar for most other configuration scenarios: the log transformation strongly increased the EIC quality and tended to increase the quality of predictive ranks. In contrast, for the RF model, the log transform only substantially improved the EIC quality for one other configuration scenario: SAPS-QCP-MED. For the other scenarios, the log transform only mildly improved EIC quality and in one scenario (SAPS-SWGCP-Q095) actually worsened it.

In summary, the log transform clearly improved the approximate GP model and somewhat improved the RF model. Thus, in the rest of this thesis, we focus on models trained on log-transformed response values.

11.5 Sequential Optimization Process

Our main interest in this thesis is on algorithm configuration, not on models or their performance. Thus, the ultimate question in this chapter is whether our new models improve the sequential model-based optimization (SMBO) approach to algorithm configuration. We first introduce configuration procedures based on the new models and then demonstrate that they significantly improve upon SPO*.

11.5.1 Configuration Procedures based on Different Models

SMBO approaches based on the models we introduced in this chapter have rather different properties than the previously-discussed SPO variants based on a DACE model fitted to empirical cost statistics. In particular, the DACE model (a noise-free GP model) can be used to fit almost arbitrary (user-defined) statistics of a configuration's cost distribution, such as mean plus variance. This is in principle also possible with our version of random forests, which computes user-defined cost statistics at its leaves. However, we would not expect it to work well since the tree construction implicitly optimizes for mean predictions. (Approximate) GP models are even less flexible: they always fit the mean function assuming Gaussian observation noise. Furthermore, the DACE predictions for previously-observed parameter configurations $\boldsymbol{\theta}$ always have zero uncertainty, even if only a single run has been performed for $\boldsymbol{\theta}$. In contrast, our approximate GP and RF models handle observation noise better and—unlike the DACE model—yield nonzero uncertainty estimates for previously-observed configurations. Intuitively, this can have a major impact on SMBO.

To underline this distinction, we give a new name to configuration procedures using models other than the noise-free DACE model: ACTIVECONFIGURATOR. This is in reference to algorithm configuration, as well as to the machine learning technique *active learning*—which queries the data points deemed most useful to improve the model fit. We abbreviate ACTIVECONFIGURATOR based on RF and approximate GP models as *AC(RF)* and *AC(GP)*, respectively.

ACTIVECONFIGURATOR uses the same components as SPO* except that it is based on a different model and sets the number of previously-used configurations, p, to zero. In Section 10.3.1, we motivated this latter mechanism by the fact that SMBO based on the DACE model attributes EIC zero to all previously-selected configurations and thus prohibits their further use.

This problem does not occur anymore for our new models, and an experimental evaluation of AC(RF) with and without this mechanism did not show any significant differences. Thus, we dropped the mechanism to simplify ACTIVECONFIGURATOR.

11.5.2 Experimental Evaluation of Configuration Procedures

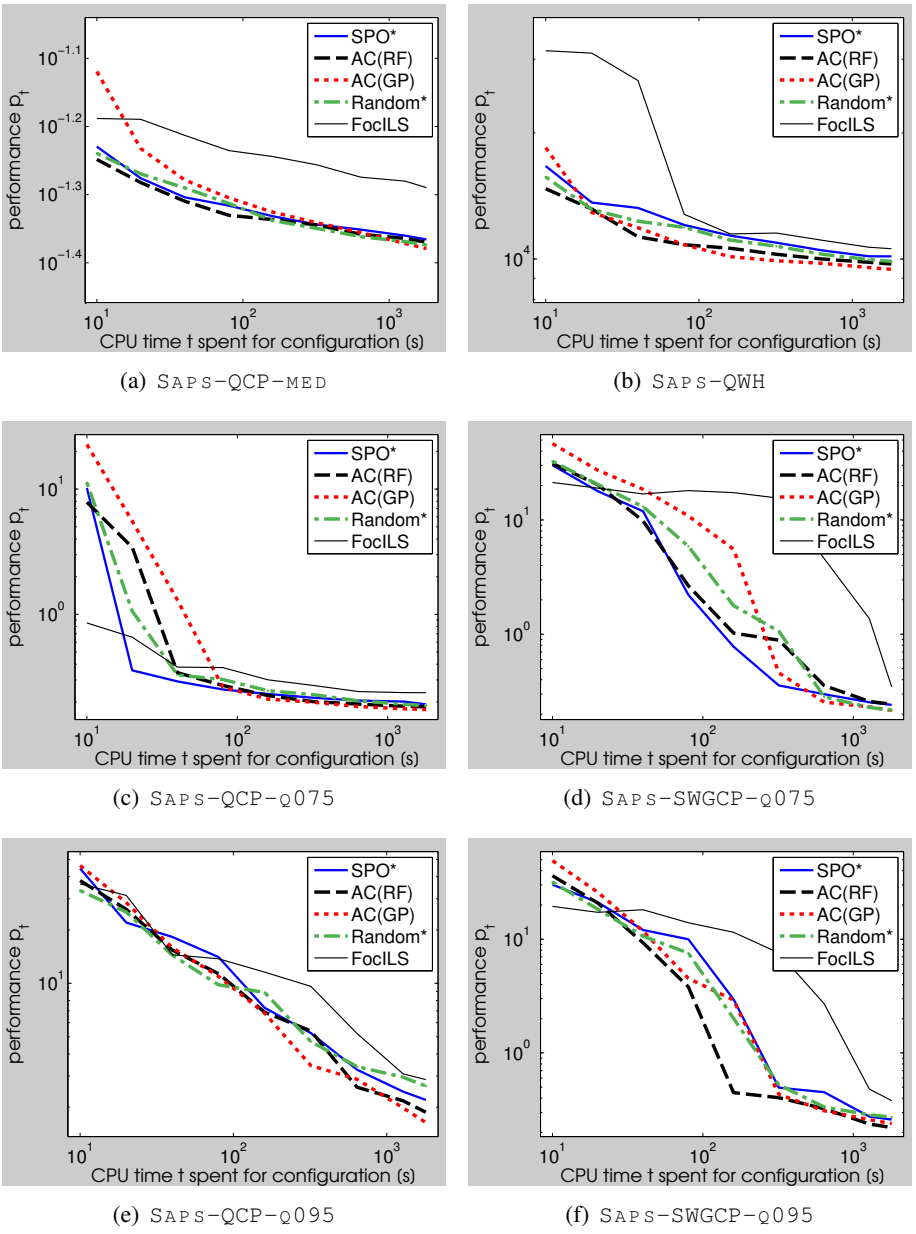
We now experimentally compare our various configuration procedures: SPO*, AC(RF), AC(GP), and, for reference, RANDOM* and FOCUSEDILS. In this evaluation, we used the seven SINGLEINSTCONT configuration scenarios considered throughout this chapter. Note that in these scenarios we optimize continuous SAPS parameters; FOCUSEDILS is the only procedure in this comparison restricted to a discretized subspace (using the same discretization as we used in Chapters 5 and 7). We performed 25 runs of each configuration procedure for each scenario and evaluated test performances $p_{test,t}$ (for SAPS-QWH: median number of SAPS search steps; for the other scenarios: SAPS penalized average runtime) for various time budgets for configuration, t.

In Figure 11.15, we show mean test performance $p_{test,t}$ for all configuration procedures and scenarios. First, we note that SPO*, AC(RF), AC(GP), and RANDOM* all yielded very competitive performance. In particular, in all SINGLEINSTCONT configuration scenarios, all of them yielded lower mean test performance than FOCUSEDILS. (This is not surprising since FOCUSEDILS is restricted to a discretized subspace which may not include the optimal configurations. Thus, here we give FOCUSEDILS performance only for reference; in Section 12.3.4, we repeat this comparison when all configurators are restricted to the same discrete configuration space.) Another trend in the data shown in Figure 11.15 is that ACTIVECONFIGURATOR based on approximate GP models was often slower at finding good parameter configurations, but, given enough time (between 100 and 1 000 seconds), typically performed best. Finally, for some configuration scenarios, RANDOM* found good configurations somewhat slower than the other configurators. However, by the end of the search process, the differences were small (except for scenario SAPS-QCP-Q095).

In Table 11.1 and Figure 11.16, we summarize test performance at then end of the search process across the 25 runs. Figure 11.16 provides boxplots, while Table 11.1 lists means and standard deviations across the 25 runs, as well as the result of pairwise significance tests. Overall, AC(GP) clearly yielded the best configurations for the time budget of half an hour; it significantly outperformed both SPO* and FOCUSEDILS in all 7 SINGLEINSTCONT scenarios, RANDOM* in 3 and AC(RF) in 2 scenarios, and yielded the best mean performance in all but one scenario. AC(RF) was second best, significantly outperforming FOCUSEDILS in all 7 scenarios, SPO* in 3 scenarios, and RANDOM* in one. As we already noted in Section 10.4, SPO* did not perform better than RANDOM* and was even significantly worse in one scenario. We conclude that the use of our improved new models, random forests and especially approximate GPs, led to significant performance improvements of SMBO.

11.6 Chapter Summary

In this chapter, we studied the use of alternative models in the sequential model-based optimization (SMBO) framework. First, we introduced a novel variant of random forests (RFs)



(a) SAPS-QCP-MED (b) SAPS-QWH

(c) SAPS-QCP-Q075 (d) SAPS-SWGCP-Q075

(e) SAPS-QCP-Q095 (f) SAPS-SWGCP-Q095

Figure 11.15: Comparison of configurators for SINGLEINSTCONT scenarios, over time. We performed 25 runs of the configurators and computed their test performance $p_{test,t}$ at time steps $t = 10, 20, 40, \ldots, 1280, 1800$ seconds; we plot mean $p_{test,t}$ across the 25 runs. We omitted scenario SAPS-SWGCP-MED, the plot for which qualitatively resembles the one for SAPS-SWGCP-Q075.

Scenario	SPO*	AC(RF)	AC(GP)	RANDOM*	FocusedILS
SAPS-QWH [$\cdot 10^3$]	10.2 ± 0.57	9.73 ± 0.45	$\mathbf{9.45 \pm 0.33}$	9.87 ± 0.41	10.6 ± 0.49
SAPS-QCP-MED [$\cdot 10^{-2}$]	4.31 ± 0.20	4.27 ± 0.18	$\mathbf{4.18 \pm 0.16}$	4.23 ± 0.15	5.13 ± 0.40
SAPS-QCP-Q075	0.19 ± 0.02	0.18 ± 0.01	$\mathbf{0.17 \pm 0.01}$	0.19 ± 0.01	0.24 ± 0.02
SAPS-QCP-Q095	2.20 ± 1.17	1.88 ± 0.97	$\mathbf{1.64 \pm 0.87}$	2.63 ± 1.24	2.87 ± 3.20
SAPS-SWGCP-MED	0.18 ± 0.03	0.17 ± 0.02	$\mathbf{0.16 \pm 0.02}$	0.17 ± 0.02	0.27 ± 0.12
SAPS-SWGCP-Q075	0.24 ± 0.04	0.24 ± 0.10	$\mathbf{0.22 \pm 0.04}$	$\mathbf{0.22 \pm 0.03}$	0.35 ± 0.08
SAPS-SWGCP-Q095	0.26 ± 0.05	$\mathbf{0.22 \pm 0.03}$	0.24 ± 0.06	0.27 ± 0.11	0.38 ± 0.16

Scenario	Pairs of configurators with statistically-significant performance differences
SAPS-QWH	SPO*/RF, SPO*/GP, SPO*/Foc, RF/GP, RF/Foc, GP/R*, GP/Foc, R*/Foc
SAPS-QCP-MED	SPO*/GP, SPO*/Foc, RF/Foc, GP/Foc, R*/Foc
SAPS-QCP-Q075	SPO*/GP, SPO*/Foc, RF/GP, RF/Foc, GP/R*, GP/Foc, R*/Foc
SAPS-QCP-Q095	SPO*/GP, RF/R*, RF/Foc, GP/R*, GP/Foc
SAPS-SWGCP-MED	SPO*/RF, SPO*/GP, SPO*/Foc, RF/Foc, GP/R*, GP/Foc, R*/Foc
SAPS-SWGCP-Q075	SPO*/GP, SPO*/R*, SPO*/Foc, RF/Foc, GP/Foc, R*/Foc
SAPS-SWGCP-Q095	SPO*/RF, SPO*/GP, SPO*/Foc, RF/Foc, GP/Foc, R*/Foc

Table 11.1: Quantitative comparison of configurators for SINGLEINSTCONT scenarios. We performed 25 runs of the configurators and computed their test performance $p_{test,t}$ for a configuration time of $t = 1\,800$s. Here, we give mean $\pm$ standard deviation across the 25 runs. We also performed pairwise Mann Whitney U tests to compare the configurators' performances across the 25 runs each, and list significantly-different pairs; we abbreviate AC(RF) as RF, AC(GP) as GP, RANDOM* as R*, and FOCUSEDILS as Foc. Figure 11.16 visualizes this data.

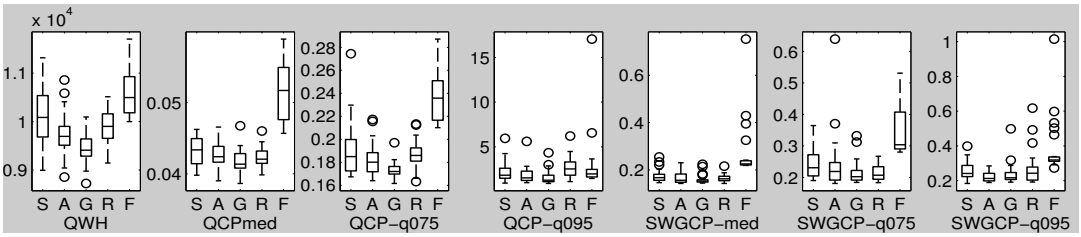


Figure 11.16: Boxplot comparison of configuration procedures for SINGLEINSTCONT scenarios. "S" stands for SPO*, "A" for AC(RF), "G" for AC(GP), "R" for RANDOM*, and "F" for FocusedILS. We show boxplots for the data presented in Table 11.1.

that provides uncertainty estimates, and described an existing approximation of Gaussian process (GP) models. We compared these models and the DACE model (the noise-free GP model used in the SPO variants discussed in the previous chapters), demonstrating that the RF model can deal much better with non-standard observation noise, such as proportional and non-stationary noise.

We studied the theoretical and empirical scaling behaviour of RF and approximate GP models. Both types of models easily scaled to predictions with many thousands of data points. RF models consistently improved when based on more data points, while the approximate GP models only improved in some scenarios.

For a set of $1\,000$ training data points, compared to the DACE model used in the SPO

variants discussed in previous chapters, learning RF and approximate GP models was two and 1.5 orders of magnitude faster, respectively. These models also clearly outperformed the DACE model in terms of predicting ranks of configurations and in terms of prediction error. In terms of correlation between the expected improvement criterion based on the model and true quality of a configuration, the approximate GP model dominated the DACE model, while the RF model performed worse in some cases.

Finally, we introduced algorithm configuration procedures based on our new models, dubbed ACTIVECONFIGURATOR and abbreviated AC: AC(GP) when based on approximate GP models and AC(RF) when based on RF models. We demonstrated that AC(GP) significantly and sometimes substantially outperformed SPO* and RANDOM* on all seven SINGLEINST-CONT scenarios. AC(RF) was second in that comparison. Thus, while—based on the rather poorly-performing DACE model—SPO* did *not* outperform RANDOM*, the use of better models *did* lead to significant improvements.

For these continuous parameter optimization tasks, all configuration procedures studied here outperformed FOCUSEDILS, which was restricted to searching a discretized subspace. In the next chapter, we will revisit this comparison when all configurators work in exactly the same discrete configuration space.

Chapter 12

Extensions I: Categorical Variables

The only real valuable thing is intuition.
—Albert Einstein, German American theoretical physicist

All existing sequential model-based optimization (SMBO) methods of which we are aware are limited to the optimization of numerical parameters. In this chapter, we extend the scope of SMBO to include the optimization of *categorical* parameters. In particular, we extend our response surface models to handle categorical inputs and introduce a simple local search mechanism to optimize expected improvement across the settings of categorical parameters. We demonstrate that the resulting SMBO configurators outperform FOCUSEDILS for the configuration of algorithms with categorical parameters on single instances.

For the experiments in this chapter, we used our SINGLEINSTCAT scenarios (described in Section 3.5.3), in which the objective is to optimize the 26 discretized parameters of SPEAR on single SAT instances. We also used a discretized version of our SINGLEINSTCONT scenarios (described in Section 3.5.2), in which the objective is to optimize the 4 discretized parameters of SAPS on single SAT instances.

12.1 Models for Partly Categorical Inputs

In the previous chapter, we discussed models for training data $\mathcal{D} = \{(\boldsymbol{\theta}_i, o_i)\}_{i=1}^n$, with each $\boldsymbol{\theta}_i \in \mathbb{R}^d$. Now, we generalize this notion, allowing each dimension of $\boldsymbol{\theta}_i$ to be either continuous or categorical.

12.1.1 Categorical Inputs in Random Forests

Random forests handle categorical input variables very naturally. Recall from Section 11.1.1 that when splitting the training data $\{(\boldsymbol{\theta}_i, o_i)\}_{i=1}^n$ at a node on a continuous variable, we select a split point s, partitioning $\{(\boldsymbol{\theta}_i, o_i)\}_{i=1}^n$ into the two sets $\{(\boldsymbol{\theta}_i, o_i)|\theta_{i,j} < s\}$ and $\{(\boldsymbol{\theta}_i, o_i)|\theta_{i,j} \geq s\}$, where $\theta_{i,j}$ denotes the value of the j-th parameter of parameter configuration $\boldsymbol{\theta}_i$. Similarly, when splitting on a categorical variable j with k possible values $u_1, \ldots, u_k$, we select the best of the $2^k - 2$ binary partitionings of $\{u_1, \ldots, u_k\}$ which leave neither of the sets empty. For the squared error loss function used in Section 11.1.1, there exists an efficient procedure for doing so: for each value u_l, compute score $s_l = \text{mean}(\{o_i|\theta_{i,j} = u_l\})$, sort $(u_1, \ldots, u_k)$

by those scores, and only consider the k binary partitions with consecutive scores in each set. With this procedure, determining the optimal partitioning takes time $O(n + k \cdot \log k)$. Since the number of values present for each variable is upper-bounded by the number of data points, this is an $O(n \cdot \log n)$ procedure, just like finding the optimal split point for a continuous variable. Often, $k << n$, in which case learning for categorical inputs is actually faster than for continuous inputs. For prediction, at each split on a categorical variable, we need to perform a member query for a set with up to $k/2$ values. This can be implemented in time $O(\log k)$ as compared to the single $O(1)$ comparison to the split point for continuous variables.

In order to improve uncertainty estimates, we made a nonstandard implementation choice in our implementation of random forest models. This concerns splits on a categorical variable j at a node p where not all of j's possible values are present in the subset of the training data at p. An example is a case where j can take on values A, B, or C, but only values A and B are present in the training data at node p. One standard solution for this case is to create a ternary partitioning. Partition R_1, containing all data points $\boldsymbol{\theta}_i$ with $\theta_{i,j} = A$ would be associated with the subtree rooted at p's left child, R_2, containing all training data points $\boldsymbol{\theta}_i$ with $\theta_{i,j} = B$ with the subtree rooted at p's right child, and the remaining region R_3 with node p. The prediction for test data points $\boldsymbol{\theta}_i$ with $\theta_{i,j} = C$ that are propagated to node p would be p's constant, c_p. In contrast, in our implementation, we assign each value that is not present in the training data available at a node n to one of n's children, chosen uniformly at random. This does not typically change the mean prediction much but better reflects our uncertainty. Consider an example. Say, the first split in all trees is on categorical variable j, which can take on k values. However, in the training data only $k - 1$ values are present. What is the prediction for a test data point with the k-th value? In the standard implementation, it is the mean of all responses in the training set, with perfect agreement across the trees and thus zero uncertainty. In contrast, in our implementation, predictions typically differ across the trees, expressing our uncertainty about inputs with the unseen value. Our implementation also has a convenient side effect: the leaves of each tree completely partition the space, and we need not worry about special cases concerning regions associated with inner nodes.

12.1.2 A Weighted Hamming Distance Kernel for Categorical Inputs in Gaussian Processes

As discussed in Section 9.2, to apply Gaussian process regression, first we need to select a parameterized kernel function $K_\lambda : \boldsymbol{\Theta} \times \boldsymbol{\Theta} \to \mathbb{R}^+$, specifying the similarity between two parameter configurations. This remains true if some or all parameters are categorical. In Section 9.2, we used the standard weighted squared distance kernel function for numerical parameters:

$$K_\lambda(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) = \exp\left[\sum_{l=1}^{d}(-\lambda_l \cdot (\theta_{i,l} - \theta_{j,l})^2)\right],$$

where $\lambda_1, \ldots, \lambda_d$ are the kernel parameters. For categorical variables, instead of (weighted) squared difference, we simply use (weighted) Hamming distance:

$$K_\lambda(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) = \exp\left[\sum_{l=1}^{d}(-\lambda_l \cdot \delta(\theta_{i,l} \neq \theta_{j,l}))\right], \tag{12.1}$$

where δ is the Kronecker delta function.

In order to show that this kernel function is valid, we use the facts that any constant is a valid kernel function, and that the space of kernel functions is closed under addition and multiplication. We also use the fact that a kernel function $k(x, z)$ is valid if we can find an embedding ϕ such that $k(x, z) = \phi(x)^\intercal \cdot \phi(z)$ (Shawe-Taylor and Cristianini, 2004). Though straightforward, we are not aware of any prior use of this kernel or proof that it is indeed valid.[1]

For elements $x, z \in \Theta_i$ of a finite domain Θ_i, we first show that the weighted Hamming distance kernel function

$$k(x, z) = \exp\left[-\lambda\delta(x \neq z)\right] \tag{12.2}$$

is valid. (Note that the domain Θ_i of any categorical parameter θ_i is indeed finite.) Refer to the finitely many elements of Θ_i as $a_1, \ldots, a_m$. First, we define m-dimensional vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ with $v_i := (x = a_i)$ and $w_i := (z = a_i)$. We then define a kernel function $k_1(x, z)$ for $x, z \in \Theta_i$ as the dot-product of embeddings $\phi(x)$ and $\phi(z)$ in an m-dimensional space:

$$k_1(x, z) = \boldsymbol{v}^\intercal \cdot \boldsymbol{w} = \sum_{i=1}^{m} v_i \cdot w_i = \delta(x = z).$$

To bring this in the form of Equation 12.2, we add the constant kernel function

$$k_2(x, z) = c = \frac{\exp(-\lambda)}{1 - \exp(-\lambda)},$$

and then multiply by the constant kernel function

$$k_3(x, z) = 1/(1 + c) = 1 - \exp(-\lambda).$$

This yields the combined kernel function

$$\begin{aligned} k(x, z) &= (k_1(x, z) + k_2(x, z)) \cdot k_3(x, z) \\ &= \begin{cases} 1 & \text{if } x = z \\ \exp(-\lambda) & \text{otherwise} \end{cases} \\ &= \exp\left[-\lambda\delta(x \neq z)\right]. \end{aligned}$$

Multiplying together d separate kernels of the form in Equation 12.2 (one kernel per parameter) yields a kernel of the general form in Equation 12.1.

[1] Couto (2005) gives a recursive kernel function for categorical data that is related since it is also based on a Hamming distance.

For a combination of continuous parameters $\mathcal{P}_{cont}$ and categorical parameters $\mathcal{P}_{cat}$, we apply the combined kernel

$$K(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) = \exp\left[\sum_{l \in \mathcal{P}_{cont}} (-\lambda_l \cdot (\theta_{i,l} - \theta_{j,l})^2) + \sum_{l \in \mathcal{P}_{cat}} (-\lambda_l \cdot \delta(\theta_{i,l} \neq \theta_{j,l}))\right].$$

Again, this kernel function is valid since the space of kernel functions is closed under multiplication.

All other equations in Gaussian process models, as well as in the projected process approximation, remain exactly the same under this alternative kernel. The only remaining question, to be studied in the next section, is how well (approximate) GP models with this kernel work in practice.

12.2 Model Quality

In this section, we compare the model quality of RF and approximate GP models for predicting the runtime of algorithms with categorical parameters. Throughout this chapter, we use two types of configuration scenarios. First, we use the SINGLEINSTCONT scenarios already used in the previous chapter, but now with discretized parameters. This allows a direct comparison of models—and also configurators—working with the original numerical parameters and with a discretized version. Secondly, we use the SINGLEINSTCAT scenarios which deal with the configuration of SPEAR and thus involve many more parameters, many of which are categorical (the remaining numerical parameters were discretized exactly as in Part III of this thesis; see Section 6.3.1 for details on this discretization). For more details on these sets of configuration scenarios, see Sections 3.5.2 and 3.5.3.

As training data for each model, we employed $1\,001$ data points: single runtimes of the target algorithm (SAPS in the SINGLEINSTCONT scenarios, SPEAR in the SINGLEINSTCAT scenarios) for its default as well as configurations from a random Latin hypercube design (LHD) with $1\,000$ design points. For discretized parameter configuration spaces, we used random LHDs as follows. We created a random LHD with interval $[0, 1]$ in each dimension, and for a parameter j with possible discrete values $u_1, \ldots, u_k$, we selected value u_i (with $i = 1, \ldots, k-1$) for LHD points $\boldsymbol{\theta}_i$ with $\theta_{i,j} \in [\frac{i-1}{k}, \frac{i}{k})$, and value u_k for points with $\theta_{i,j} \in [\frac{k-1}{k}, 1]$. As before, according to our penalized average runtime (PAR) criterion (see Section 3.4), runs that timed out after $\kappa_{max} = 5$ seconds were counted as 50 seconds. Also as before, to assess qualitative model performance, we used two sets of parameter configurations: *Random*, a set of 100 randomly-sampled configurations, and *Good*, a set of 100 "good" configurations determined using FOCUSEDILS (with a time budget of five hours) as described in Section 10.2.1. Both *Random* and *Good* are subsets of the discretized parameter configuration space; in order to enable a direct comparison of model quality on the same test sets, we also use these sets to evaluate models learned using the continuous space.

12.2.1 Effect of Discretization on Model Quality

For the SINGLEINSTCONT configuration scenarios, we can use training data gathered with the original, continuous, target algorithm parameters or with discretized parameters. Here, we compare the quality these two types of training data yield, both for RF and approximate GP models.

The results of this comparison, given in Figure 12.1, show some interesting trends. Performance of the GP models deteriorated when trained using discretized parameter configurations: for all of our three measures of quality (see Section 10.2.1), performance decreased for most configuration scenarios. In contrast, the RF models tended to *improve* when trained on discretized parameter settings: for all three measures of model quality, performance improved in most configuration scenarios, sometimes quite substantially (see, *e.g.*, EIC quality for the SWGCP scenarios in Figure 12.1(b)). Runtimes for constructing the RF models were marginally smaller for the discretized data since the number of values considered for each discretized parameter was $k = 7$, much smaller than the number of data points, $n = 1\,000$.

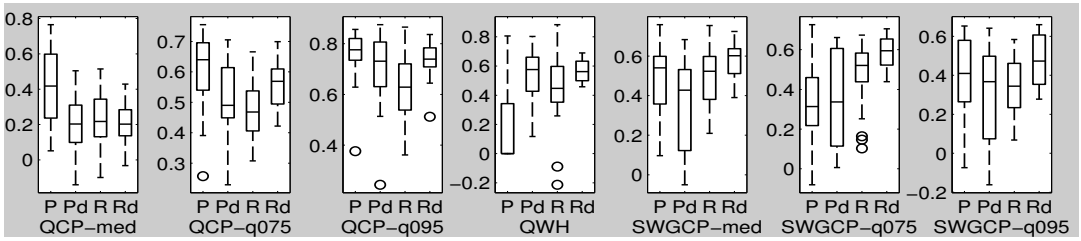
Overall, for the discretized version of the SINGLEINSTCONT scenarios, RF models tended to yield better predictions than approximate GP models. This is in contrast to continuous inputs, for which approximate GP models tended to perform better (see Section 11.4.2). In particular, for the discretized versions of the SWGCP scenarios, RF models performed better than approximate GPs with respect to all model performance measures. For the other scenarios, there was no clear winner. The difference of our results for numerical and categorical inputs is not surprising since RF models are well known to handle categorical inputs well.

12.2.2 Predictive Performance for Many Categorical Parameters

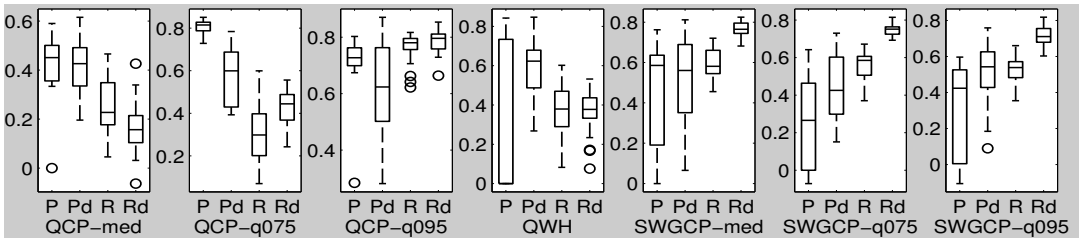
In Figure 12.2, we compare RF and approximate GP models for runtime predictions in the SINGLEINSTCAT scenarios, all of which deal with optimizing the 26 parameters of SPEAR (10 of which were categorical to start with, and 16 of which were numerical and discretized as described in Section 6.3.1). For most of these configuration scenarios, there was no clear winner. Consider, for example, scenario SPEAR-SWV-MED: here, the RF model yielded much better rank predictions and lower RMSE, but the GP model led to much better correlation of EIC and test performance. The only case for which one model (the RF) performed substantially better with respect to all three model performance measures is scenario SPEAR-IBM-MED. We will show later (in Section 12.3.4) that for this scenario, as well as for some of the SWGCP scenarios (for which RF models performed better than approximate GP models), configuration procedures based on RF models indeed yielded the best results.

12.3 Sequential Optimization Process

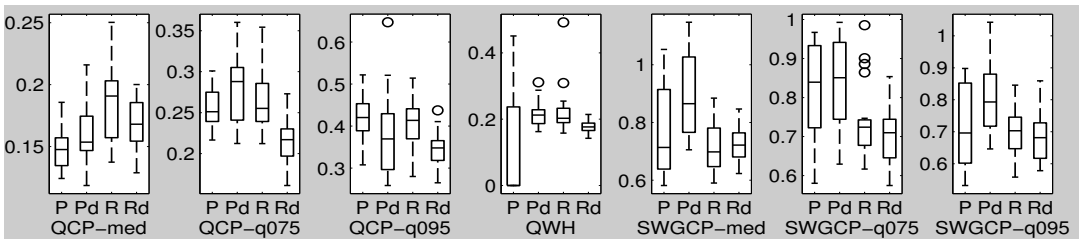
We demonstrated in the last section that our RF models could handle categorical variables better than approximate GP models. Now, we study the use of these methods within the SMBO framework, enabling model-based configuration of algorithms with categorical parameters. First, however, we describe how we optimize expected improvement in the presence of categorical parameters.



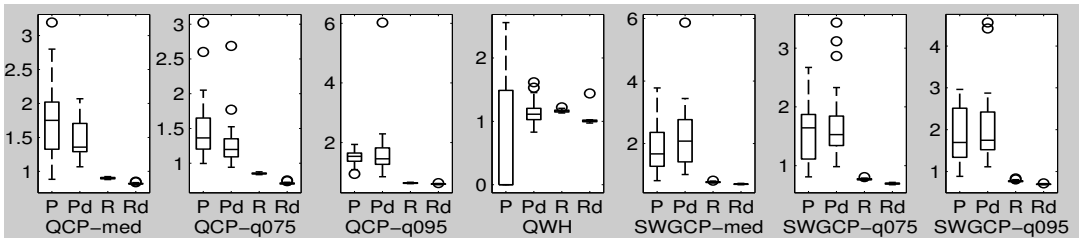
(a) Quality of predictive ranks (high is good, 1 is optimal)



(b) EIC quality (high is good, 1 is optimal)

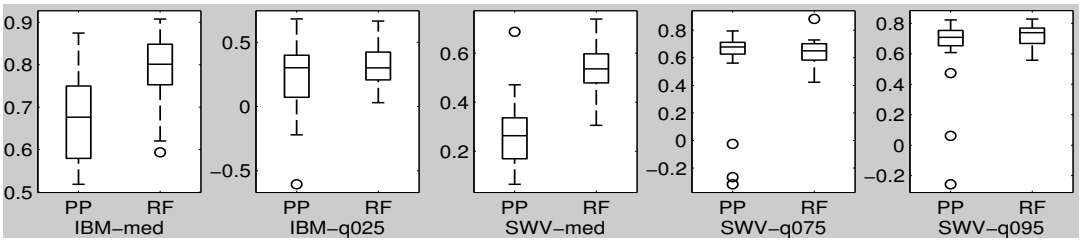


(c) Root mean squared error (RMSE; low is good, 0 is optimal)

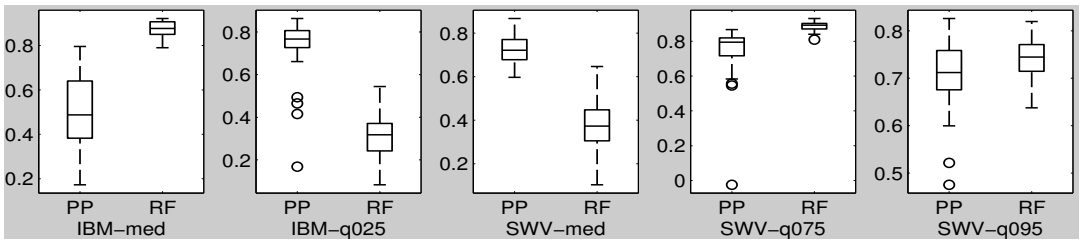


(d) CPU time (in seconds)

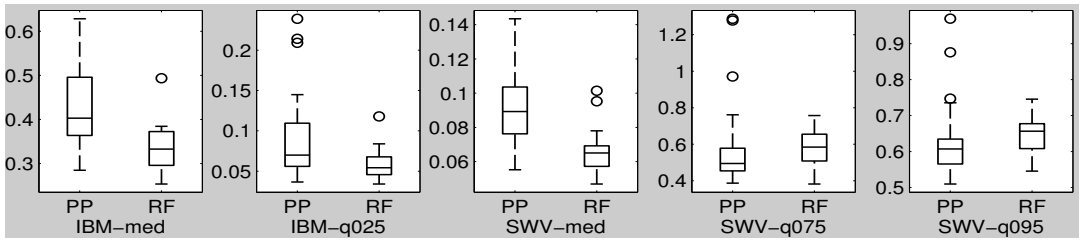
Figure 12.1: Comparison of approximate GP and RF models for SINGLEINSTCONT scenarios with discretized parameters. We performed 25 runs for each model with different training but identical test data and show boxplots for the respective quantities across the 25 runs. In each plot, "P" and "Pd" denotes the approximate GP model trained on continuous and discretized data, respectively. Likewise, "R" and "Rd" denote the RF model trained on continuous and discretized data.



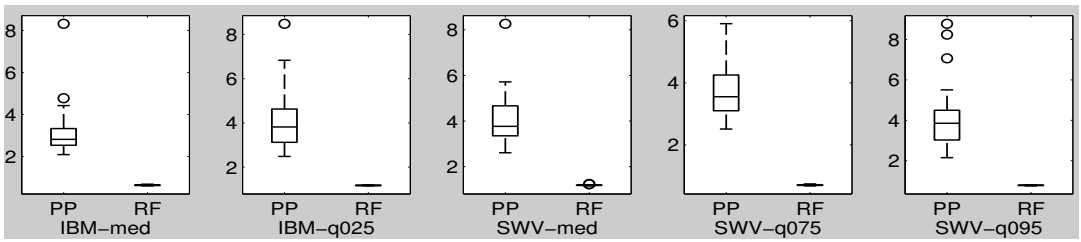
(a) Quality of predictive ranks (high is good, 1 is optimal)



(b) EIC quality (high is good, 1 is optimal)



(c) Root mean squared error (RMSE; low is good, 0 is optimal)



(d) CPU time (in seconds)

Figure 12.2: Comparison of approximate GP ('PP') and RF models for SINGLEINSTCAT scenarios. We performed 25 runs for each model with different training but identical test data and show boxplots for the respective quantities across the 25 runs.

12.3.1 Optimization of Expected Improvement for Categorical Parameters

The SMBO methods we introduced (SPO$^+$, SPO*, and ACTIVECONFIGURATOR) all optimized expected improvement by simply drawing $10\,000$ configurations $\boldsymbol{\theta}$ uniformly at random from $\boldsymbol{\Theta}$ and sorting them with respect to expected improvement. For very large parameter configuration spaces with only few good configurations, this cannot be expected to scale well. While this is true for continuous and categorical configuration spaces alike, we have not encountered large enough continuous configuration spaces for the problem to manifest itself. (In experiments we omit due to their preliminary nature, we compared three methods for optimizing expected improvement in $\mathbb{R}^{17}$: random sampling, DIRECT (Jones et al., 1993), and CMA-ES (Hansen and Ostermeier, 1996). DIRECT and CMA-ES did not consistently find configurations with higher EIC than random sampling, and, even in cases where they did, the performance of the resulting configurator did not change significantly.) However, in discrete parameter configuration spaces as large as those we consider here (*e.g.*, $|\boldsymbol{\Theta}| = 1.38 \cdot 10^{37}$ in CPLEX), we can expect random sampling to perform poorly.

We thus introduce a simple local search strategy to optimize expected improvement in discrete parameter configuration spaces. Like PARAMILS, this method performs a local search in parameter configuration space based on a one-exchange neighbourhood, but here the objective function is deterministic: the expected improvement criterion (EIC) $E[I_{exp}(\boldsymbol{\theta})]$ (see Section 10.3.2). Note that it is not crucial to find the configuration with optimal EIC—finding a good local optimum quickly is typically sufficient in practice. More importantly, we need to keep the computational cost for EIC optimization low in order to enable SMBO to execute many iterations within the given time budget (where an SMBO iteration is one cycle through model fitting, selecting new configurations, and performing new runs in the intensification procedure). In order to compute EIC for a parameter configuration $\boldsymbol{\theta}$, we only require model predictions $\mu_{\boldsymbol{\theta}}$ and $\sigma^2_{\boldsymbol{\theta}}$, which are computationally quite cheap. However, with tens of thousands of evaluations of EIC, the cumulative computational cost can be substantial. For this reason, we only perform a few local searches from different starting points to optimize EIC. Better mechanisms (such as, *e.g.*, iterated local search) could doubtlessly be devised.

Procedure 12.1 details our new mechanism for selecting promising parameter configurations. We perform one local search starting at a random configuration, as well as ten local searches starting at previously-seen configurations with high EIC. To increase robustness of the EIC optimization, we still add $10\,000$ random configurations and then sort all configurations by EIC. Finally, we interleave random configurations in the list of promising configurations to be evaluated. In the subsequent intensification procedure (unchanged from SPO*'s Procedure 10.5 on page 165), we evaluate parameter configurations from that list for at least as much time as we spent learning the model and optimizing EIC. That procedure is guaranteed to evaluate at least one of the interleaved random configurations in each SMBO iteration.

12.3.2 Convergence of ACTIVECONFIGURATOR for Categorical Parameters

For finite parameter configuration spaces, $\boldsymbol{\Theta}$, we can prove a similar convergence result for ACTIVECONFIGURATOR as for FOCUSEDILS, re-using one Lemma devised for that proof (Lemma 9 on page 83), and replicating another one (Lemma 7) with minor changes (basically

Procedure 12.1: SelectNewParameterConfigurations($\mathcal{M}, \boldsymbol{\theta}_{inc}, \mathbf{R}$) in ACTIVECONFIGURATOR for discrete configuration spaces
num$_{ls}$ is a parameter of ACTIVECONFIGURATOR for discrete configuration spaces; we set it to num$_{ls}$=10 in all our experiments

Input : Model, $\mathcal{M}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$; sequence of target algorithm runs, $\mathbf{R}$
Output: Sequence of parameter configurations to evaluate, $\vec{\boldsymbol{\Theta}}_{new}$

// ===== Select configuration with LS starting at random $\boldsymbol{\theta}$
Select $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ uniformly at random
$\boldsymbol{\theta}_{lo} \leftarrow$ local optimum found by best-improvement local search in parameter configuration space, $\boldsymbol{\Theta}$, starting from $\boldsymbol{\theta}$ and optimizing $E[I_{exp}(\boldsymbol{\theta})]$ (see Section 10.3.2)
$\boldsymbol{\Theta}_{ls} \leftarrow \{\boldsymbol{\theta}_{lo}\}$

// ===== Select configurations with LS starting at configurations $\boldsymbol{\theta}$ with high $E[I_{exp}(\boldsymbol{\theta})]$
$\boldsymbol{\Theta}_{seen} \leftarrow \bigcup_{i=1}^{n} \{\mathbf{R}[i].\boldsymbol{\theta}\}$
$\boldsymbol{\Theta}_{start} \leftarrow$ num$_{ls}$ configurations $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{seen}$ with highest $E[I_{exp}(\boldsymbol{\theta})]$
for $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{start}$ **do**
    $\boldsymbol{\theta}_{lo} \leftarrow$ local optimum found by best-improvement local search in parameter configuration space, $\boldsymbol{\Theta}$, starting from $\boldsymbol{\theta}$ and optimizing $E[I_{exp}(\boldsymbol{\theta})]$
    $\boldsymbol{\Theta}_{ls} \leftarrow \boldsymbol{\Theta}_{ls} \cup \{\boldsymbol{\theta}_{lo}\}$

// ===== Select configurations at random, and sort all configurations by EI
$\boldsymbol{\Theta}_{rand} \leftarrow$ set of 10 000 elements drawn uniformly at random from $\boldsymbol{\Theta}$
Let $\vec{\boldsymbol{\Theta}}_{ei}$ be a list of all $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{ls} \cup \boldsymbol{\Theta}_{rand}$, sorted by decreasing $E[I_{exp}(\boldsymbol{\theta})]$

// ===== Interleave configurations with high EI and (new) random configurations
for $i = 1, \ldots,$ *length*$(\vec{\boldsymbol{\Theta}}_{ei})$ **do**
    Append $\vec{\boldsymbol{\Theta}}_{ei}[i]$ to $\vec{\boldsymbol{\Theta}}_{new}$
    Draw a parameter configuration $\boldsymbol{\theta}$ uniformly at random from $\boldsymbol{\Theta}$ and append it to $\vec{\boldsymbol{\Theta}}_{new}$
Let t_{ei} denote the total time spent in the current call of this procedure
return $[\vec{\boldsymbol{\Theta}}_{new}, t_{ei}]$

replacing ILS iterations with SMBO iterations).

Lemma 15 (Unbounded number of evaluations)**.** *Let $N(J, \boldsymbol{\theta})$ denote the number of runs* ACTIVECONFIGURATOR *has performed with parameter configuration $\boldsymbol{\theta}$ at the end of SMBO iteration J. Then, for any constant K and configuration $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ (with finite $\boldsymbol{\Theta}$),* $\lim_{J\to\infty} P[N(J, \boldsymbol{\theta}) \geq K] = 1$.

Proof. In each SMBO iteration, ACTIVECONFIGURATOR evaluates at least one random configuration (performing at least one new run for it), and with a probability of $p = 1/|\boldsymbol{\Theta}|$, this is configuration $\boldsymbol{\theta}$. Hence, the number of runs performed with $\boldsymbol{\theta}$ is lower-bounded by a binomial random variable $\mathcal{B}(k; J, p)$. Then, for any constant $k < K$ we obtain $\lim_{J\to\infty} \mathcal{B}(k; J, p) = \lim_{J\to\infty} \binom{J}{k} p^k (1-p)^{J-k} = 0$. Thus, $\lim_{J\to\infty} P[N(J, \boldsymbol{\theta}) \geq K] = 1$. □

Theorem 16 (Convergence of ACTIVECONFIGURATOR)**.** *When* ACTIVECONFIGURATOR *optimizes a cost measure c based on a consistent estimator $\hat{c}_N$ and a finite configuration space $\boldsymbol{\Theta}$, the probability that it finds the true optimal parameter configuration $\boldsymbol{\theta}^* \in \boldsymbol{\Theta}$ approaches one as the number of SMBO goes to infinity.*

Proof. According to Lemma 15, $N(\boldsymbol{\theta})$ grows unboundedly for each $\boldsymbol{\theta} \in \boldsymbol{\Theta}$. For each $\boldsymbol{\theta}_1$, $\boldsymbol{\theta}_2$, as $N(\boldsymbol{\theta}_1)$ and $N(\boldsymbol{\theta}_2)$ go to infinity, Lemma 9 states that in a pairwise comparison, the truly better configuration will be preferred. Thus eventually, ACTIVECONFIGURATOR visits all finitely many parameter configurations and prefers the best one over all others with probability arbitrarily close to one. □

We note that the same convergence result also holds for RANDOM* (a random search with the same intensification mechanism as ACTIVECONFIGURATOR, introduced in Section 11.5.1). We will thus rely on empirical results to distinguish the approaches (see Section 12.3.4).

12.3.3 Quantifying the Loss Due to Discretization

In order to apply some configuration procedures, such as PARAMILS and ACTIVECONFIGURATOR for categorical parameters, to optimize algorithms with (partially) numerical parameters, these numerical parameters need to be discretized. Such a discretization clearly comes at an opportunity cost since the search is restricted to a subspace that might not contain the optimal configuration. In principle, this opportunity cost could be arbitrarily large; in practice it depends on many factors, such as the nature of the parameter response, the granularity of the discretization, and the quality of the selected discrete parameter values. In order to estimate the magnitude of losses incurred by the discretization we used most prominently, we evaluated it for our SINGLEINSTCONT scenarios.

We report results for RANDOM* searching the continuous SAPS configuration space and the discretized subspace (recall from Section 3.2.1 that in this discretization we used seven values for each of the four parameters; this is the same discretization used throughout Part III of this thesis). Table 12.1 and Figure 12.3 compare the performance of the configurations found at the end of the search process. In six of seven cases restricting the search to a discrete subspace led to significant performance losses compared to search in the continuous space; the average performance loss was as large as 35% in the case of SAPS-SWGCP. This clearly demonstrates that for algorithms with few continuous parameters a search in the continuous space can yield much better results than in a discretized subspace.

Next, we studied the impact of discretization on our ACTIVECONFIGURATOR variants, quantifying their loss incurred by restricting the search to a discrete subspace instead of the original continuous space. Figure 12.4 shows the results. For both RF and approximate GP models, restricting the search space to discrete parameters clearly worsened performance. However, we note that this change had a larger impact when using GP models. In particular, for continuous optimization AC(GP) outperformed AC(RF) in scenarios SAPS-QWH, SAPS-QCP-Q095, and SAPS-SWGCP-Q095 (see also Section 11.5.2). In contrast, for discrete optimization, AC(RF) performed as good as AC(GP), and even better in the case of SAPS-SWGCP-Q095. We attribute this to the better adaptation of RF models to categorical inputs as described in Section 12.2.1.

Even though the discretization of parameters led to the quite substantial opportunity costs described in this section, we employed it in the remainder of this thesis in order to allow a direct comparison of our SMBO methods to PARAMILS. In future work, we plan to extend ACTIVECONFIGURATOR with a mechanism for mixed continuous/discrete optimization of

Scenario	Continuous	Discretized	p-value
SAPS-QWH [$\cdot 10^3$]	$\mathbf{9.93 \pm 0.44}$	10.54 ± 0.36	$\mathbf{1.1 \cdot 10^{-4}}$
SAPS-QCP-MED [$\cdot 10^{-2}$]	$\mathbf{4.21 \pm 0.18}$	4.96 ± 0.25	$\mathbf{3.6 \cdot 10^{-8}}$
SAPS-QCP-Q075	$\mathbf{0.19 \pm 0.01}$	0.23 ± 0.02	$\mathbf{3.7 \cdot 10^{-7}}$
SAPS-QCP-Q095	2.62 ± 1.36	$\mathbf{2.27 \pm 0.63}$	0.86
SAPS-SWGCP-MED	$\mathbf{0.17 \pm 0.02}$	0.23 ± 0.01	$\mathbf{2.0 \cdot 10^{-6}}$
SAPS-SWGCP-Q075	$\mathbf{0.22 \pm 0.02}$	0.29 ± 0.01	$\mathbf{3.1 \cdot 10^{-8}}$
SAPS-SWGCP-Q095	$\mathbf{0.26 \pm 0.06}$	0.32 ± 0.05	$\mathbf{8.7 \cdot 10^{-5}}$

Table 12.1: Quantification of loss due to discretization for SINGLEINSTCONT scenarios. We performed 25 runs of RANDOM* using the continuous and discretization SAPS configuration spaces and computed their test performance $p_{test,t}$ (PAR over $N = 1\,000$ test runs using the methods' final incumbents $\boldsymbol{\theta}_{inc}(t)$) for a configuration time of $t = 1\,800$s. Here, we give mean $\pm$ standard deviation across the 25 runs and a p-value for a Mann Whitney U test for differences between the performances. Figure 12.3 visualizes this data.

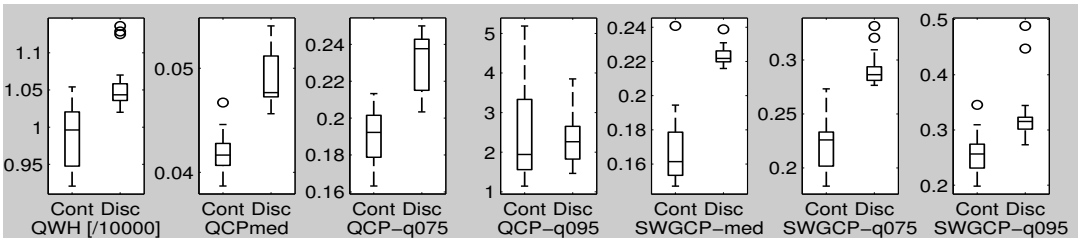


Figure 12.3: Quantification of loss due to discretizing parameters, for RANDOM* on the SINGLEINSTCONT scenarios. We ran RANDOM* for optimizing the original, continuous, SAPS parameters ("Cont") and for optimizing its discretized parameters ("Disc"). We performed 25 runs of the configurators and show boxplots of their test performance $p_{test,t}$ for a time budget of $t = 1\,800$ seconds.

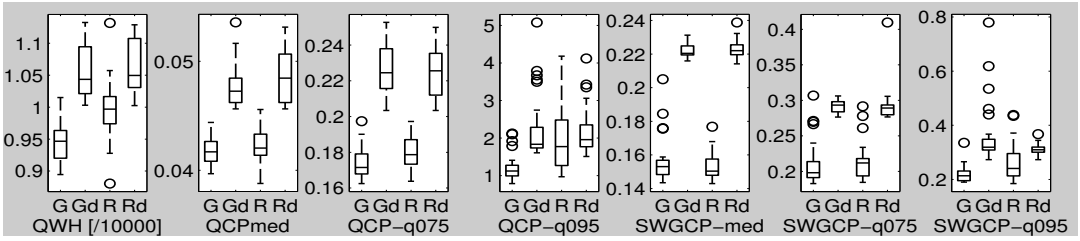


Figure 12.4: Quantification of Loss Due to Discretizing Parameters in SMBO. We ran configurators AC(RF) ("R") and AC(GP) ("G") on the SINGLEINSTCONT scenarios with the original continuous SAPS parameters, as well as AC(RF) ("Rd") and AC(GP) ("Gd") using discretized SAPS parameters. We performed 25 runs of the configurators and show boxplots of their test performance $p_{test,t}$ for a time budget of $t = 1\,800$ seconds.

Scenario	AC(RF)	AC(GP)	RANDOM*	FocusedILS	significant
SAPS-QWH [$\cdot 10^4$]	$\mathbf{1.04 \pm 0.04}$	1.05 ± 0.04	1.05 ± 0.05	1.06 ± 0.05	–
SAPS-QCP-MED [$\cdot 10^{-2}$]	$\mathbf{4.78 \pm 0.13}$	4.90 ± 0.27	4.92 ± 0.20	5.15 ± 0.41	1/3, 1/4, 2/4
SAPS-QCP-Q075	$\mathbf{0.22 \pm 0.01}$	$\mathbf{0.22 \pm 0.01}$	0.23 ± 0.01	0.24 ± 0.02	2/4
SAPS-QCP-Q095	2.37 ± 1.01	$\mathbf{2.17 \pm 0.91}$	2.20 ± 0.82	2.93 ± 3.15	–
SAPS-SWGCP-MED	$\mathbf{0.22 \pm 0.00}$	$\mathbf{0.22 \pm 0.01}$	0.23 ± 0.01	0.27 ± 0.12	1/4, 2/4
SAPS-SWGCP-Q075	$\mathbf{0.29 \pm 0.02}$	0.30 ± 0.05	$\mathbf{0.29 \pm 0.01}$	0.35 ± 0.08	1/4, 2/4, 3/4
SAPS-SWGCP-Q095	0.36 ± 0.14	0.35 ± 0.10	$\mathbf{0.32 \pm 0.03}$	0.40 ± 0.19	3/4
SPEAR-SWV-MED	$\mathbf{0.61 \pm 0.01}$	$\mathbf{0.61 \pm 0.01}$	0.62 ± 0.01	0.62 ± 0.01	1/2, 1/3, 1/4
SPEAR-SWV-Q075	$\mathbf{0.60 \pm 0.11}$	0.79 ± 0.66	0.63 ± 0.09	0.78 ± 0.50	1/3, 1/4, 2/4
SPEAR-SWV-Q095	$\mathbf{0.87 \pm 0.08}$	$\mathbf{0.87 \pm 0.05}$	0.88 ± 0.07	1.00 ± 0.28	1/2, 1/3, 1/4
SPEAR-IBM-Q025	$\mathbf{0.63 \pm 0.01}$	0.64 ± 0.01	0.64 ± 0.01	0.65 ± 0.01	1/2, 1/3, 1/4, 3/4
SPEAR-IBM-MED	$\mathbf{3.35 \pm 0.59}$	3.53 ± 0.75	3.64 ± 0.93	9.97 ± 8.88	1/3, 1/4, 2/4, 3/4

Table 12.2: Quantitative comparison of configurators for SINGLEINSTCAT and discretized SINGLEINSTCONT scenarios. We performed 25 runs of the configurators and computed their test performance $p_{test,t}$ (PAR over $N = 1\,000$ test instances using the methods' final incumbents $\boldsymbol{\theta}_{inc}(t)$) for a configuration time of $t = 1\,800$s. Here, we give mean $\pm$ standard deviation across the 25 runs. Column "significant" lists the pairs of configurators for which a Mann Whitney U test judged the performance difference to be significant with confidence level 0.05; '1' stands for AC(RF), '2' for AC(GP), '3' for RANDOMSEARCH, and '4' for FOCUSEDILS. Figure 12.7 visualizes this data.

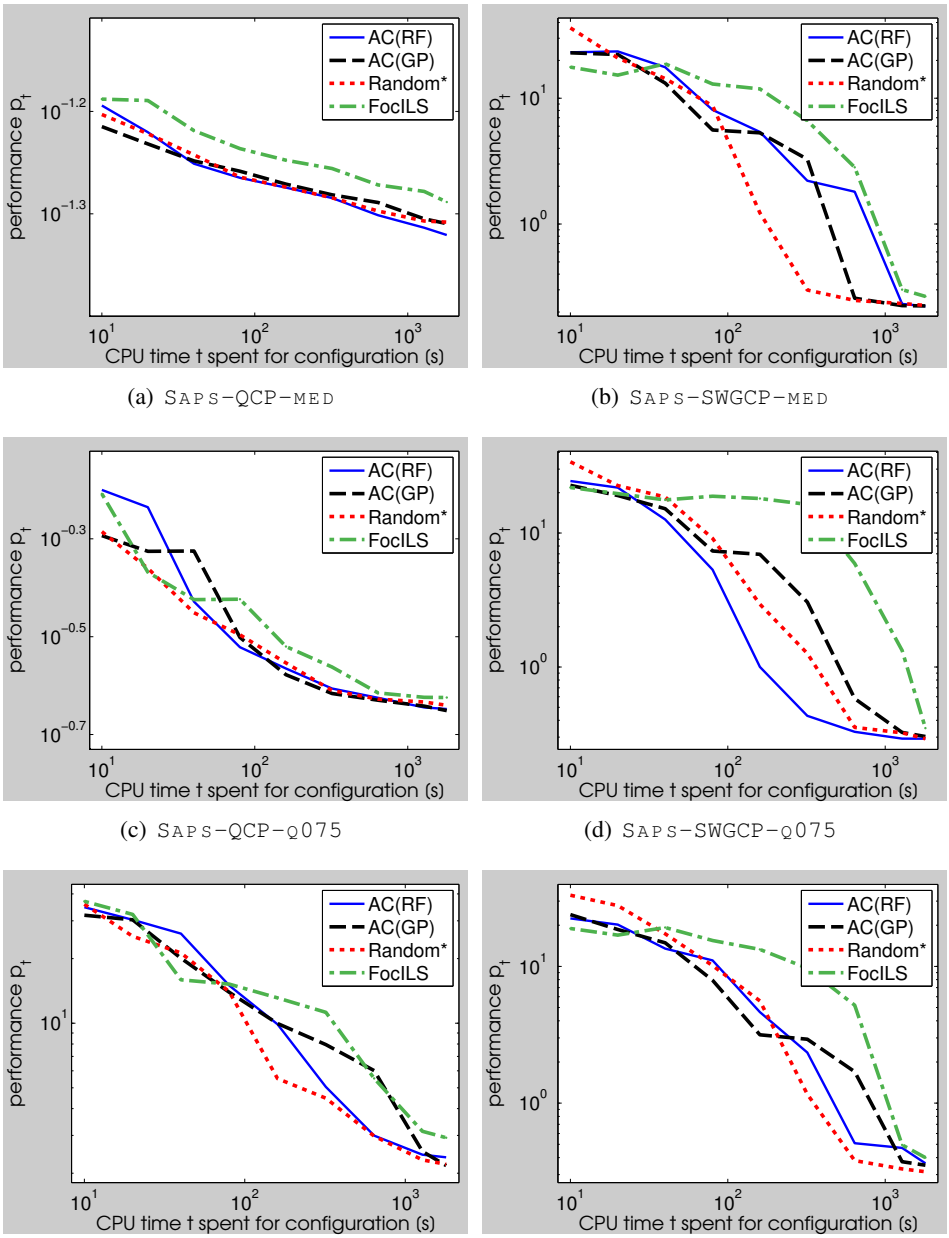
EIC and thereby enable it to optimize mixed continuous/discrete parameters without the need for a discretization.

12.3.4 Experimental Evaluation of Configuration Procedures

We now experimentally compare configuration procedures for the configuration of algorithms with categorical parameters: AC(RF), AC(GP), and, for reference, RANDOM* and PARAMILS. For this evaluation, we used the seven SINGLEINSTCONT scenarios with discretized SAPS parameters and the five SINGLEINSTCAT scenarios considered throughout this chapter. We performed 25 runs of each configuration procedure for each scenario and evaluated test performance (SAPS and SPEAR penalized average runtime) for various time budgets.

We present the results of this comparison in Figures 12.5 and 12.6 (plotting mean test performance for varying time budgets), and Figure 12.7 and Table 12.2 (comparing test performances at the end of the trajectory). Overall, AC(RF) performed best in this comparison, followed by AC(GP) and RANDOM* (about tied), and then FOCUSEDILS.

For the SINGLEINSTCONT configuration scenarios (see Figure 12.5), in all of which the target algorithm, SAPS, has only 4 parameters, AC(RF) and RANDOM* performed comparably, each of them being fastest to find good configurations for some scenarios. Some of the 25 FOCUSEDILS yielded rather poor performance, causing poor mean performance for short time budgets. This was especially the case for the SWGCP configuration scenarios (see right column of Figure 12.5) and SAPS-QWH (see Figure 12.6(a)). As shown in Figure 12.7(a), some FOCUSEDILS runs also yielded poor performance at the end of the search process (after $t = 1\,800$ seconds), and in 5 of the 7 SINGLEINSTCONT scenarios, FOCUSEDILS was

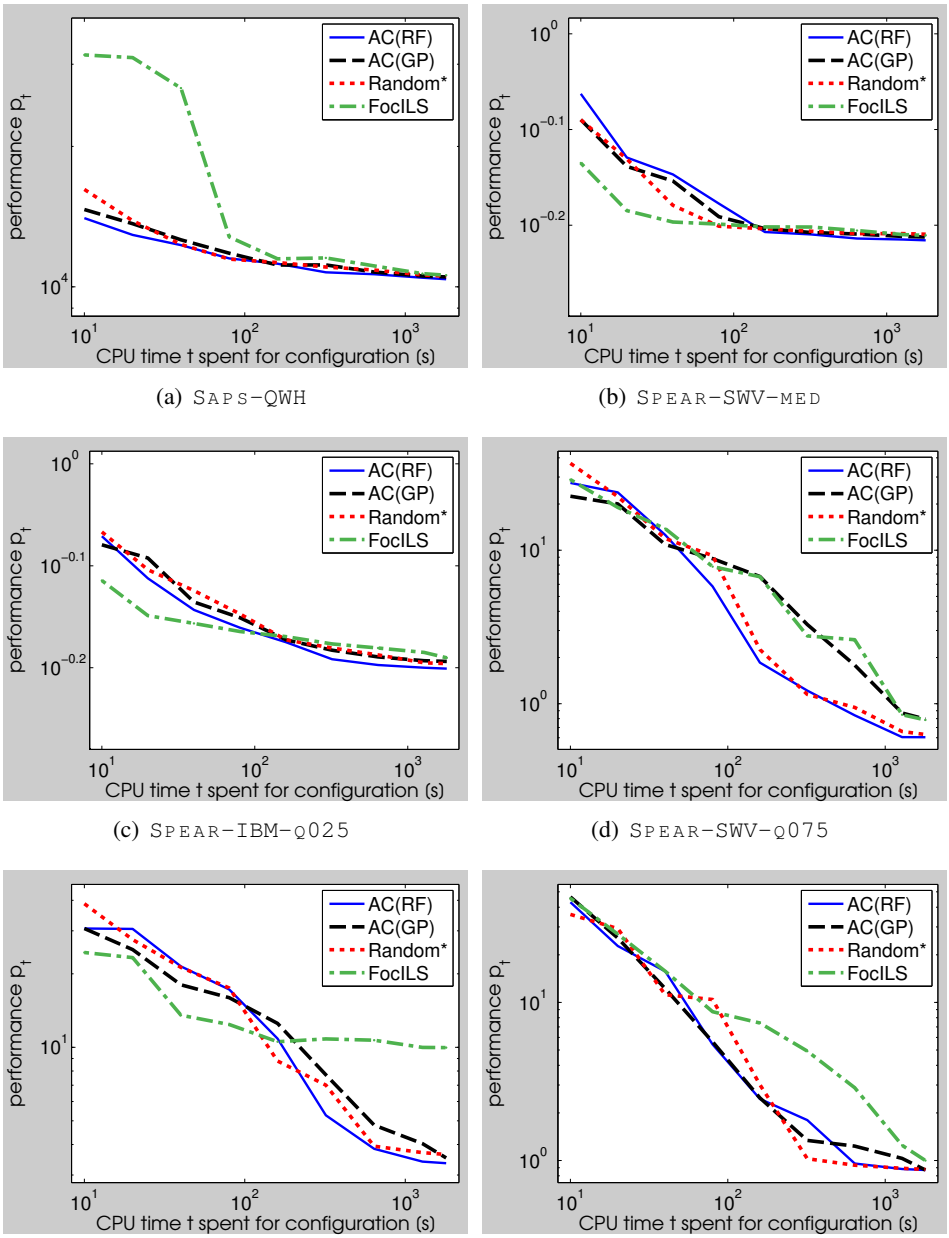


(a) SAPS-QCP-MED (b) SAPS-SWGCP-MED

(c) SAPS-QCP-Q075 (d) SAPS-SWGCP-Q075

(e) SAPS-QCP-Q095 (f) SAPS-SWGCP-Q095

Figure 12.5: Comparison of configurators for SINGLEINSTCONT scenarios (except SAPS-QWH) with discretized parameters. We performed 25 runs of the configurators and computed their test performance $p_{test,t}$ at times $t = 10, 20, 40, \ldots, 1280, 1800$. We plot mean $p_{test,t}$ across the 25 runs.



(a) SAPS-QWH

(b) SPEAR-SWV-MED

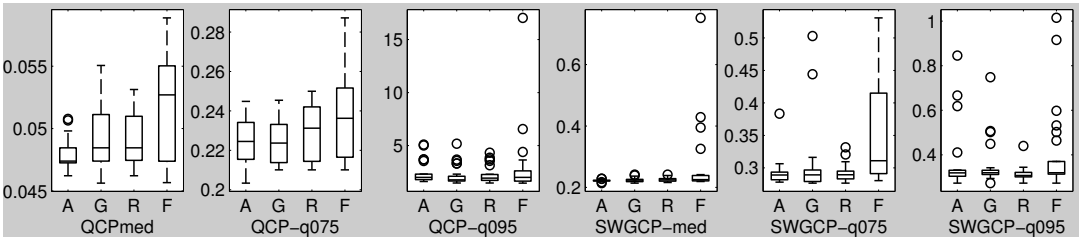
(c) SPEAR-IBM-Q025

(d) SPEAR-SWV-Q075

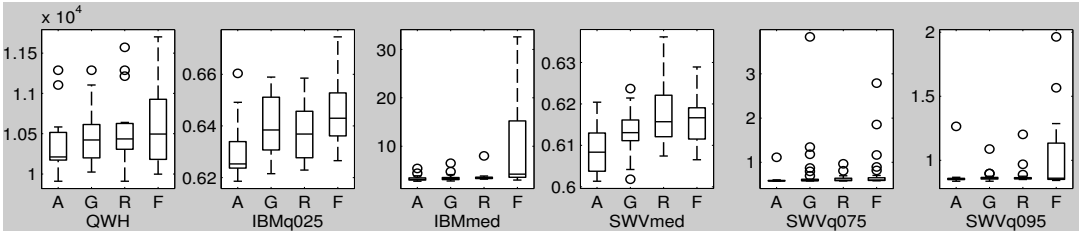
(e) SPEAR-IBM-MED

(f) SPEAR-SWV-Q095

Figure 12.6: Comparison of configurators for SAPS-QWH with discretized parameters and SINGLEINSTCAT scenarios. We performed 25 runs of the configurators and computed their test performance $p_{test,t}$ at times $t = 10, 20, 40, \ldots, 1280, 1800$. We plot mean $p_{test,t}$ across the 25 runs.



(a) SINGLEINSTCONT scenarios except SAPS-QWH, with discretized domains



(b) SAPS-QWH with discretized domains, and SINGLEINSTCAT scenarios

Figure 12.7: Boxplot comparison of configuration procedures for setting categorical parameters for single instances. 'A' stands for AC(RF), 'G' for AC(GP), 'R' for RANDOM*, and 'F' for FocusedILS. We show boxplots for the data presented in Table 12.2.

statistically significantly worse than at least one of the other methods. The only statistically significant difference between the three other methods was that AC(RF) performed better than RANDOM* for scenario SAPS-QCP-MED.

For the SINGLEINSTCAT configuration scenarios (see Figure 12.6), in all of which the target algorithm, SPEAR, has 26 parameters, we note some different trends. First of all, for these scenarios, the local-search-based method FOCUSEDILS found good configurations fastest in 3 of the 5 configuration scenarios, SPEAR-SWV-MED, SPEAR-IBM-Q025, and SPEAR-IBM-MED (see Figures 12.6(b), 12.6(c), and 12.6(e)). However, after 100 seconds the other methods caught up and tended to perform better. Indeed, at the end of the search process, AC(RF) yielded significantly better mean performance than FOCUSEDILS for all of the five SINGLEINSTCAT scenarios. Performance differences were rather substantial for some configuration scenarios, most notably SPEAR-IBM (where some FOCUSEDILS runs yielded parameter configurations with very poor performance, timing out in a large fraction of runs). Notably, even though AC(RF) only performed significantly better than RANDOM* for 1 of the 7 SAPS configuration scenarios, it performed significantly better for *all* the 7 SPEAR configuration scenarios (although the performance differences were small). This clearly demonstrates that random search does not search higher-dimensional configuration spaces as effectively as SMBO methods guided by a good model. However, at the same time, if there are many very good parameter configurations (as seems to be the case for SPEAR in these scenarios), then a simple random search with a strong good intensification strategy can yield good configurations. The intensification mechanism used in ACTIVECONFIGURATOR and RANDOM* (unchanged from SPO*'s

Procedure 10.5) appears to be superior to that of FOCUSEDILS, which would explain the significant and sometimes substantial advantage of these methods.

12.4 Chapter Summary

In this chapter, we extended the scope of sequential model-based optimization (SMBO) to the configuration of algorithms with *categorical* parameters. First, we extended our random forest (RF) and approximate Gaussian process (GP) models to handle categorical inputs. The quality of approximate GP models deteriorated somewhat when trained on categorical inputs, whereas the quality of the RF models improved. Hence, in contrast to what we observed in the previous chapter for continuous inputs, on average, RF models now yielded higher model quality.

Next, we introduced configuration procedures that optimize categorical parameters using these models. In order to optimize the expected improvement criterion (EIC) across a discrete parameter configuration space, these procedures—variants of ACTIVECONFIGURATOR (AC)—apply a local search in parameter configuration space. We also proved that for finite configuration spaces, AC converges to the optimal configuration (based on any type of model). In our experimental results for the configuration of discretized parameters of SAPS and SPEAR on single problem instances, AC based on RF models performed best, followed by AC based on GP models and RANDOM* (about tied with AC(GP)).

All methods performed clearly better than FOCUSEDILS. This is especially significant since—in contrast to previous chapters—all methods in this comparison search the same discretized configuration space.

Chapter 13

Extensions II: Multiple Instances

A man should look for what is, and not for what he thinks should be.
—Albert Einstein, German American theoretical physicist

The sequential model-based optimization (SMBO) methods we have introduced so far only deal with the optimization of algorithm parameters for single problem instances. In this chapter, we remove that restriction, extending SMBO to handle general algorithm configuration problems defined across *sets of problem instances*.

There are multiple ways in which we could extend SMBO to multiple instances. One possible straightforward extension is along the same lines as BASICILS in Chapter 5: simply perform runs on the same N instances (or, to be more precise, the same N $\langle$instance, seed$\rangle$ pairs) to evaluate each configuration. However, as we showed in our empirical analysis in Chapter 4 and in the evaluation of BASICILS against FOCUSEDILS in Chapter 5, there exists no single best fixed N. Instead, it is more effective to discard poor configurations quickly and evaluate the best configurations using additional runs, as we did in FOCUSEDILS or in the intensification mechanisms of the SMBO variants introduced in the previous chapters. However, once we use different instances for the evaluation of different configurations, we face the question of how to integrate that information into our models.

A second possible way of extending SMBO to multiple instances is to simply build models ignoring the information about which instance was used in which algorithm run. This approach treats the additional inter-instance variance as additional (extraneous) noise. Even if the model ignores the existence of different problem instances, the comparison between parameter configurations can still apply a blocking scheme that *does* take into account which instances (and seeds) each configuration has been run on. As we will see in Section 13.6.2, this approach can lead to effective algorithm configuration procedures.

A final possibility is to explicitly integrate information about the used instances into our response surface models. If we have access to a vector of *features* $\boldsymbol{x}_i$ describing each training problem instance $\pi_i \in \Pi$, we can learn a joint model that predicts algorithm runtime for combinations of parameter configurations and instance features, and then aggregates these predictions to yield a predictive distribution for the cost measure to be optimized. Many different types of instance features can be used. One simple feature that can always be computed is the runtime of an algorithm (*e.g.*, the algorithm we are optimizing, with some

fixed parameter configuration, such as the default) for the problem instance. If it too expensive to run the target algorithm for each instance, an alternative trivial feature is the *index i* of an instance in an ordered set of training instances. Finally, for the main problems we study in this thesis—propositional satisfiability (SAT) and mixed integer programming (MIP)—there exist collections of features that have been show to correlate with instance hardness and can be computed in time polynomial in the problem size.

In this chapter, we study the quality of models and configuration procedures, based on these different sets of instance features. We first review the SAT and MIP features we use and then introduce a simple extension of our random forest (RF) models to handle the prediction of cost measures defined across multiple instances. We compare model quality based on the different feature sets, introduce a new intensification mechanism that blocks on instances and random number seeds, and evaluate configuration procedures based on this new mechanism and the different instance sets.

13.1 Instance Features

Existing work on empirical hardness models (Leyton-Brown et al., 2002; Nudelman et al., 2004; Hutter et al., 2006; Xu et al., 2007a; Leyton-Brown et al., 2009) convincingly demonstrated that it is possible to predict algorithm runtime based on features of the problem instance to be solved. In particular, that work has shown that—based on a vector of polytime-computable instance features—it is possible to gain relatively accurate predictions of the runtime of algorithms for solving SAT instances and MIP-encoded instances of the winner determination problem in combinatorial auctions. These predictions have been exploited to construct portfolio-based algorithm selection mechanisms (Leyton-Brown et al., 2003a,b; Xu et al., 2007b, 2008). In particular, the portfolio-based algorithm selection approach SATzilla—based on such predictions—has repeatedly won multiple categories of the bi-annual SAT competition, demonstrating the maturity of the approach.

In the context of algorithm configuration, instance features can incorporate any kind of information that is available for all *training* instances. In algorithm configuration, the aim is to find the best-performing configuration for the training set, without any knowledge of the test set. At test time, this fixed configuration is then evaluated on previously-unseen test instances. Thus, we never require access to features of test instances—the same configuration is used for *any* test instance. Thus, in contrast to per-instance approaches, such as SATzilla, in our setting it suffices to have features for the training instances; in particular, we do not require an executable procedure that computes features for a given problem instance. Indeed, instance features could, for example, be the result of an extensive offline analysis of instances. Correspondingly, we treat features for the training instances as a (potentially empty) user-defined input to model-based algorithm configuration procedures.

13.1.1 SAT

For SAT instances in the form of CNF formulae, we used the features that have been constructed for the 2009 version of SATzilla (Xu et al., 2009). Figure 13.1 lists these 126 features. 84 of these features were originally introduced by Nudelman et al. (2004), and the others

were added by Xu et al. (2008, 2009). In the final 2009 version of SATzilla, not all these features were considered because some of them can be computationally expensive. Note that in per-instance approaches, this computational expense needs to be limited since it counts as part of the ("target algorithm") runtime for solving an instance. In contrast, in algorithm configuration instance features can be computed offline for a set of training instances, once and for all. Thus, we employed the full feature set, relying on the random forest model to select the best features. Here, we report experiments for the QCP and SWGCP instance sets (see Section 3.3). For these, the feature computation took 30 seconds on average (without LP features: 9.4 seconds), with a maximum of 241 seconds (without LP features: 172 seconds).

13.1.2 MIP

We constructed our MIP features by adapting and extending a set of existing features for instances of the winner determination problem (WDP) in combinatorial auctions. These features were introduced by Leyton-Brown et al. (2002). For our experiments with MIP-encoded WDP instances, we could have used these original WDP features as is. However, in order to be able to handle general MIP instances (such as the benchmark sets CLS, MJA, MIK, and QP in our CPLEX configuration scenarios, see Section 3.3.2), we generalized these features for general representations of MIP instances. In particular, we use functions provided by the commercial MIP solver CPLEX in order to compute these features. This enables us to compute features for *any* problem CPLEX can read and side-steps the necessity for parsing MIP instances with a separate piece of feature computation code (which is not simple due to various standards for representing MIP instances).

We computed our features using CPLEX 10.1.1, which uses the rather general representation of MIP instances discussed in Section 3.1.2 and replicated here for convenience:

$$\begin{aligned} \text{minimize} \quad & 1/2 \cdot \boldsymbol{x}^\intercal \cdot \boldsymbol{Q} \cdot \boldsymbol{x} + \boldsymbol{c}^\intercal \cdot \boldsymbol{x} \\ \text{subject to} \quad & \boldsymbol{A} \cdot \boldsymbol{x} \bowtie \boldsymbol{b} \\ & \boldsymbol{a}_i^\intercal \cdot \boldsymbol{x} + \boldsymbol{x}^\intercal \cdot \boldsymbol{Q}_i' \cdot \boldsymbol{x} \leq r_i \text{ for } i = 1, \ldots, q \\ & l_i \leq x_i \leq u_i \text{ for } i = 1, \ldots, n \\ & x_i \text{ is integer for } i \text{ in a subset of } \{1, \ldots, n\}. \end{aligned} \tag{13.1}$$

The instance features we computed concern mostly the linear constraint matrix, $\boldsymbol{A}$.

In Figure 13.2, we list the instance features we compute. Features 1–2, 4–23, and 25–27 are straightforward generalizations of the WDP features by Leyton-Brown et al. (2002) to the general case of MIP (each WDP bid b_i has a MIP variable x_i associated with it, and the price of the bid is the linear coefficient of x_i in the objective function, c_i). We omitted five of the original WDP features that did not prominently appear in the subsets of most predictive features in the analysis carried out by Leyton-Brown et al. (2009); these five features were the "node eccentricity statistics", the "average minimum path length", and the "ratio of clustering coefficient and average minimum path length". We also added a number of additional features. Feature 3 counts the number of nonzero elements of $\boldsymbol{A}$, a measure for problem size in a sparse problem encoding. Feature 24 is the objective function value achieved with an LP

Problem Size Features:

1.–2. **Number of variables and clauses in original formula**: denoted v and c, respectively

3.–4. **Number of variables and clauses after simplification with SATelite**: denoted v' and c', respectively

5.–6. **Reduction of variables and clauses by simplification**: (v-v')/v' and (c-c')/c'

7. **Ratio of variables to clauses**: v'/c'

Variable-Clause Graph Features:

8.–12. **Variable node degree statistics:** mean, variation coefficient, min, max, and entropy

13.–17. **Clause node degree statistics:** mean, variation coefficient, min, max, and entropy

Variable Graph Features:

18–21. **Node degree statistics:** mean, variation coefficient, min, and max

22.–26. **Diameter:** mean, variation coefficient, min, max, and entropy

27.–31. **Clustering Coefficient:** mean, variation coefficient, min, max, and entropy

Clause Graph Features:

32–36. **Node degree statistics:** mean, variation coefficient, min, max, and entropy

Balance Features:

37.–41. **Ratio of positive to negative literals in each clause:** mean, variation coefficient, min, max, and entropy

42.–46. **Ratio of positive to negative occurrences of each variable:** mean, variation coefficient, min, max, and entropy

47.–49. **Fraction of unary, binary, and ternary clauses**

Proximity to Horn Formula:

50. **Fraction of Horn clauses**

51.–55. **Number of occurrences in a Horn clause for each variable:** mean, variation coefficient, min, max, and entropy

DPLL Probing Features:

56.–60. **Number of unit propagations:** computed at depths 1, 4, 16, 64 and 256

61.–62. **Search space size estimate:** mean depth to contradiction, estimate of the log of number of nodes

LP-Based Features:

63.–66. **Integer slack vector:** mean, variation coefficient, min, and max

67. **Ratio of integer variables in LP solution**

68. **Objective function value of LP solution**

Local Search Probing Features, based on 2 seconds of running each of SAPS and GSAT:

69.–78. **Number of steps to the best local minimum in a run:** mean, median, variation coefficient, 10th and 90th percentiles

79.–82. **Average improvement to best in a run:** mean and coefficient of variation of improvement per step to best solution

83.–86. **Fraction of improvement due to first local minimum:** mean and variation coefficient

87.–90. **Coefficient of variation of the number of unsatisfied clauses in each local minimum:** mean and variation coefficient

Clause Learning Features (based on 2 seconds of running Zchaff_rand):

91.–99. **Number of learned clauses:** mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

100.–108. **Length of learned clauses:** mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

Survey Propagation Features

109.–117. **Confidence of survey propagation:** For each variable, compute the higher of $P(true)/P(false)$ or $P(false)/P(true)$. Then compute statistics across variables: mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

118.–126. **Unconstrained variables:** For each variable, compute $P(unconstrained)$. Then compute statistics across variables: mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

Figure 13.1: 11 groups of SAT features; these were introduced by Nudelman et al. (2004) and Xu et al. (2008, 2009).

Problem Size Features:

1.–2. **Number of variables and constraints**: denoted n and m, respectively

3. **Number of nonzero entries in the linear constraint matrix, $\boldsymbol{A}$**

Variable-Constraint Graph Features:

4–7. **Variable node degree statistics**: mean, max, min, and stddev

8–11. **Constraint node degree statistics**: mean, max, min, and stddev

Variable Graph (VG) Features:

12–17. **Node degree statistics**: max, min, stddev, 25% and 75% quantiles

18–19. **Clustering Coefficient:** mean and stddev

20. **Edge Density**: number of edges in the VG divided by the number of edges in a complete graph having the same number of nodes

LP-Based Features:

21–23. **Integer slack vector:** mean, max, L_2 norm

24. **Objective function value of LP solution**

Objective Function Features:

25. **Standard deviation of normalized coefficients:** $\{c_i/m\}_{i=1}^{n}$

26. **Standard deviation of** $\{c_i/n_i\}_{i=1}^{n}$, where n_i denotes the number of nonzero entries in column i of $\boldsymbol{A}$

27. **Standard deviation of** $\{c_i/\sqrt{n_i}\}_{i=1}^{n}$

Linear Constraint Matrix Features:

28.–29. **Distribution of normalized constraint matrix entries,** $\boldsymbol{A}_{i,j}/b_i$: mean and stddev (only of elements where $b_i \neq 0$)

30.–31. **Variation coefficient of normalized absolute nonzero entries per row**: mean and stddev

Variable Type Features:

32.–33. **Support size of discrete variables**: mean and stddev

34. **Percent unbounded discrete variables**

35. **Percent continuous variables**

General Problem Type Features:

36. **Problem type**: categorical feature attributed by CPLEX (LP, MILP, FIXEDMILP, QP, MIQP, FIXEDMIQP, MIQP, QCP, or MIQCP)

37. **Number of quadratic constraints**

38. **Number of nonzero entries in matrix of quadratic coefficients of objective function, $\boldsymbol{Q}$**

39. **Number of variables with nonzero entries in $\boldsymbol{Q}$**

Figure 13.2: Eight groups of features for the mixed integer programming problem.

relaxation; this feature is most likely only useful when problem instances within a distribution have comparable objective function values.

Our *linear constraint matrix* features (28–31) capture information about the coefficients of the linear constraint matrix, $\boldsymbol{A}$. For features 28–29, we compute the normalized coefficients, $\boldsymbol{A}'_{i,j} = \boldsymbol{A}_{i,j}/b_i$ (defined only for rows with $b_i \neq 0$) and use mean and standard deviation of $\{\boldsymbol{A}'_{i,j} | b_i \neq 0\}$ (set to 0 if that set it empty). Features 30-31 capture how much coefficients vary within each linear constraint. We compute the variation coefficient v of the nonzero absolute entries of each row of $\boldsymbol{A}$ and use its mean and standard deviation across rows.

Our *variable type* features (32–35) include the range of discrete variables (always 2 for the binary WDP variables), the ratio of continuous variables, as well as the ratio of unbounded discrete variables.

Finally, our *general problem type* features (36–39) include a categorical problem type defined by CPLEX and three features concerning quadratic constraints and quadratic terms in the objective function. The computation of these 39 features for instances in benchmark set `Regions100` took 4.4 seconds on average, with a maximum of 8.2 seconds.

13.1.3 Generic Features

Even in cases where no features exist for distinguishing the training instances, we can construct generic features. One straightforward and often rather characteristic feature for instance π is the performance of a given algorithm on π. If multiple algorithms are available, we can gather the performance of these algorithms on all training instances. In order to study a generic feature that can always be computed, here we use the performance of the target algorithm's default configuration as an instance feature. In cases where this data is not available and is too expensive to gather, we can use a trivial feature for instance π_i: its integer-valued *index* i. We encode this generally-available feature as a categorical variable with domain $1, \ldots, |\Pi|$. As we demonstrate in Section 13.4.1, using these generic features can strongly improve model quality compared to using no features at all.

13.1.4 Principal Component Analysis (PCA) to Speed Up Learning

When a large number of instance features is available, learning a random forest (RF) model can be slow. While RF models are known to perform implicit feature selection (Hastie et al., 2009), learning time scales linearly in the sum of number of parameters and features. (This is at least the case in our implementation, where we select a constant fraction of variables for each split. If we were to select a constant number instead, learning time would effectively be independent of the number of parameters and features.)

To shorten the time required for model learning, we reduce the number of features by applying *principal component analysis* (*PCA*, see, *e.g.*, Hastie et al., 2009). PCA identifies so-called principal components, orthogonal directions in which the data—here, the matrix $(\boldsymbol{x}_1 \cdots \boldsymbol{x}_{|\Pi|})$ of features for all training instances—has maximal variance. Projecting the data into the space spanned by the first k principal components is an effective way to preserve much of the variance while reducing dimensionality. The number of principal components to use, k, is an algorithm parameter that affects model quality and the time complexity of model building (linear in k). In Section 13.4.2, we evaluate various choices of k.

Instead of applying PCA, we could have used feature selection methods (Guyon and Elisseeff, 2003), such as forward selection as done in SATzilla (Nudelman et al., 2004; Xu et al., 2008). While PCA only takes into account the variation in the features, such feature selection methods also take into account the response variable (here: runtime). Their drawback, however, is computational efficiency: PCA only requires the construction of a single RF model, while forward selection would need to construct a RF model to evaluate each considered subset of features. (Note that in linear regression models, such as used in SATzilla, this can be sped up by using rank-1 updates; that is not the case for RF models.) In future work, it appears promising to apply computationally more expensive feature selection steps in regular intervals throughout the algorithm configuration process.

13.2 Modelling Cost Measures Defined Across Multiple Instances

So far, in this thesis we have only discussed models that are trained using pairs $(\boldsymbol{\theta}_i, o_i)$ of parameter configurations and their performance in single algorithm runs. Now, we extend these inputs to the model building procedure to include instance features. Let $\boldsymbol{x}_i$ denote the column vector of features for training instance $\pi_i \in \Pi$. Our training data is then $\{(\boldsymbol{\theta}_i, \boldsymbol{x}_i, o_i)\}_{i=1}^n$. Combining parameter values, $\boldsymbol{\theta}_i$, and instance features, $\boldsymbol{x}_i$, into one input vector yields the pairs of training data $\{([\boldsymbol{\theta}_i^\intercal \ \boldsymbol{x}_i^\intercal]^\intercal, o_i)\}_{i=1}^n$. Denoting the number of parameters as d and the number of instance features as e, the dimensionality of each training data point is $d + e$.

Given these $d + e$-dimensional training data points, we aim to construct models that take as input only a parameter configuration $\boldsymbol{\theta}$ (a column vector of length d) and predict its cost measure, $c(\boldsymbol{\theta})$, across multiple instances.

13.2.1 Gaussian Processes: Existing Work for Predicting Marginal Performance Across Instances

One particular cost measure of interest is *expected performance* across instances and multiple runs of a randomized algorithm. This cost measure was previously studied by Williams et al. (2000) in the context of Gaussian process regression. In particular, they considered the problem of minimizing a (noise-free) blackbox function f with $d + e$ inputs, where d inputs are *control* variables $\boldsymbol{X}_C$ and the remaining e inputs are uncontrollable *environmental* variables $\boldsymbol{X}_E$. The objective in their work was to find the setting $\boldsymbol{x}_C$ of control variables $\boldsymbol{X}_C$ minimizing the *marginal*

$$c(\boldsymbol{x}_C) = \sum_{\boldsymbol{x}_E \in \boldsymbol{X}_E} w(\boldsymbol{x}_E) \cdot f(\boldsymbol{x}_C, \boldsymbol{x}_E)$$

across settings $\boldsymbol{x}_E$ of the environmental variables $\boldsymbol{X}_E$, where $w(\cdot)$ is a weighting function with $\sum_{\boldsymbol{x}_E \in \boldsymbol{X}_E} w(\boldsymbol{x}_E) = 1$. The environmental variables in this problem formulation correspond exactly to instance features in our case. Williams et al. fitted a noise-free GP model to the $d + e$ inputs, assuming a so-called uninformative prior distribution for the variance of the Gaussian stochastic process (instead of optimizing this variance as a hyper-parameter, as we did when using GP models in the previous chapters). This led to predictions of $f(\boldsymbol{x}_C, \boldsymbol{x}_E)$ that follow a Student t distribution. Exploiting the fact that linear combinations of Student t-distributed random variables are still Student t-distributed, Williams et al. (2000) showed that the marginal predictions of this noise-free GP model for $c(\boldsymbol{x}_C)$ are also Student t-distributed. They then used a standard expected improvement criterion (adapted to Student t instead of Gaussian distributions) to decide which setting of the control variables to select next. They also introduced a mechanism for actively deciding which instantiation of environmental variables to use next, which is possible since these environmental variables can be controlled at the time of optimization. We discuss this further in Section 14.3.3.

In principle, this approach can be applied out-of-the-box for optimizing algorithm parameters to minimize mean performance across instances, such as, for example, our penalized

average runtime (PAR) criterion (see Section 3.4). However, note that the marginalization step is incompatible with transformations of the data. In particular, when using a log transformation on the original data, we face the issue of fitting the geometric mean we previously discussed in Sections 9.4 and 11.2.1. Recall that in the case of single problem instances, this problem did not manifest itself in our empirical evaluation. We attributed this to the fact that for many noise distributions the minima of geometric and arithmetic means coincide. However, in the presence of multiple problem instances of potentially widely-varying hardness this cannot be expected to be the case. For example, it might be possible to speed up an algorithm by two orders of magnitude for "easy" problem instances, leading to one order of magnitude slowdown for "hard" instances. Assuming roughly equal proportions of such easy and hard instances, such a modification would be judged beneficial under a geometric mean optimization objective but not when trying to optimize arithmetic mean. One may wonder whether this problem could be avoided by applying an inverse transformation before the marginalization across instances. This is not possible for a log transformation, because the exponential transformation required in this context is not linear. This means that the resulting predictions would no longer be Student-t-distributed, and thus we could no longer use a closed-form formula for predictions. For these reasons, it is an open question whether the approach of Williams et al. (2000) can be usefully applied to optimize mean algorithm runtime across multiple instances. For categorical parameters, ACTIVECONFIGURATOR based on RF models also showed more promise than ACTIVECONFIGURATOR based on GP models (see Section 12.3.4). Thus, here we concentrate on these RF models, leaving the application of Williams et al.'s approach to algorithm configuration to future work.[1]

13.2.2 Random Forests: Prediction of Cost Measures Across Multiple Instances

We extended our random forest (RF) model framework to predict cost measures defined across multiple instances. The model construction mechanism described in Section 11.1 (Procedure 11.1 on page 170) also works in the presence of instance features. The training data is now $\{([\boldsymbol{\theta}_i^\intercal\ \boldsymbol{x}_i^\intercal]^\intercal, o_i)\}_{i=1}^n$ instead of just $\{(\boldsymbol{\theta}_i, o_i)\}_{i=1}^n$, with d-dimensional parameter configurations $\boldsymbol{\theta}_i$ and e-dimensional instance feature vectors $\boldsymbol{x}_i$. At each split point during tree construction, we now have a choice between $d + e$ split variables (d algorithm parameters and e instance features). Note that when applying PCA as described in Section 13.1.4, we do so once, offline, resulting in k instance features (the original features projected onto the principal components), and we set $e = k$.

Prediction in RF models as defined in Procedure 11.2 on page 172 can then be used to predict the performance of a given ⟨configuration, instance⟩ combination. One could use this approach to predict cost measures across multiple instances, $\pi_1, \ldots, \pi_n$: first predict performance $\hat{\mu}_1, \ldots, \hat{\mu}_n$ for a single parameter configuration $\boldsymbol{\theta}$ and instances $\pi_1, \ldots, \pi_n$, and then compute sample statistic $\hat{\tau}(\hat{\mu}_1, \ldots, \hat{\mu}_n)$ to arrive at a prediction $\hat{\mu}(\boldsymbol{\theta})$ of $\boldsymbol{\theta}$'s cost measure, $c(\boldsymbol{\theta})$.

[1] We contacted Brian Williams, who generously agreed to provide the implementation of the approach used in their paper. However, since this implementation has not been made available to the public and was written nine years ago, tracking it down takes some time.

However, note that such an approach does not yield straightforward uncertainty estimates. In particular, if we were to predict means and variances separately for each of the instances, then it is unclear how to combine them. Rather, we should predict a joint distribution across these instances and then compute the measure we are optimizing from this joint distribution. When the cost measure we are optimizing is an expected value, this amounts to marginalizing out (computing the mean across) instances as done by Williams et al. (2000) (see previous section).

Procedure 13.1 implements such an approach in the RF model framework, mapping parameter configurations $\boldsymbol{\theta}_i$ to predicted cost statistics $\hat{\mu}(\boldsymbol{\theta})$ and uncertainty estimates $\hat{\sigma}(\boldsymbol{\theta})$. For each tree, T_b, and each instance, π_i, first, it identifies the tree's regions that $[\boldsymbol{\theta}_i^\intercal \ \boldsymbol{x}_i^\intercal]^\intercal$ falls in. Then, it computes the sample statistic of the sets of response values associated with the regions for all instances, thereby computing a tree prediction $\hat{\mu}_b$ of the cost measure, $c(\boldsymbol{\theta}_i)$. The uncertainty in this prediction is then computed as the empirical variance across tree predictions. Note that in Procedure 13.1, the sample statistic $\hat{\tau}$ is now computed on a set of weighted response values in order to avoid overcounting data from leaves with many response values. In our experiments for multiple instances, we only used the penalized average runtime as a cost statistic. In that case, the sample statistic on weighted data, $\hat{\tau}(\{(o_i, w_i)\}_{i=1}^n)$, simplifies to $1/(\sum_{i=1}^n w_i) \cdot \sum_{i=1}^n w_i \cdot o_i$. Performance metrics based on other sample statistics, such as, quantiles, can also be generalized to work on weighted data.

We also experimented with, but ultimately decided to discard, modifications of the standard mechanism for constructing RF models. These modifications were motivated by scenarios in which hardness varies strongly across instances but varies little across parameter configurations. In such scenarios, the standard mechanism mostly (or, in the extreme case, only) selects splits on instance features since they are most predictive of response values. Although the resulting model can capture performance differences across instances well, if it does not split on parameter values it predicts the same cost for each parameter configuration (and is thus useless for selecting promising configurations). To avoid this effect, we experimented with a mechanism for probabilistically forcing splits on parameter values, with the probability for forced splits on parameters increasing for deeper levels of the tree. This mechanism improved performance for some scenarios but worsened it for others. In particular, when forcing increasingly many splits on parameter values, model quality became more and more similar to that of models not using any features (which sometimes perform very well). However, since this mechanism did not consistently improve performance, we dropped it to increase our methods's conceptual simplicity.

13.3 Experimental Setup

In this chapter, to study model quality and performance of our SMBO configuration procedures we employ the five BROAD configuration scenarios (defined in Section 3.5.4) we already used in Chapters 5 and 7; their use allows a direct comparison to the results reported there. For model evaluation, we also use three of our COMPLEX configuration scenarios (described in Section 3.5.7).

For the SAPS-QCP and SPEAR-QCP scenarios, when computing problem instance features

Procedure 13.1: Prediction with Random Forest for Multiple Instances(RF, $\boldsymbol{\theta}$, t, Π)

Input : Random Forest RF=$\{T_1, \ldots, T_B\}$; parameter configuration, $\boldsymbol{\theta}$; transformation t; training set of instances, Π
Output : Predicted mean μ and variance σ^2 of cost measure $c(\theta)$

for $b = 1, \ldots, B$ **do**
  $\mathcal{Q} \leftarrow \emptyset$
  for all $\pi_i \in \Pi$ **do**
    Let R_m be the region of T_b containing $[\boldsymbol{\theta}_i^\intercal \; \boldsymbol{x}_i^\intercal]^\intercal$, with stored set of response values $\mathcal{O}_m$
    $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{\langle o_i, 1/|\mathcal{O}_m| \rangle \mid t(o_i) \in \mathcal{O}_m\}$
  $y_b \leftarrow t(\hat{\tau}(\mathcal{Q}))$, where $\hat{\tau}$ now works on a set of *weighted* responses $\langle o_i, w_i \rangle$
$\mu \leftarrow \frac{1}{B} \cdot \sum_{b=1}^{B} y_b$
$\sigma^2 \leftarrow \frac{1}{B-1} \cdot \sum_{b=1}^{B} (y_b - \mu)^2$
return $[\mu, \sigma^2]$

we noticed that some instances could already be solved by applying the polynomial-time preprocessor SATelite (Eén and Biere, 2005). Since the feature computation code we used started by applying this preprocessor, it did not yield meaningful features for such trivial instances. We thus removed these instances from the respective training sets (we removed 45/1000 instances for SAPS-QCP, and 24/1000 instances for SPEAR-QCP). We did not modify the test sets, ensuring that reported test performances remained directly comparable to those obtained with PARAMILS.

13.4 Model Quality

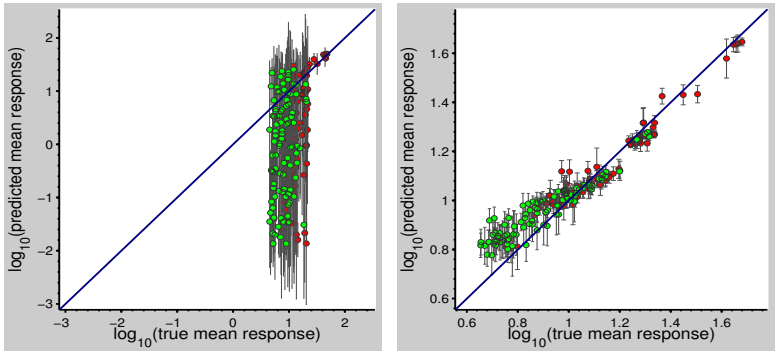
To study the quality of our RF models based on various sets of instance features, for each of our five BROAD configuration scenarios, we employed 10 001 data points. These were single runtimes of the respective target algorithm (CPLEX, SPEAR, or SAPS) for its default and for 10 000 randomly-sampled configurations. For each of these configurations, we sampled a problem instance uniformly at random from the set of training instances. (Here, we used a larger training set than in previous experiments with single-instance modelling problems since, intuitively, the learning task is harder due to the additional inter-instance variation.)

As in previous chapters, we used two sets of test configurations to evaluate model predictions: *Random*, a set of 100 randomly-sampled configuration, and *Good*, a set of 100 "good" configurations determined with a subsidiary configuration procedure (here FOCUSEDILS with a time budget of five hours) as described in Section 10.2.1. As throughout, we used our standard parameters for the RF model ($B = 10$, $n_{min} = 10$, *perc*=$5/6$).

13.4.1 Evaluation of Different Sets of Instance Features

We now study the quality of our RF models based on various sets of instance features:

1. the empty set (no features)
2. the 126 SAT or 39 MIP features described in Sections 13.1.1 and 13.1.2 (using the full



(a) No instance features (b) 126 SAT features

(c) Single feature: runtime of default (d) Single feature: instance index

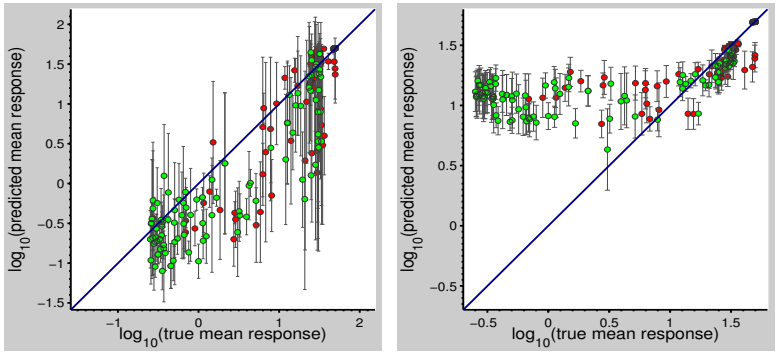
Figure 13.3: RF model quality based on different features, for scenario SAPS-QCP. We plot predicted mean $\pm$ stddev *vs* actual cost for test sets *Good* (shown in green) and *Random* (shown in red).

set, without applying PCA)

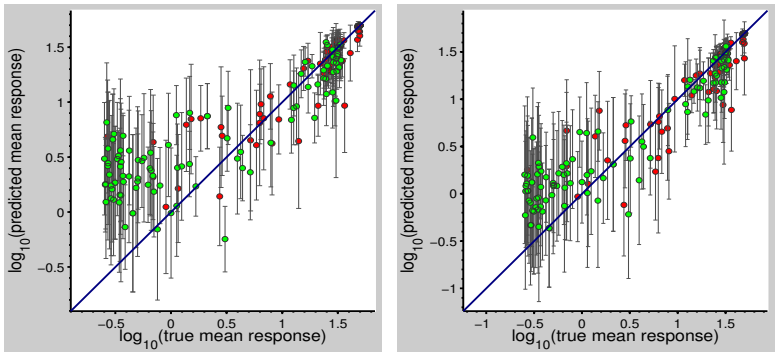
3. the generic feature "runtime of the default configuration" described in Section 13.1.3, and
4. the generic instance index feature described in Section 13.1.3.

In Figures 13.3 and 13.4, we qualitatively compare model quality based on these four sets of features, for the two configuration scenarios in which instance features were most and least useful. For configuration scenario SAPS-QCP (see Figure 13.3), instance features dramatically improved model quality. The 126 SAT features yielded the best performance, but the two single generic features already substantially improved model quality compared to not using any features at all.

In contrast, for configuration scenario SAPS-SWGCP (see Figure 13.4), models that completely ignored the existence of different instances performed very well, and in particular much *better* than models based on the 126 SAT features. Both of the generic single features



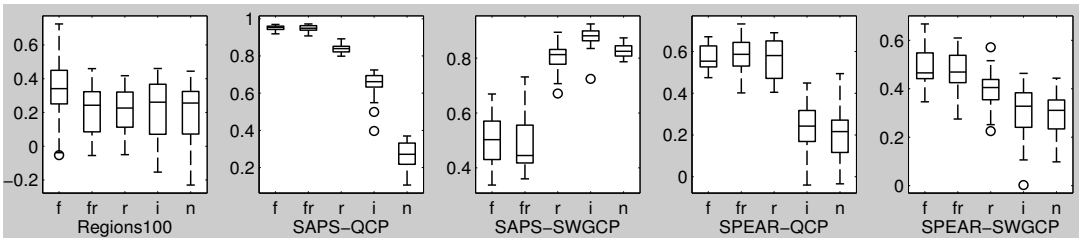
(a) No instance features (b) 126 SAT features



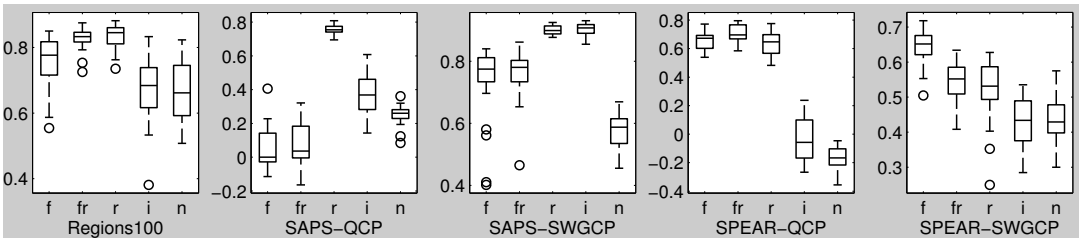
(c) Single feature: runtime of default (d) Single feature: instance index

Figure 13.4: RF model quality based on different features, for scenario SAPS-SWGCP We plot predicted mean $\pm$ stddev *vs* actual cost for test sets *Good* (shown in green) and *Random* (shown in red).

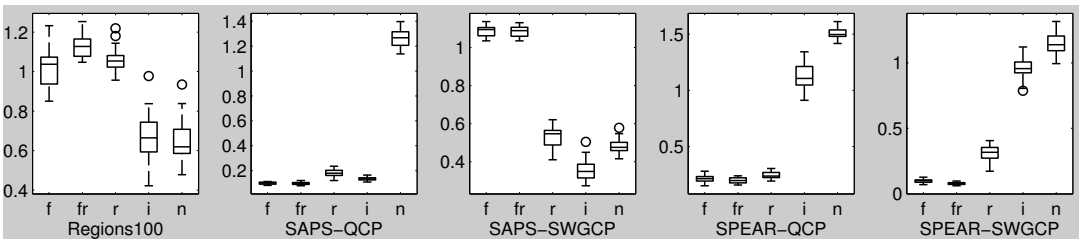
led to performance comparable to that without features. This result is counter-intuitive since we typically would expect that RF models ought to filter out and only use features that help to improve performance. We thus emphasize that the results reported here are preliminary and can likely be improved considerably. There are several possible explanations for this example of poor model performance when using SAT features. First, it has previously been observed that instance features, such as the ones we used, are not very predictive for the type of SWGCP instances used in this case (Xu et al., 2007a). Second, we employ very short cutoff times of 5 seconds per run, in combination with our penalized average runtime criterion (counting timed-out runs as 50 seconds). As we show in Figure 13.7(b) on page 225, about a third of the runs timed out, creating a very unbalanced training set. Our model predictions for combinations of parameter configurations and instances (shown in Figure 13.7(d) on page 225) captured this partition into poor and good parameter configurations, but missed the more subtle differences that distinguished the cost measures of good and very good configurations. In future work, we plan to study the causes for this poor performance in more detail.



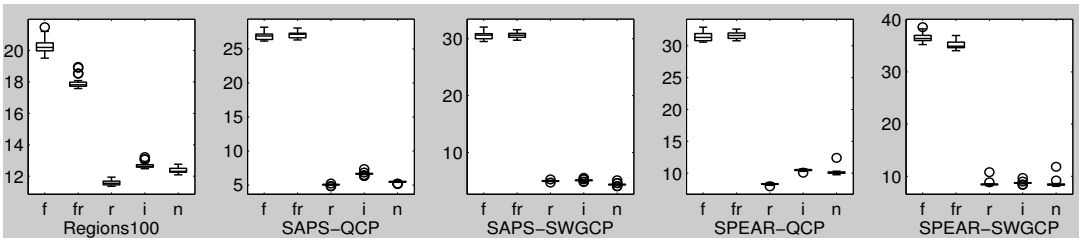
(a) Quality of predictive ranks (high is good, 1 is optimal)



(b) EIC quality (high is good, 1 is optimal)



(c) Root mean squared error (RMSE; low is good, 0 is optimal)



(d) CPU time (in seconds); for the SAPS scenarios, the time for 'r', 'i', and 'n'; is around five seconds per run

Figure 13.5: RF model quality based on different sets of instance features. We performed 25 runs for each model with different training but identical test data and show boxplots for the respective quantities across the 25 runs. In each plot, 'f' denotes the 126 SAT / 39 MIP features 'r' the generic default runtime feature, 'fr' the union of 'f' and 'r', 'i' the generic index feature, and 'n' no features at all.

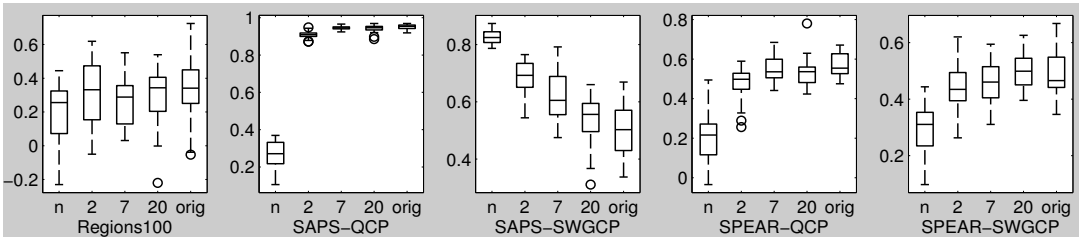
In Figure 13.5, we quantify the performance of our four feature sets (and one additional one) across multiple runs on all five BROAD configuration scenarios. The additional set of features we considered here is the union of the 126 SAT / 39 MIP features and the runtime of the default. As we show in Figure 13.5, adding the default runtime feature did not change performance much as compared to the 126 SAT / 39 MIP features alone (the most visible effects were that in the case of CPLEX-REGIONS100, it improved performance with respect to two measures of model quality, and for SPEAR-SWGCP, it worsened EIC quality). Otherwise, it did not change performance much. Differences in the quality of predictive ranks and in RMSE are most striking for the two scenarios we detailed in Figures 13.3 and 13.4, SAPS-QCP and SAPS-SWGCP. For the former, features improved model quality substantially, whereas for the latter using the 126 SAT features substantially worsened performance (again, we emphasize that this latter result is counterintuitive and preliminary, and that we plan to study its causes in future work). Surprisingly, the single generic default runtime feature yielded very high-quality models. In particular, only models based on this feature alone performed well on both of SAPS-QCP and SAPS-SWGCP. It also yielded rather good performance for the remaining scenarios, making it the overall most robust choice. Finally, also note the differences in the time required to learn models: the versions using many features were consistently slower than the versions using just one feature or none at all. For CPLEX-REGIONS100 we optimize $d = 63$ parameters, such that using $e = 39$ MIP features only caused a slowdown of a factor about 1.5. However, using the $e = 126$ SAT features caused slowdown factors around 3 for the SPEAR scenarios ($d = 26$) and 5 for the SAPS scenarios ($d = 4$).

13.4.2 Evaluation of PCA to Speed Up Learning

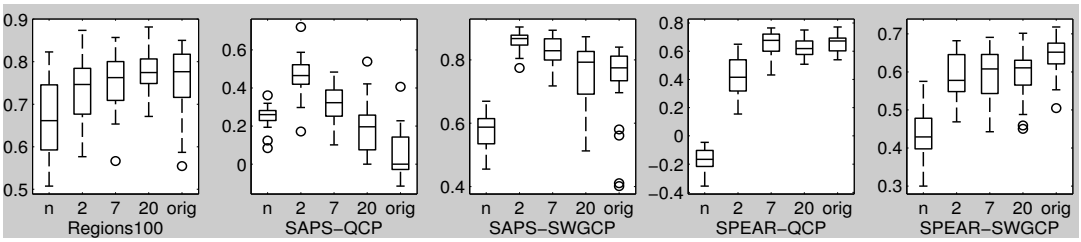
Next, we studied the performance of applying PCA to reduce this time complexity. As we show in Figure 13.6(d), using only a few principal components of the 126 SAT/39 MIP features speeds up the model learning substantially (to the point where the time required is basically the same as without features). Figure 13.6 also shows that as we use more principal components, typically there is a gradual change of model quality, from being comparable to the featureless model to being comparable to the model with all 126 SAT / 39 MIP features. Clearly, the best number of principal components depends on the configuration scenario. For the experiments in the remainder of this chapter we chose to use 7 principal components, which yielded good average performance.

13.5 Prediction of Matrix of Runtimes

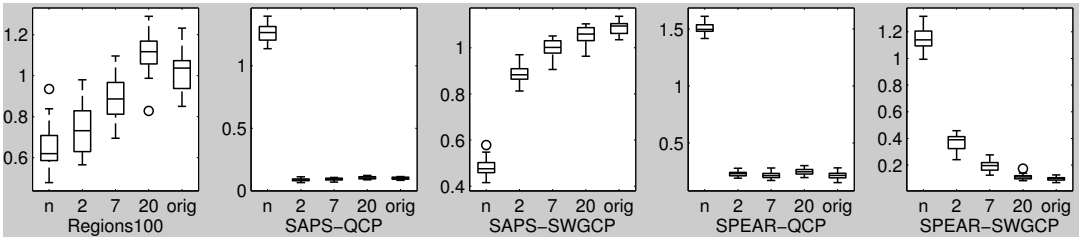
Finally, we evaluated how well our RF models can predict runtimes for combinations of parameter configurations and instance features. While our current approach for algorithm configuration only requires predictions of cost measures defined across instance features, such predictions can be very useful in future work on actively deciding which problem instance to use for a run (see Section 14.3.3), as well as for instance-based selection of algorithm parameters (see Section 14.3.5). They can also be used to transform our expensive offline empirical analysis approaches from Chapter 4 into computationally-cheap online analysis approaches using predicted instead of true runtimes (see Section 14.3.1).



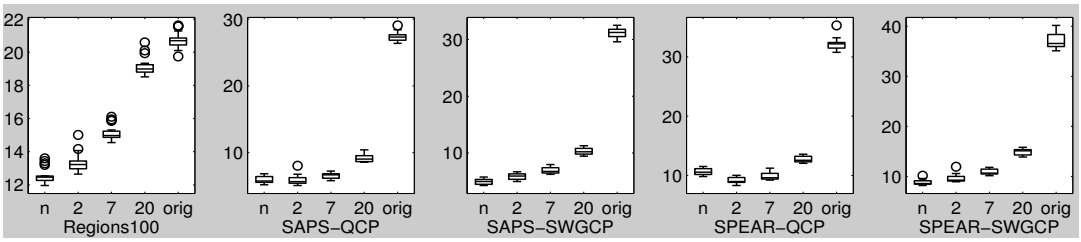
(a) Quality of predictive ranks (high is good, 1 is optimal)



(b) EIC quality (high is good, 1 is optimal)



(c) Root mean squared error (RMSE; low is good, 0 is optimal)



(d) CPU time (in seconds)

Figure 13.6: RF model quality based on different number of principal components of 126 SAT/39 MIP features. We performed 25 runs of models with no features ('n'), 2, 7, and 20 principal components, and with the original set of features without PCA ('orig'). We show boxplots across those 25 runs for each of our performance measures.

In this section, we used the full 126 SAT and 39 MIP features introduced in Sections 13.1.1 and 13.1.2, without applying PCA. The reason we did not use PCA here is that some benchmark sets (in particular, benchmark sets BMC and SWV) contain a large fraction of instances for which most of our SAT features are not defined. More specifically, over 50% of the instances in set SWV could be solved by applying the polynomial-time preprocessor SATelite (Eén and Biere, 2005) (note that we already pointed out this large fraction of trivial SWV instances in our empirical analysis in Section 4.2). For such instances, the feature computation code we use only yielded meaningful values for the original number of variables and clauses (before preprocessing); all other features have default values (0 for some features, -512 for others). One option would be to simply discard these trivial instances (as we indeed did for the small fraction of QCP instances in the remainder of this chapter). However, since we wanted to predict runtimes for all instances $\pi \in \Pi$ and trusted in the ability of random forest models to filter out meaningless features, we simply used the features as outputted by the feature computation code (including the values 0 and -512). We did not apply PCA to this data since we feared this might make it harder to detect meaningless features.

For the evaluation of model quality in this section, we used the same type of data underlying our empirical analysis in Chapter 4: a $M \times P$ matrix containing the runtime of $M = 1\,000$ randomly-sampled parameter configurations on a set of $P = 2\,000$ problem instances. We trained RF models on runtimes of $10\,000$ randomly-sampled combinations of the first $M/2$ configurations and $P/2$ instances, and tested predictions on the second half of configurations and instances, thus testing generalization performance to both previously-unseen instances and configurations. Since the goal of this experiment was not to predict cost measures across instances, but to predict runtimes on single instances, we used our original RF model prediction, outlined in Procedure 11.2 on page 172 (instead of Procedure 13.1, which predicts cost statistics $\hat{\mu}(\boldsymbol{\theta})$ *across instances*).

In Table 13.1, we list quantitative results for the BROAD configuration scenarios and three COMPLEX configuration scenarios with larger cutoff times per run. For each scenario, we provide results for three experiments:

1. **Prediction for unseen configurations on training instances**, in column "Π_{train}, $\boldsymbol{\Theta}_{test}$". This is important in algorithm configuration to evaluate how promising new configurations are; all model quality plots in the rest of the thesis are of this form.
2. **Prediction for training configurations on unseen test instances**, in column "Π_{test}, $\boldsymbol{\Theta}_{train}$". This is important in the context of selecting on a per-instance basis which of a set of given parameter configurations will perform best on a previously-unseen test instance.
3. **Prediction for unseen configurations on unseen instances**, in column "Π_{test}, $\boldsymbol{\Theta}_{test}$". This most general case is important for per-instance algorithm configuration, where we perform a search through a (potentially large configuration space) to identify the configuration that is most promising for a previously-unseen test instance. (Note that such a search can be carried out very quickly; we routinely perform exactly such a search in every of the thousands of iterations of our SMBO procedures, taking on the order of seconds.)

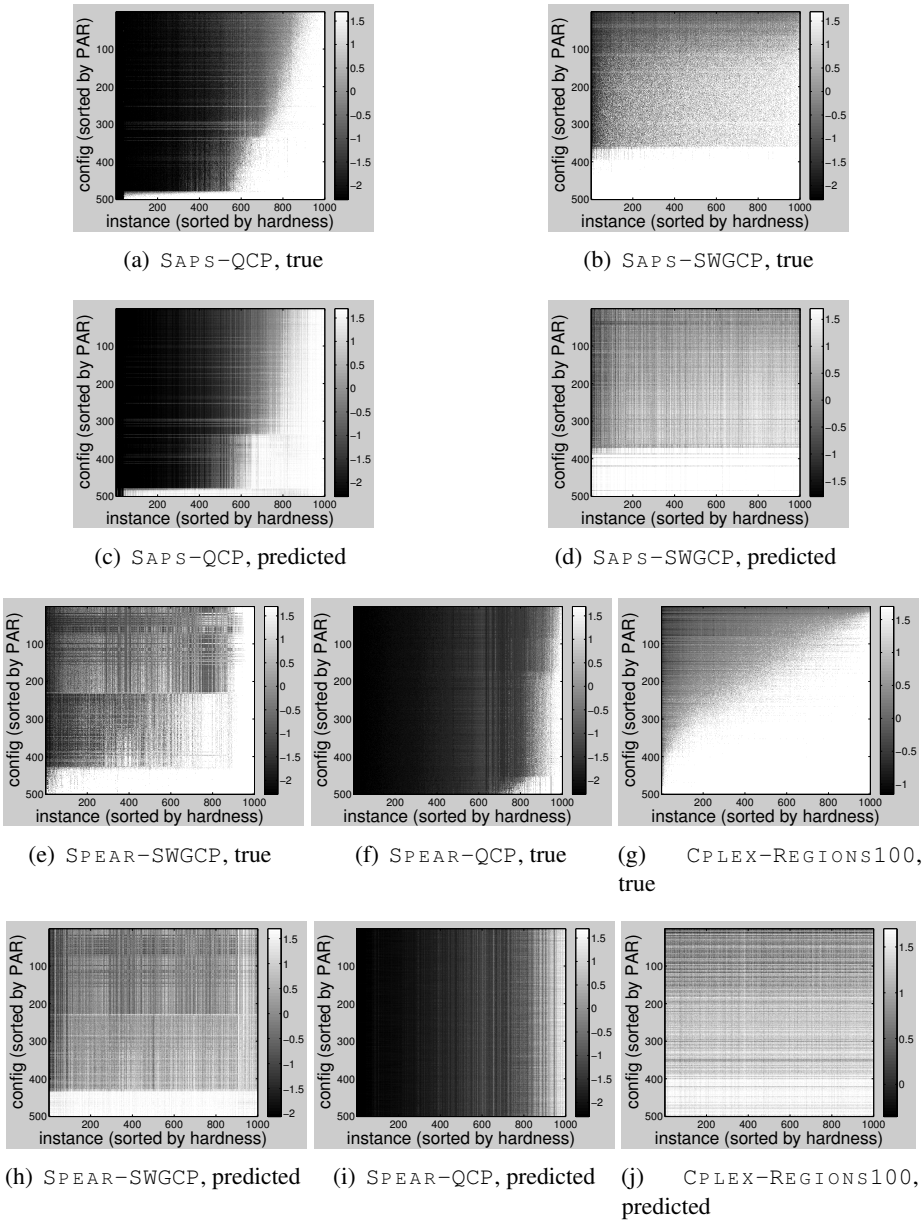
Scenario	$\lvert\Pi_{train}\rvert$	κ_{max}	time [CPU s]	Π_{train}, Θ_{test}	Π_{test}, Θ_{train}	Π_{test}, Θ_{test}
SAPS-QCP	1 000	5	26.7	0.41 / 0.95	0.43 / 0.95	0.44 / 0.95
SAPS-SWGCP	1 000	5	31.2	0.98 / 0.60	0.98 / 0.60	0.98 / 0.59
SPEAR-QCP	1 000	5	31.7	0.51 / 0.93	0.57 / 0.91	0.58 / 0.91
SPEAR-SWGCP	1 000	5	38.8	0.91 / 0.72	1.02 / 0.61	1.05 / 0.58
CPLEX-REGIONS100	1 000	5	21.0	0.60 / 0.52	0.54 / 0.61	0.61 / 0.50
SPEAR-SWV	50	300	20.3	0.54 / 0.97	0.56 / 0.95	0.57 / 0.95
SPEAR-IBM	50	300	15.6	0.38 / 0.96	0.94 / 0.82	0.94 / 0.82
CPLEX-ORLIB	70	300	20.3	0.61 / 0.94	0.82 / 0.88	0.89 / 0.86

Table 13.1: Model quality for predictions of ⟨configuration, instance⟩ combinations. We trained on 10 000 randomly-sampled combinations of parameter configurations $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{train}$ and $\pi \in \Pi_{train}$. We report test performance for three different combinations of instances and parameter configurations, corresponding to the three experiments listed in the text. We computed root mean squared error (RMSE) and Spearman correlation coefficient, ρ, between the predicted and actual runtimes of the respective test set, and for each combination report results in the form RMSE/ρ. For each configuration scenarios, we also list the captime, κ_{max}, as well as the time required to build the RF model.

The two sets of configuration scenarios listed in Table 13.1 differ in important aspects. Firstly, the BROAD configuration scenarios contained larger training instance sets, Π_{train}, than the COMPLEX scenarios. In particular, for the BROAD scenarios they contained 1 000 instances each, whereas in the case of the COMPLEX scenarios they only contained 50 instances for SPEAR-SWV and SPEAR-IBM, and 70 for CPLEX-ORLIB, half of the P instances we had runtime data available for (the other half was used as test set, Π_{test}). Secondly, these three COMPLEX scenarios employed a much larger captime ($\kappa_{max} = 300$s) than the BROAD scenarios ($\kappa_{max} = 5$s).

Consequently, generalization performance differed between the two sets of configuration scenarios. Recall that in either case, we trained on 10 000 randomly-sampled ⟨parameter configuration, instance⟩ combinations. For the BROAD scenarios, where these 10 000 samples were spread across 1 000 training instances, this resulted in good generalization to previously-unseen test instances, that is, the difference between experiments 1 and 3 from above was rather small (compare columns 5 and 7 of Table 13.1). In contrast, for the COMPLEX configuration scenarios, where the 10 000 training data points were only spread over much fewer instances (50, 50, and 70), generalization to previously-unseen instances was sometimes much worse than "just" generalization to unseen configurations. (compare columns 5 and 7 of Table 13.1). However, model predictions on previously-seen instances (the task important in the context of algorithm configuration), were very good for the COMPLEX scenarios, with rank correlation coefficients between predicted and actual runtime in the high 90% range. We attribute this strong performance to the larger cutoff times used: the data is richer than for the BROAD scenarios, in which every run is cut off after five seconds. The difference between experiments 2 and 3 from above was rather small for all configuration scenarios (compare columns 6 and 7 of Table 13.1), that is, generalization to unseen parameter configurations generally worked well.

In Figures 13.7 and 13.8, we provide plots for each of the configuration scenarios listed



(a) SAPS-QCP, true

(b) SAPS-SWGCP, true

(c) SAPS-QCP, predicted

(d) SAPS-SWGCP, predicted

(e) SPEAR-SWGCP, true

(f) SPEAR-QCP, true

(g) CPLEX-REGIONS100, true

(h) SPEAR-SWGCP, predicted

(i) SPEAR-QCP, predicted

(j) CPLEX-REGIONS100, predicted

Figure 13.7: Predictions of runtime matrix, for BROAD configuration scenarios, compared to the true matrix of runtimes. We show data for configurations and instances withheld during training.

in Table 13.1, in which we compare the true matrix of runtimes to the predicted matrix. For this visualization, we sorted the rows of both true and predicted matrix by the true penalized average runtime of the respective configuration; we also sorted the columns of either matrix by the respective instance's true average hardness (PAR across runs of all configurations on the instance). For the five BROAD scenarios, in Figure 13.7, we only show the matrix for the $M/2$ unseen test configurations and the $P/2$ unseen test instances. For the three COMPLEX scenarios, in Figure 13.8, we show plots for that setting as well as predictions on unseen test configurations for the training instances.

We discuss the predictive quality for two of the BROAD scenarios in detail. For configuration scenario SAPS-QCP, model predictions were excellent: the true and predicted matrices in Figures 13.7(a) and 13.7(c) are very similar. The model perfectly predicted the fact that about 5% of the previously-unseen parameter configurations performed very poorly, except for the about 5% easiest instances (also previously-unseen). The model also predicted interactions of parameter configurations and instance features very well; this is visible by the correct prediction of some horizontal stripes (for example around parameter configuration 300, in the sorted list). The RF model we evaluate here is the model which corresponds to the one used for the predictions in Figure 13.3(b); not surprisingly, the very good predictions for individual instances here transferred to very good predictions of the cost measure across instances. For scenario SAPS-SWGCP (see Figures 13.7(b) and 13.7(d)), the model correctly predicted that about a third of the configurations yielded very poor performance, but missed more subtle differences for the good parameter configurations and in instance hardness. The model we evaluate here corresponds to the model in Figure 13.3(b), which also performed poorly for predictions across instances. We hypothesize that this poor performance is at least in part due to the large fraction of very poor parameter configurations: if instance features are used in the predictive model, the poor performance in each training run using such a configuration $\boldsymbol{\theta}$ and an instance π will in part be attributed to instance π and thereby cause predictions to deteriorate. We plan to study the causes for the poor performance in this case in future work in more detail.

We summarize results for the remaining three BROAD scenarios. For scenario SPEAR-QCP (see Figures 13.7(f) and 13.7(i)), the model accurately identified a small portion of very hard instances, but largely missed the fact that some parameter configurations performed much better than others. For scenario SPEAR-SWGCP, the model captured the fact that relative rankings of configurations were rather unstable across problem instances. In particular, it correctly predicted that configurations 1–230 (sorted by PAR) behaved differently than configurations 231–430 (and that configurations past 430 behaved poorly throughout). It also correctly predicted many horizontal and vertical stripes (indicators of the instability of relative rankings), but missed the fact that instances 700–880 were very hard for configurations 231–430. For scenario CPLEX-REGIONS100 (see Figures 13.7(g) and 13.7(j)), the model captured the performance differences across parameter configurations, but missed the differences in instance hardness. We hypothesize that this might be due to the fact that the instances in the `Regions100` benchmark set are very homogeneous (they have very similar numbers of variables and constraints, see Section 3.3.2), and detecting any differences between them might be hard; splits on instance features might thus be wasteful of the data. In future work,

we hope to improve model quality for this case.

Finally, we discuss predictive quality for the three COMPLEX scenarios, shown in Figure 13.8. For configuration scenario SPEAR-SWV, note that, although over 50% of the instances could be solved by a polynomial-time preprocessor (SATelite, see our discussion at the beginning of this section), there were still rather substantial performance differences for solving them, which, in fact, were predicted correctly by the model. Predictive performance was already relatively good for the hardest case (generalization to previously-unseen instances and previously-unseen configuration), see the top two rows of Figure 13.8 and column 7 ("Π_{test}, $\boldsymbol{\Theta}_{test}$") of Table 13.1. This performance substantially improved further for predictions on previous-seen instances, the case that is important in the context of algorithm configuration (see the bottom two rows of Figure 13.8 and column 5 ("Π_{train}, $\boldsymbol{\Theta}_{test}$") of Table 13.1). For that case, the true and predicted runtime matrices were almost identical: compare Figures 13.8(g) and 13.8(j) (for SPEAR-SWV); 13.8(h) and 13.8(k) (for SPEAR-IBM); and Figures 13.8(i) and 13.8(l) (for CPLEX-ORLIB). In particular, in the case of CPLEX-ORLIB, note that the complex interactions between parameter configurations and instances were predicted extremely well, especially the prominent vertical stripes (indicating that an instance was very easy for some configurations but hard for others, whereas other instances were similarly-hard for all configurations).

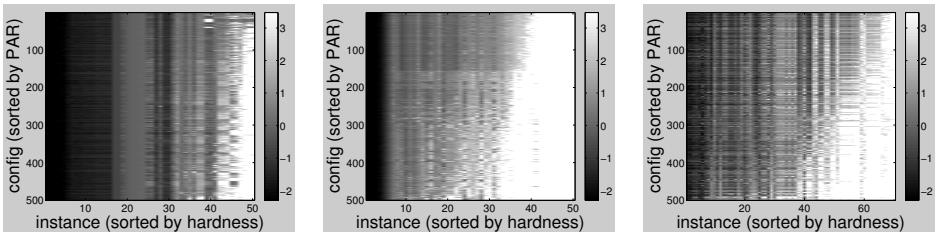
Overall, these results are very promising. They suggest the possibility of exploiting model predictions of instance hardness in algorithm configuration procedures, and of predicting on a per-instance basis which parameter configurations are likely to perform well, lines of research we plan to follow in the future (see Sections 14.3.3 and 14.3.5). However, we also note that predictions were rather poor in some cases.

13.6 Sequential Optimization Process

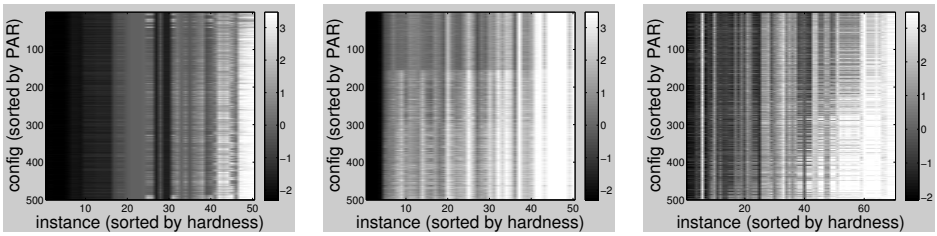
We now study SMBO configuration procedures for configuration scenarios with multiple instances. We first introduce a new intensification mechanism that implements a blocked comparison, then evaluate SMBO procedures based on the instance features discussed in this chapter, and then compare SMBO to FOCUSEDILS and RANDOM*.

13.6.1 Blocking on Instances and Seeds

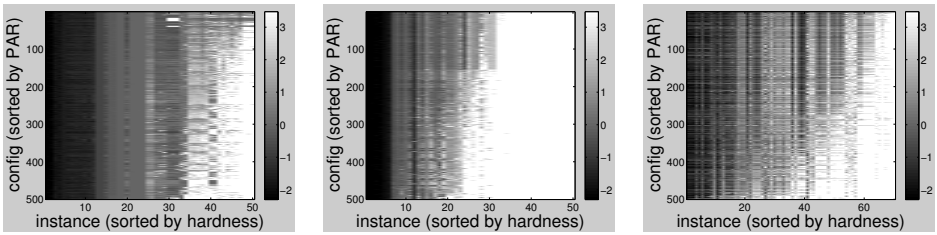
When we compare empirical cost statistics, $\hat{c}_N(\boldsymbol{\theta}_1)$ and $\hat{c}_N(\boldsymbol{\theta}_2)$, of two parameter configurations, $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$, defined across multiple instances, intuitively, the variance in this comparison is lower if we use the same N instances (and seeds) to compute both empirical estimates, $\hat{c}_N(\boldsymbol{\theta}_1)$ and $\hat{c}_N(\boldsymbol{\theta}_2)$. We discussed the advantages of such a blocking scheme in Section 3.6.1 and used it in our empirical analysis of algorithm configuration scenario (see Section 4.2) as well as in FOCUSEDILS (see Section 5.3). We now define a new intensification mechanism in the SMBO framework that implements such a blocking mechanism. We then demonstrate that for configuration scenarios with multiple instances this new mechanism performs better than our previously-best intensification mechanism (the one we used in SPO*, ACTIVECONFIGURATOR, and RANDOM* throughout the previous chapters), defined in Procedure 10.5 on page 165.



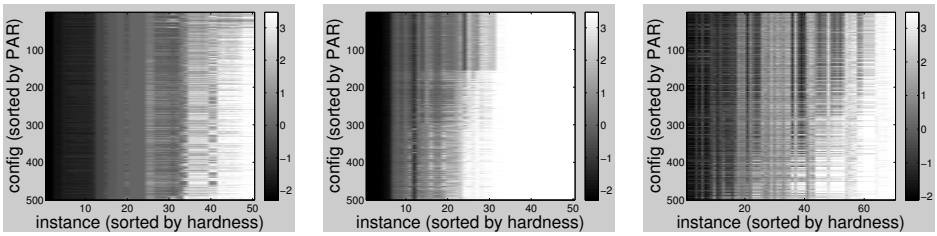
(a) SPEAR-SWV, true, Π_{test} (b) SPEAR-IBM, true, Π_{test} (c) CPLEX-ORLIB, true, Π_{test}



(d) SPEAR-SWV, predicted, Π_{test} (e) SPEAR-IBM, predicted, Π_{test} (f) CPLEX-ORLIB, predicted, Π_{test}



(g) SPEAR-SWV, true, Π_{train} (h) SPEAR-IBM, true, Π_{train} (i) CPLEX-ORLIB, true, Π_{train}



(j) SPEAR-SWV, predicted, Π_{train} (k) SPEAR-IBM, predicted, Π_{train} (l) CPLEX-ORLIB, predicted, Π_{train}

Figure 13.8: Predictions of runtime matrix, for the three COMPLEX scenarios with $\kappa_{max} = 300$. We show true and predicted data for configurations and instances withheld during training (top two rows); and for configurations withheld during training but instances used during training (bottom two rows).

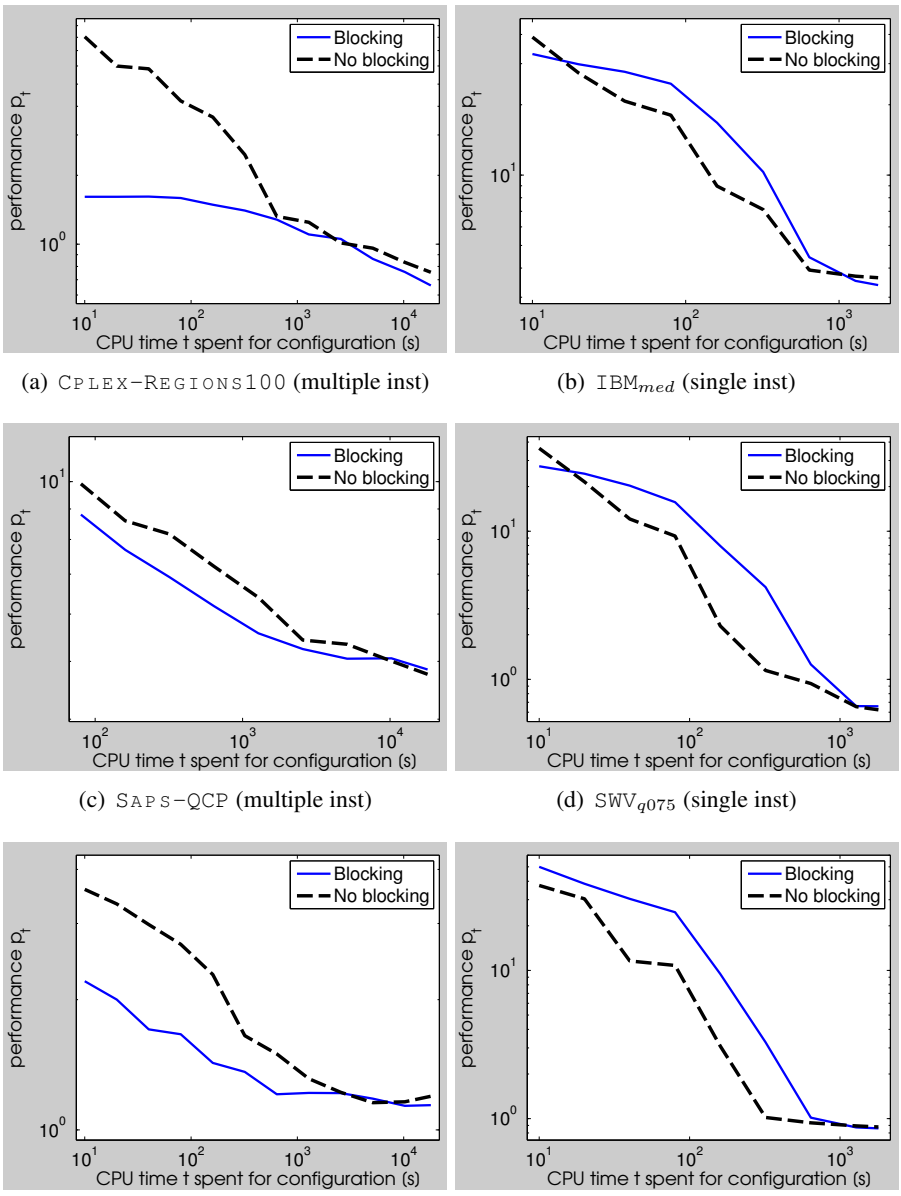
Procedure 13.2: Intensify($\boldsymbol{\theta}_{new}$, $\boldsymbol{\theta}_{inc}$, $\mathbf{R}$) with Blocking in ACTIVECONFIGURATOR for Multiple Instances

maxR is a parameter of configurators using this procedure, set to 2 000 in all our experiments

Input : Sequence of parameter configurations to evaluate, $\vec{\boldsymbol{\Theta}}_{new}$, here with one element; incumbent configuration, $\boldsymbol{\theta}_{inc}$; model, $\mathcal{M}$; sequence of target algorithm runs, $\mathbf{R}$
Output: Sequence of target algorithm runs, $\mathbf{R}$; incumbent configuration, $\boldsymbol{\theta}_{inc}$

$L_{inc} \leftarrow \{(\pi, s) | \exists i . \mathbf{R}[i] = (\theta_{inc}, \pi, s, \cdot, \cdot)\}$
if $|L_{inc}| < maxR$ **then**
    Select $\pi \in \Pi$ and seed $s \in S$ uniformly at random
    $\mathbf{R} \leftarrow$ ExecuteRun($\mathbf{R}$, $\boldsymbol{\theta}_{inc}$, π, s)
$N \leftarrow 1$
while *true* **do**
    $L_{new} \leftarrow \{(\pi, s) | \exists i . \mathbf{R}[i] = (\theta_{new}, \pi, s, \cdot, \cdot)\}$
    $L \leftarrow$ random subset of $L_{inc} \setminus L_{new}$ of size $\min(N, |L_{inc} \setminus L_{new}|)$
    for $(\pi, s) \in L$ *in randomized order* **do**
        $\mathbf{R} \leftarrow$ ExecuteRun($\mathbf{R}$, $\boldsymbol{\theta}_{new}$, π, s)
    $L_{new} \leftarrow \{(\pi, s) | \exists i . \mathbf{R}[i] = (\theta_{new}, \pi, s, \cdot, \cdot)\}$
    $\mathbf{R}_{inc} \leftarrow \{\mathbf{R}[i] | \mathbf{R}[i] = (\theta_{inc}, \pi, s, \cdot, \cdot)\}$
    $\mathbf{R}_{new} \leftarrow \{\mathbf{R}[i] | \mathbf{R}[i] = (\theta_{new}, \pi, s, \cdot, \cdot)$ and $(\theta_{inc}, \pi, s, \cdot, \cdot) \in \mathbf{R}_{inc}\}$
    if $\hat{c}(\boldsymbol{\theta}_{inc}, R_{inc}) < \hat{c}(\boldsymbol{\theta}_{new}, R_{new})$ **then return** [$\boldsymbol{\theta}_{inc}$, $\mathbf{R}$]
    else if $L_{inc} \subseteq L_{new}$ **then return** [$\boldsymbol{\theta}_{new}$, $\mathbf{R}$]
    else $N \leftarrow 2 \cdot N$

Procedure 13.2 defines this new intensification mechanism. Briefly, whenever a new configuration, $\boldsymbol{\theta}_{new}$, is compared to the incumbent configuration, $\boldsymbol{\theta}_{inc}$, we first perform an additional run for the incumbent, using a randomly-selected ⟨instance, seed⟩ combination, and then perform a sequence of runs using $\boldsymbol{\theta}_{new}$. For that sequence of runs, we select ⟨instance, seed⟩ combinations at random from those that have been used for the incumbent. (Note that since we now take control of instances and seeds used in each run, we need to redefine the procedure to execute runs; see Procedure 13.3.) We always compare $\boldsymbol{\theta}_{inc}$ and $\boldsymbol{\theta}_{new}$ based on the ⟨instance, seed⟩ combinations they both have been run for. This mechanism is similar to the one implemented in FOCUSEDILS (see Procedure $better_{Foc}$ on page 82). The main difference is that instead of using a fixed ordering of instances and seeds, every comparison in this new procedure is based on a different randomly-selected subset of instances and seeds. It thereby avoids issues arising from unfortunate fixed orderings of instances and seeds, which we *did* indeed encounter with FOCUSEDILS (see our discussion of an extreme example in Section 8.1.2). In Section 13.6.2, we demonstrate that—based on this improved intensification mechanism—even random search can substantially outperform FOCUSED-ILS on some configuration scenarios, clearly arguing for the strength of the intensification mechanism.



(a) CPLEX-REGIONS100 (multiple inst)

(b) IBM_{med} (single inst)

(c) SAPS-QCP (multiple inst)

(d) SWV_{q075} (single inst)

(e) SPEAR-QCP

(f) SWV_{q095} (single inst)

Figure 13.9: Evaluation of RANDOM* based on intensification Procedures 13.2 (blocking) and 10.5 (no blocking), on 3 BROAD configuration scenarios (multiple instances, left column) and 3 SINGLEINSTCAT scenarios (single instances, right column). We performed 25 runs of each procedure and show mean $p_{test,t}$ across the 25 runs for a varying time budget.

Procedure 13.3: ExecuteRuns($\mathbf{R}$, $\boldsymbol{\theta}$, π, s) with Multiple Instances

Input : Sequence of target algorithm runs, $\mathbf{R}$; parameter configuration, $\boldsymbol{\theta}$; problem instance, π; seed, s
Output: (Extended) sequence of target algorithm runs, $\mathbf{R}$

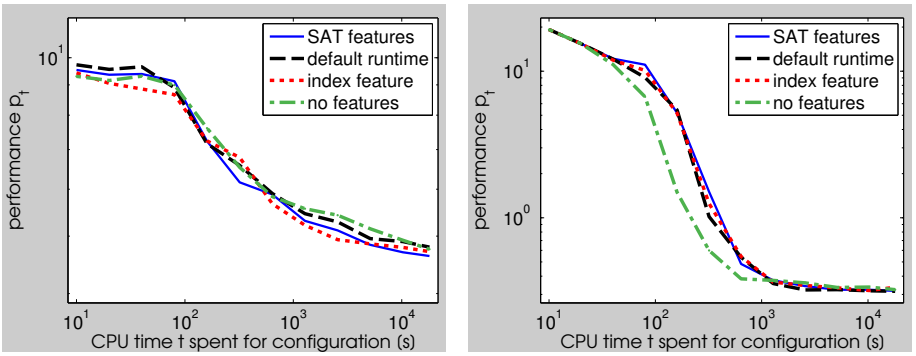
1 Execute $A(\boldsymbol{\theta})$ with seed s and captime κ_{max} on instance π, store response in o
2 Append $(\boldsymbol{\theta}, \pi, s, \kappa, o)$ to $\mathbf{R}$
3 **return** $\mathbf{R}$

13.6.2 Experimental Evaluation of SMBO Configuration Procedures

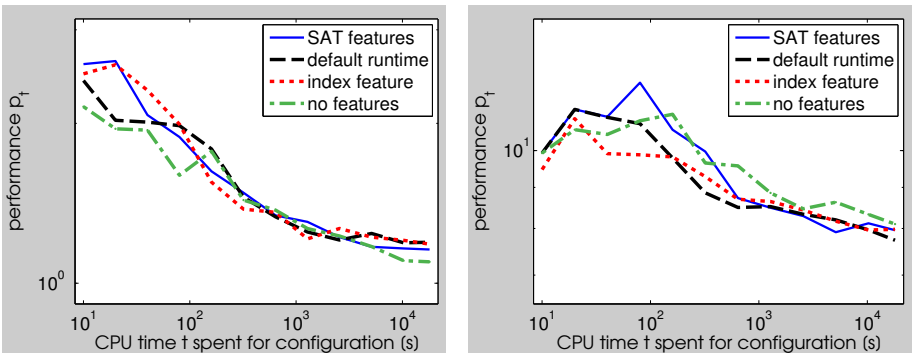
We first experimentally evaluated our new intensification procedure. In this evaluation, we used configuration procedure RANDOM*, based on our previous intensification mechanism, Procedure 10.5 on page 165 (with does not use any blocking) and our new Procedure 13.2. We chose RANDOM* in order to study the intensification procedure in isolation. (In ACTIVECONFIGURATOR, the different data we gather using different intensification procedures would also affect the model and could thereby have an additional, indirect, effect on performance.) Figure 13.9 shows the results with the two intensification procedures. For configuration scenarios with multiple instances (left column), blocking tended to help, whereas not blocking (and rather using all available runs to estimate each configuration's performance) tended to be better for single-instance scenarios. For the two remaining scenarios from either set, (not shown), neither of the intensification procedures was substantially better than the other.

Now, we compare four variants of ACTIVECONFIGURATOR (AC) that only differ in the type of instance features they use. All these AC variants are based on RF models and use our new intensification mechanism. We studied AC variants based on the four sets of features discussed in Section 13.4.1, plus PCA: (1) the empty set (no features) (2) the first seven principal components of the 126 SAT/39 MIP features described in Sections 13.1.1 and 13.1.2; (3) the generic feature "runtime of the default configuration" described in Section 13.1.3; and (4) the generic instance index described in Section 13.1.3. Apart from this difference in features, all variants were identical.

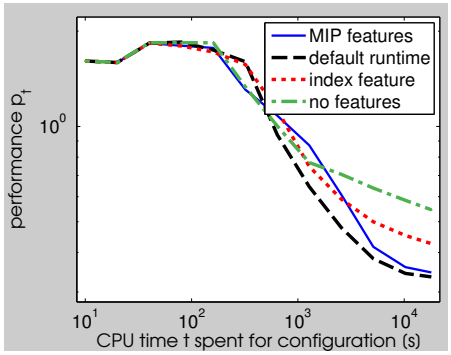
Figures 13.10 and 13.11 show the results of this comparison. Notably, for most scenarios, AC performed quite similarly based on the different types of models; this can be explained by the random configurations AC interleaves regardless of model quality and the common intensification procedure. For SAPS-SWGCP, AC using no features at all was fastest to find good configurations (see Figure 13.10(b) but also note that at the end of its trajectory the featureless variant of AC yielded marginally worse performance than other variants). The good initial performance of this variant is not surprising since, as discussed in Section 13.4.1, featureless models yielded the best performance for SAPS-SWGCP (see Figure 13.4 on page 219). What is more surprising is the good performance of the featureless variant for SPEAR-QCP; even though initial model quality without features was very poor for this scenario (see Figure 13.5 on page 220), this variant led to the best performance at the end of the trajectory (see Figure 13.11). We hypothesize that differences between the performance in sequential optimization we see here and the quality of the models discussed in Section 13.4 is due to the qualitatively different training data that models are based on. While in Section 13.4, parameter configurations and instances were sampled uniformly at random, here the sequential optimization gathered much more runs of well-performing parameter configurations and selecting instances (and seeds) by



(a) SAPS-QCP (b) SAPS-SWGCP



(c) SPEAR-QCP (d) SPEAR-SWGCP



(e) CPLEX-REGIONS100

Figure 13.10: Performance of AC(RF) variants based on different sets of features. We performed 25 runs of the configurators and show mean $p_{test,t}$ across the 25 runs for a varying time budget, t.

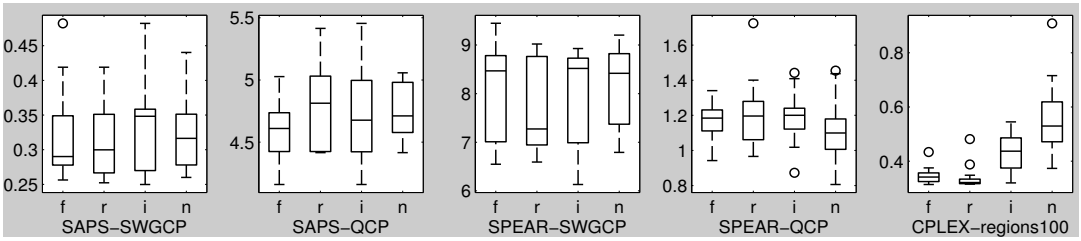


Figure 13.11: Final performance of AC(RF) variants based on different sets of features. We performed 25 runs of the configurators and show boxplots of their test performance $p_{test,t}$ for a time budget of $t = 18\,000$ seconds. Based on a Mann Whitney U test (see Section 3.6.2), all pairwise performance differences for scenario CPLEX-REGIONS100 were significant. The only other significant difference was between using no features and the index feature for SPEAR-QCP.

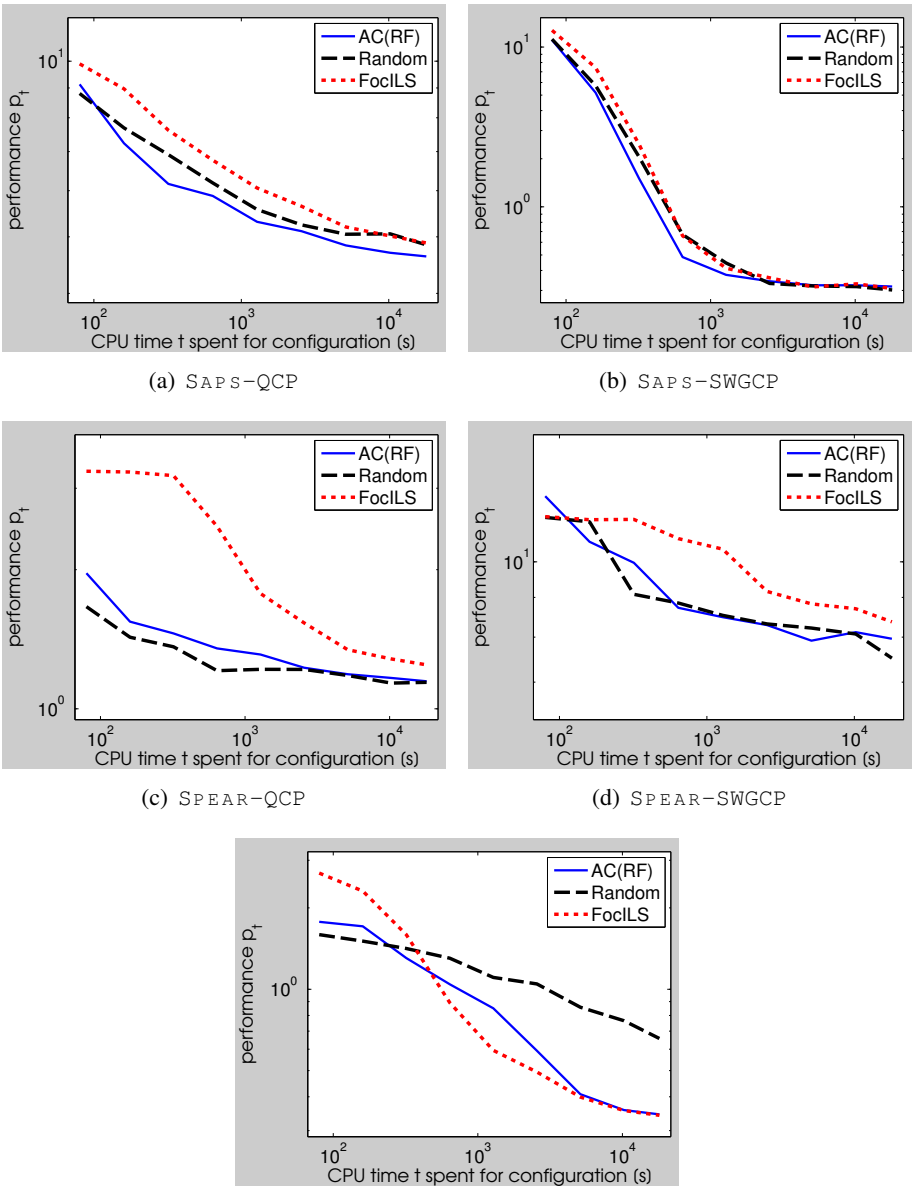
a blocking scheme. In the future, thus, we plan to study the quality of models learned from the data gathered in the sequential optimization process.

By far the clearest distinction between the different models is for scenario CPLEX-REGIONS100, the most complex configuration scenario. As we will see later in this section, random search alone did not find very good configurations for this scenario, such that AC variants based on poor models would also yield poor results. For this scenario, the default runtime feature yielded best performance, followed by the first seven principal components of the 39 MIP features, the index feature, and no features at all; all pairwise differences were statistically significant. These results correlate well with EIC quality for that configuration scenario (see Figure 13.5 on page 220), but not with the quality of predictive ranks or RMSE. The other AC variant that showed consistently robust performance was the one based on the seven principle components of the 126 SAT / 39 MIP features.

Finally, we compared that latter version of ACTIVECONFIGURATOR (AC) against RANDOM* and FOCUSEDILS. We show the results of this comparison in Figures 13.12 and 13.13, and summarize them in Table 13.2. ACTIVECONFIGURATOR significantly outperformed both RANDOM* and FOCUSEDILS in two scenarios and was never significantly worse. FOCUSEDILS was significantly worse than RANDOM* for scenario SPEAR-SWGCP—this may be due to the improved intensification mechanism used in the latter. Importantly, CPLEX-REGIONS100, the configuration scenario with the largest parameter configuration space, clearly required an informed search approach: for this scenario, the more informed configurators, FOCUSEDILS and ACTIVECONFIGURATOR performed substantially better than RANDOM*.

13.6.3 Chapter Summary

In this chapter, we extended sequential model-based optimization (SMBO) methods to handle general configuration scenarios with multiple instances. We evaluated models based on different sets of instance features: no features, a generic feature based on the runtime of the algorithm default on the instance, a generic index feature, and sets of SAT / MIP features



(a) SAPS-QCP (b) SAPS-SWGCP

(c) SPEAR-QCP (d) SPEAR-SWGCP

(e) CPLEX-REGIONS100

Figure 13.12: Performance of ACTIVECONFIGURATOR, RANDOM* and FOCUSEDILS for BROAD configuration scenarios. We performed 25 runs of the configurators and show boxplots of their test performance $p_{test,t}$ for a varying time budget, t.

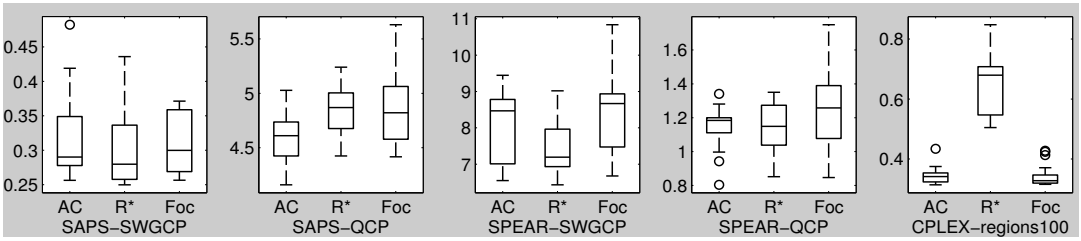


Figure 13.13: Performance of ACTIVECONFIGURATOR, RANDOM* and FOCUSEDILS for BROAD configuration scenarios. We performed 25 runs of the configurators and show boxplots of their test performance $p_{test,t}$ for a time budget of $t = 18\,000$ seconds.

Scenario	AC(RF)	RANDOM*	FocusedILS	significant
SAPS-SWGCP	0.32 ± 0.06	$\mathbf{0.30 \pm 0.05}$	0.31 ± 0.04	–
SAPS-QCP	$\mathbf{4.63 \pm 0.25}$	4.84 ± 0.23	4.88 ± 0.38	1/2, 1/3
SPEAR-SWGCP	7.96 ± 0.94	$\mathbf{7.52 \pm 0.80}$	8.37 ± 0.93	2/3
SPEAR-QCP	1.15 ± 0.12	$\mathbf{1.14 \pm 0.15}$	1.24 ± 0.22	1/3
CPLEX-REGIONS100	$\mathbf{0.34 \pm 0.03}$	0.66 ± 0.10	$\mathbf{0.34 \pm 0.03}$	1/2, 2/3

Table 13.2: Quantitative comparison of configurators for BROAD scenarios. We performed 25 runs of the configurators and computed their test performance $p_{test,t}$ (PAR over $N = 1\,000$ test instances using the methods' final incumbents $\boldsymbol{\theta}_{inc}(t)$) for a configuration time of $t = 18\,000$s. We give mean $\pm$ standard deviation across the 25 runs. Column "significant" lists the pairs of configurators for which a Mann Whitney U test judged the performance difference to be significant with confidence level 0.05; '1' stands for AC(RF), '2' for RANDOMSEARCH, and '3' for FOCUSEDILS. Figure 13.13 visualizes this data.

taken and extended from the literature. In terms of model quality, the generic runtime feature performed surprisingly well. This feature and the first seven principal components of the SAT / MIP features yielded the most robust performance across configuration scenarios.

We also evaluated the quality of predictions for combinations of parameter configurations and instances. Here, we found promising results for predicting the runtime of previously-unseen configurations on training instances, for predicting the runtime of training configurations on unseen test instances, and for predicting the runtime of unseen configurations on unseen test instances. In some cases, especially for configuration scenarios with large cutoff times, we achieved very good predictions, reaching rank correlations between predicted and actual runtimes of around 0.95. Overall, these results are very promising for future work on modelling instance hardness and on per-instance approaches.

Finally, we evaluated variants of ACTIVECONFIGURATOR (AC) based on different sets of features. We found that for most of the BROAD configuration scenarios, the choice of features was not very important and attributed this to the robustness of the AC framework. For the most complex BROAD configuration scenario, CPLEX-REGIONS100, AC based on the default runtime feature performed best, closely followed by a variant based on the 39 MIP features. Comparing that latter variant to FOCUSEDILS and RANDOM*, we found that for each of the

five configuration scenarios it yielded state-of-the-art performance, significantly outperforming FOCUSEDILS in two configuration scenarios and, alongside FOCUSEDILS, clearly outperforming RANDOM* for the most complex BROAD configuration scenario, CPLEX-REGIONS100.

Part V

Conclusion

—in which we put our contributions into perspective and offer advice to the practitioner

Chapter 14

Conclusion

I believe that if you show people the problems and you show them the solutions they will be moved to act.
—Bill Gates, American industrialist

In this thesis, we discussed the problem of automated algorithm configuration: given an algorithm, a set of parameters for the algorithm, and a set of input data, find parameter values under which the algorithm achieves the best possible performance on the input data.

In Part I, we motivated and formally defined the problem (Chapter 1), and discussed related work (Chapter 2).

In Part II, we introduced seven sets of configuration scenarios, centered around state-of-the-art algorithms for solving the propositional satisfiability (SAT) problem and the mixed integer programming (MIP) problem (Chapter 3) and introduced an empirical analysis approach for studying these configuration scenarios (Chapter 4).

In Part III, we introduced our model-free PARAMILS framework for algorithm configuration (Chapters 5 and 7), and discussed a variety of "real-life" applications of PARAMILS (Chapters 6 and 8). Most notably, in our case study for configuring the tree search SAT solver SPEAR for large industrial software verification and bounded model checking instances, we achieved average speedups in mean runtime of a factor above 500 for software verification and of 4.5 for bounded model checking, thereby substantially advancing the state of the art (Chapter 6). Furthermore, we automatically optimized 63 parameters of CPLEX, the most widely-used commercial optimization tool in industry and academia. Here, we achieved substantial speedups over the well-engineered CPLEX default settings, in some domains exceeding a factor of ten (Section 8.1).

In Part IV, we studied an alternative framework for algorithm configuration, Sequential Model-Based Optimization (SMBO). Previous to our work, SMBO methods have been limited to the optimization of numerical parameters. Within this scope, we evaluated two existing SMBO methods, SKO and SPO (Chapter 9), studied and improved the components of SPO (Chapter 10), and finally gained further significant improvements by using alternative response surface models in our novel ACTIVECONFIGURATOR framework (Chapter 11). Then, we extended the scope of SMBO to include the important case of categorical parameters (Chapter 12) and multiple instances (Chapter 13), both vital in general algorithm configuration.

We conclude this thesis by giving advice to practitioners, discussing our findings about the dimensions of algorithm configuration, laying out future work, and providing a longer-term outlook.

14.1 General Advice for Practitioners

We now give some general advice to practitioners who would like to apply automated algorithm configuration procedures, such as those we introduced in this thesis, for their problems.

14.1.1 Necessary Ingredients To Apply Algorithm Configuration

In order to apply an automated algorithm configuration procedure, a practitioner must supply the following ingredients.

A parameterized algorithm $\mathcal{A}$ It must be possible to set $\mathcal{A}$'s configurable parameters externally, *e.g.*, in a command line call. Often, a search for hard-coded parameters hidden in the algorithm's source code can lead to a large number of additional parameters to be exposed.

Domains for the parameters Algorithm configurators need to know the allowed domains Θ_i for each parameter θ_i. Depending on the configurator, it may be possible to include additional knowledge about dependencies between parameters, such as the conditional parameters supported by PARAMILS. Some configurators, such as PARAMILS as well as ACTIVECONFIGURATOR for mixed numerical/categorical problems, require a discretization of numerical parameters to a finite domain. Depending on the type of parameter, a uniform spacing of values or some other spacing, such as uniform on a log scale, is typically reasonable.

A set of problem instances The more homogeneous the problem set of interest, the better we can expect any algorithm configuration procedure to perform on it. While it is possible to configure an algorithm for good performance on rather heterogeneous instance sets (e.g., on industrial SAT instances, as we did with SPEAR as reported in Section 6.3.2), the results for a homogeneous subset of interest can be expected to improve when we configure on instances from that subset (see Section 6.3.3). Whenever possible, the set of instances should be split into disjoint training and test sets in order to safeguard against over-tuning. When configuring on a small and/or heterogeneous benchmark set, configuration procedures might not find configurations that perform well on an independent test set.

An objective function While we optimized median performance in our first study on algorithm configuration (Hutter et al., 2007b), we have since found cases where optimizing median performance led to parameter configurations with good median but poor overall performance. In these cases, optimizing for mean performance yielded more robust parameter configurations. However, when optimizing mean performance one has to define the cost for unsuccessful runs. In this thesis, we have penalized such runs by our penalized average runtime (PAR)

criterion, which counts timeouts at time κ_{max} as $a \cdot \kappa_{max}$ with a constant $a \geq 1$; we set $a = 10$ throughout this thesis. How to deal with unsuccessful runs in a more principled manner is an open research question.

A cutoff time for unsuccessful runs The smaller the maximal cutoff time, κ_{max}, for each run of the target algorithm is chosen, the more quickly any configuration procedure will be able to explore the configuration space. However, choosing too small a cutoff risks the failure mode we experienced with our CPLEX-QP scenario (see Section 8.1.2). Recall that there, choosing $\kappa_{max} = 300$ seconds as a timeout yielded a parameter configuration that was very good when judged with that cutoff time, but performed poorly for longer cutoffs. In all of our other experiments, parameter configurations performing well with low cutoff times (often as low as $\kappa_{max} = 5$ seconds) turned out to scale well to harder problem instances. In many configuration scenarios, in fact, we noticed that our automatically-found parameter configurations showed much *better* scaling behaviour than the default configuration. We attribute this trend to our use of (penalized) mean runtime as a configuration objective.[1] In our problem formulation, κ_{max} is a user-defined quantity, and how to set it in a principled way is an open problem. We plan to develop approaches for tackling this problem in the future (see Section 14.3.3). In the meantime, we advise to use similar captimes κ_{max} as one would employ in manual algorithm design: high enough that good performance with cutoff time κ_{max} gives the algorithm designer confidence that performance will scale to their problem size of interest, and low enough for experiments to finish in a reasonable timeframe.

Computational resources The amount of (computational) time required for automated algorithm configuration clearly depends on the target application. If the target algorithm only has few parameters and takes seconds to solve instances from a homogeneous benchmark set of interest, in our experience a single half-hour configuration run can often suffice to yield good results and a time budget of five hours is rather conservative (see, *e.g.*, the case of SAPS-SWGCP in Figure 13.10(b) on page 232). In contrast, if many parameters are being optimized, different configurations perform well on different subsets of instances, or only results with a large cutoff time can reliably indicate performance on the instances of interest, then the time requirements of automated algorithm configuration grow. In our experiments with FOCUSEDILS, we also regularly perform multiple parallel configuration runs and pick the one with best *training* performance in order to deal with variance across configuration runs. We expect that the need for (and potential of) doing so is smaller for more robust configurators.

[1] The mean is often dominated by the hardest instances in a distribution. In manual tuning, algorithm developers typically pay more attention to easier instances, simply because repeated profiling on hard instances takes too long. In contrast, a "patient" automatic configurator can achieve better results because it avoids this bias.

14.1.2 Which Configuration Procedure to Apply When?

In this thesis, we discussed various types of algorithm configuration problems and configuration procedures. Here, we summarize which existing configurator can be expected to perform best for which type of configuration problem. We provide some general recommendations but note that the best available algorithm configuration procedures will almost certainly change over the next few years.

We discuss types of configuration scenarios in order of increasing complexity.

Optimization of Numerical Parameters of Deterministic Algorithms for Single Instances

We did not specifically target the conceptually simplest case: optimizing the performance of a deterministic algorithm on a single problem instance. For algorithms with continuous parameters, this problem can be addressed by the wide range of existing algorithms for (deterministic) blackbox function optimization. On the model-based side, the most prominent approaches are variants of the efficient global optimization (EGO) algorithm (Jones et al., 1998), which we discussed in Section 9.1. On the model-free side, there exist many different approaches, a good sample of which were compared in the 2009 GECCO workshop on blackbox optimization benchmarking.[2] In this comparison, CMA-ES (Hansen and Ostermeier, 1996; Hansen and Kern, 2004), which we describe in Section 3.2.3, yielded overall best performance; this was also the outcome of the session on real-parameter optimization at the 2005 IEEE Congress on Evolutionary Computation.[3]

Deterministic algorithms with *categorical* parameters could be configured on single instances by using BASICILS(1) or ACTIVECONFIGURATOR with noise-free models and the parameter *maxR* set to 1 (see Section 5.2 for BASICILS and Section 11.5.1 for ACTIVECONFIGURATOR). We have, however, not yet performed any experiments for this case.

Optimization of Numerical Parameters of Randomized Algorithms for Single Instances

For this type of problem, we studied a variety of model-based approaches: sequential kriging optimization (SKO), sequential parameter optimization (SPO) and variants thereof, RANDOM*, and ACTIVECONFIGURATOR. Out of these methods, our experiments suggest that ACTIVECONFIGURATOR based on approximate Gaussian process (GP) models yields the best results. In particular, in Section 9.6.2, we saw that SPO was more robust "out-of-the-box" than SKO. We studied various versions of SPO: SPO$^+$ was much more robust than the previous variants SPO 0.3 and SPO 0.4 (Section 10.3.1), and SPO* had lower computational overhead than SPO$^+$ (Section 10.4). The simple approach RANDOM* performed similarly to SPO* (see Section 10.4), but in Section 11.5.2, we demonstrated that ACTIVECONFIGURATOR—based on its better models—significantly outperformed SPO* and RANDOM*.

It is of course possible to discretize the continuous domains and apply PARAMILS. However, as we showed in Section 12.3.3, the discretization of parameters can severely restrict

[2] http://www.isgec.org/gecco-2009/workshops.html

[3] http://www.lri.fr/~hansen/cec2005.html

the search space, handicapping any method that relies on user-defined discretizations.

Some model-free methods that partook in the aforementioned 2009 GECCO workshop on blackbox optimization benchmarking also apply for stochastic optimization. However, the stochastic optimization benchmarks in that workshop only evaluated the true quality of the best configuration a method ever evaluated, instead of an incumbent returned by the method. (The reasoning for this evaluation was that the problem of determining the best incumbent to return was perceived to be easier than the problem of finding good parameter settings in the first place.) In our applications, the determination of the incumbent was certainly not straightforward (see, *e.g.*, the sensitivity of SKO and, in some case, SPO). It is thus unclear how these methods would compare to ours; we plan to study this in the future.

Optimization of Categorical Algorithm Parameters for Single Instances

The list of possible configuration procedures for categorical parameters is shorter: variants of RANDOMSEARCH, PARAMILS, and ACTIVECONFIGURATOR(AC). In our experiments in Section 12.3.4, AC based on a random forest model performed best, closely followed by AC based on an approximate Gaussian process model. RANDOM* was worse, but still outperformed FOCUSEDILS; we attribute this at least in part to the new—and apparently superior—intensification mechanism shared by AC and RANDOM*.

Optimization of Categorical Algorithm Parameters for Multiple Instances

Finally, for configuration scenarios in which we optimize algorithm performance across a set of problem instances, we considered variants of RANDOMSEARCH, PARAMILS, and ACTIVECONFIGURATOR (based on random forests, AC(RF)). On our BROAD set of configuration scenarios, we showed that BASICILS outperformed RANDOMSEARCH (see Section 5.2.2) and that FOCUSEDILS outperformed BASICILS (see Section 5.3.3). Later, in Section 13.6.2, we showed that RANDOM* was surprisingly efficient even for target algorithms with many parameters, such as SPEAR(with 26 parameters), and significantly outperformed FOCUSEDILS in two of the five BROAD scenarios On average, AC(RF) performed best for these scenarios, in turn outperforming RANDOM* in two of the scenarios. As configuration scenarios get more challenging (in terms of search space size and sparsity of good configurations), there appears to be a need for more informed search methods: both FOCUSEDILS and AC(RF) clearly outperformed RANDOM* for the most complex BROAD scenario, CPLEX-REGIONS100 (see Section 13.6.2).

Other Cases

For the optimization of numerical algorithm parameters for multiple instances, two of our methods apply directly: AC(RF) and RANDOM*. We could also discretize parameters and use FOCUSEDILS. We have not carried out experiments in this domain yet, but would expect the discretization to cause a performance loss comparable to the one we found in our experiments for the single-instance case (see Section 12.3.3).

Likewise, for the optimization of mixed categorical/numerical parameters, we can either

apply RANDOM* or discretize parameters and then apply FOCUSEDILS or ACTIVECONFIGURATOR. RANDOM* showed excellent performance in many configuration tasks (see, *e.g.*, Section 13.6.2), but did not scale to the most complex one, the configuration of CPLEX. That case required more informed methods, such as FOCUSEDILS or ACTIVECONFIGURATOR. In the future, we plan to integrate direct support for mixed categorical/numerical parameters into our configuration procedures (see Section 14.3.2).

14.2 Dimensions of Algorithm Configuration

In Section 1.2.2, we identified three dimensions of algorithm configuration: (1) the sequential search strategy being used; (2) which instances to use and how many runs to perform for each of them; and (3) how to choose the cutoff time, after which unsuccessful algorithm runs are terminated.

14.2.1 Sequential Search Strategy: Model-free or Model-based?

We introduced two frameworks for algorithm configuration: a model-free approach based on iterated local search (Part III of this thesis) and a model-based approach that employs random forest (RF) or Gaussian process (GP) response surface models (Part IV of this thesis). Each of these two families of approaches has benefits and drawbacks. We discuss these in the following sections.

Advantages of Model-Free Approaches

Conceptual Simplicity Our model-free iterated local search approach is clearly simpler to understand and to implement than our sequential model-based optimization (SMBO) framework. In particular, the SMBO framework requires the construction of a response surface model. The construction of such a model is more complicated than it might appear at first sight. Firstly, the modelling mechanism must scale to tens of thousands of training data points. Secondly, parameter configuration space, $\mathbf{\Theta}$, is often high-dimensional (63-dimensional in the case of CPLEX) and good models need to be able to distinguish between important and unimportant parameters. Thirdly, often these parameters are mixed numerical/categorical, rendering most standard regression methods inapplicable. A final complication in learning these response surface models is that the time spent doing so is part of the overall time budget for configuration, and that we require a new model in each of the thousands of SMBO iterations we often perform. Thus, model construction has to be very efficient (in our experiments, it typically takes on the order of seconds). A second non-trivial step in SMBO is the optimization of the expected improvement criterion (EIC) in each SMBO iteration. Just like in the algorithm configuration problem, this optimization takes place in parameter configuration space. Any (typically model-free) optimization procedure applied for this task could also be used for the original algorithm configuration problem (with the difference that—unlike evaluations of a configuration's cost—EIC evaluations are cheap). Thus, the model-free framework is clearly conceptually simpler.

Robustness During the development of our model-based configuration procedures we have observed pathological behaviour much more frequently than in the development of PARAMILS. We observed one example of such pathological behaviour when we used SKO without a log transformation in Section 9.6.2. We attributed these problems to the fact that SKO computes its incumbent based on its model. In our experiments with our own models, we repeatedly observed that—due to overconfident or simply poor model predictions—this did not yield a robust configuration procedure. For that reason, throughout our work on model-based algorithm configuration we have relied on separate intensification procedures to determine reliable incumbents. Another prominent problem we observed are the failure modes of SMBO we discussed in Section 11.2.2. In practice, it appears that these failure modes are largely alleviated by interleaving random parameter configurations. Thus, while "pure" model-based techniques are often very sensitive, they appear to be robust in combination with a good intensification mechanism and interleaved random target algorithm runs.

We have also observed that FOCUSEDILS's performance is not very robust across repeated runs. We believe that this is at least in part due to unfortunate orderings in its $\langle$instance, seed$\rangle$ lists. Each FOCUSEDILS run prominently uses the first few instances in its list; if those are not very representative of the whole instance set performance typically suffers.

Invariants with Respect to Transformations of the Cost Statistic In our model-free iterated local search framework, we solely compare parameter configurations with respect to the (user-defined) cost statistic, making the framework invariant to monotone transformations of that statistic. In particular, PARAMILS (and all variants of RANDOMSEARCH) are invariant to any transformation that, for all N, $\boldsymbol{\theta}_1$, and $\boldsymbol{\theta}_2$, preserves the relative rankings of empirical cost statistics $\hat{c}_N(\boldsymbol{\theta}_1)$ and $\hat{c}_N(\boldsymbol{\theta}_2)$. This is not the case for model-based approaches, since transformations also affect the response surface model.

Conditional Parameters In our model-free search framework, it is straightforward to support conditional parameters, that is, parameters that are only relevant depending on the setting of other, higher-level parameters. In particular, in PARAMILS, we simply adapted the neighbourhood relation to exclude parameter configurations that only differ in parameters that are inconsequential given the settings of the other parameters.

For most types of response surface models, it is not straightforward to exploit knowledge about conditional parameters. In principle, it seems possible to construct kernel functions for Gaussian process models that take conditional parameter dependencies into account. Likewise, in random forests, one can restrict allowable splits at a node to those guaranteed to be relevant given the values of split variables higher up in the tree. Finally, in order to exploit conditional parameters, the optimization of the expected improvement criterion (EIC) would also have to be adapted (*e.g.*, by adapting the neighbourhood in the local search for EIC optimization similarly as we did in PARAMILS). However, we have not implemented any of these methods in the model-based framework since they are much less straightforward than in the model-free case.

For SPEAR and CPLEX, which have some conditional parameters, this limitation of the model-based framework did not seem to negatively affect performance. However, in

the SATENSTEIN framework most of the parameters are conditional. We thus expect that configurators exploiting these conditional parameter dependencies will perform better. For this reason, we have not yet experimented with SATENSTEIN in our model-based framework.

Advantages of Model-Based Approaches

Our main motivation for pursuing model-based approaches was that response surface models provide a host of information that is not available when using model-free search. In particular, response surface models can be used to interpolate performance between parameter configurations, and to extrapolate to previously-unseen regions of the configuration space. They can also predict instance hardness and the probability of solving an instance as a function of the captime; these predictions can be used in future, more sophisticated, algorithm configuration procedures (see Section 14.3.3). They can also be used to quantify the importance of each parameter, interactions between parameters, and interactions between parameters and instance features (see Section 14.3.4). Finally, these models can also be used to determine whether different configurations can be expected to perform well on different instances (see Section 14.3.5).

Note that, even without these more sophisticated components, ACTIVECONFIGURATOR already achieved the best performance for various types of configuration scenarios, in particular, the SINGLEINSTCONT scenarios (Section 11.5.2), the SINGLEINSTCAT scenarios (Section 12.3.4), and, on average, the BROAD scenarios (Section 13.6.2). Thus, we believe that there is much promise in developing these model-based approaches further. Overall, we expect that hybrids between model-free and model-based approaches can be constructed that combine the strengths of either approach (see Section 14.3.2).

14.2.2 Which Instances to Use and How Many Runs to Perform?

The second dimension of algorithm configuration contains two parts: (1) which problem instances should we use and (2) how many runs should we use for each of them to evaluate a parameter configuration? We address these subquestions in turn.

Blocking on Instances and Seeds

When we compare two parameter configurations $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ based on N runs of each of $A(\boldsymbol{\theta}_1)$ and $A(\boldsymbol{\theta}_2)$, intuitively, the variance in this comparison is lower if we use the same N instances (and seeds) for the two configurations. However, if we have available the result of $N' > N$ runs for one of the configurations, then it is unclear whether it is better to use this additional information to improve the estimate for that configuration or to only use the matching N runs to enable a blocked comparison.

As we showed in Section 13.6.1, in 3 of the 5 BROAD configuration scenarios (in which we optimize performance across a set of instances), an intensification procedure based on blocking performed better than one without. In contrast, in 3 of the 5 SINGLEINSTCAT configuration scenarios (in which we only optimize performance for a single instance), it was better *not* to use the blocking scheme. (In the remaining scenarios both intensification procedures

performed similarly). These results suggest that blocking on ⟨instance, seed⟩ combinations helps in the presence of multiple (rather different) instances. In contrast, for single instances the variance reduction due to blocking on seeds was smaller than the additional variance due to not using all $N' > N$ available runs. However, a more comprehensive study is needed to conclusively demonstrate these points.

Which Instances To Use

We showed in Chapter 4 that in some configuration scenarios certain instances are not informative for comparing parameter configurations. In particular, very hard problem instances that, within the captime, cannot be solved by any considered parameter configuration, only slow down the search. In Chapter 6, we manually removed very hard training instances in order to speed up the configuration process. We have not yet implemented a mechanism for automating this step but discuss the possibility in Section 14.3.3. In fact, we plan to implement a much more general procedure that still allows the use of very hard instances for the evaluation of some very good configurations predicted to stand a good chance at solving them.

FOCUSEDILS (see Section 5.3) uses a fixed ordering of instances (and associated seeds). This ordering differs across its runs, resulting in a large variance across runs. We described one extreme case of this problem for configuration scenario CPLEX-MIK in Section 8.1.2. There, one FOCUSEDILS run selected a parameter configuration that ended up not solving *any* test instance within 300 seconds, whereas nine other independent FOCUSEDILS runs yielded parameter configurations with average test set runtimes of below 2 seconds. More generally, we also observed that FOCUSEDILS's performance varied more than that of BASICILS. We exploited this variance by performing k FOCUSEDILS runs and using the configuration found in the run with best training performance. However, this causes a k-fold overhead, which we would like to avoid.

Further evidence that the fixed order of instances used in FOCUSEDILS is suboptimal comes from our experiments in Section 13.6.2. There, the simple procedure RANDOM* (based on intensification Procedure 10.5, which uses a different random subset of instances for each comparison) performed *better* than FOCUSEDILS in 2 of 5 scenarios. Only for configuration scenario CPLEX-REGIONS100, which is hardest in terms of size of configuration space while featuring a very homogeneous instance set, FOCUSEDILS performed clearly better than RANDOM*.

Adapting the Number of Runs

We demonstrated in Chapter 4 that there exists no single best fixed choice of the number of runs to perform for the evaluation of each parameter configuration considered. We also showed that FOCUSEDILS, which adapts the number of runs, $N(\boldsymbol{\theta})$, for the evaluation of each configuration, $\boldsymbol{\theta}$, often outperforms BASICILS, which uses a fixed N for each configuration (see Section 5.3). For this reason, in our model-based framework we only constructed intensification procedures that adaptively selected the number of runs to perform for each configuration. In our experiments for the configuration of algorithms with categorical parameters for single problem instances (see Chapter 12), ACTIVECONFIGURATOR and even

RANDOM* performed better than FOCUSEDILS. We attribute this in part to their new and improved intensification procedure (Procedure 10.5); we plan to integrate that procedure into PARAMILS in the near future.

14.2.3 Which Captime To Use?

We showed in Chapter 4 that there exists no optimal fixed captime for all configuration scenarios. From Chapter 7, it is also clear that adaptive capping helps to speed up a range of configuration procedures. This speedup was most pronounced in configuration scenarios with a large spread in the quality of configurations. In the best case, capping can reduce the time spent for evaluating the worst configurations to the time needed for the best configurations. Consequently, the speedups due to adaptive capping were most pronounced for those configuration procedures performing many runs on poor configurations, RANDOMSEARCH(N) and BASICILS(N).

14.3 Future Work

We have answered a variety of important questions about algorithm configuration, but, obviously, many open questions remain. Indeed, algorithm configuration is a rich and fruitful field of study that offers more opportunities for future work than could be handled by one person or even a single research group. We thus hope that other researchers will join in the effort to push existing analysis, techniques, and applications even further.

14.3.1 Configuration Scenarios and Their Analysis

Young areas of research require standard benchmarks to ensure constant progress is being made. We plan to collect existing benchmarks, make them publicly available, extend them, and study them with our empirical analysis techniques.

Constructing a Suite of Standard Benchmarks for Algorithm Configuration

The configuration scenarios we used throughout this thesis can be used as a core for a future suite of standard benchmarks for algorithm configuration. These seven sets of configuration scenarios, introduced in Chapter 3, differ in many important characteristics, such as the problems being solved, the number of target algorithm parameters, the type (numerical/categorical) of parameters, and the homogeneity of instance sets. We would like to create a benchmark suite in which we systematically vary these and other characteristics. We also plan to include new, interesting, configuration scenarios from other application domains, as well as configuration scenarios used in other lines of work on algorithm configuration (see, *e.g.*, Birattari, 2004).

One issue in the development of algorithm configuration procedures is that typical runs on interesting configuration scenarios are computationally expensive since they require tens of thousands of target algorithm runs on instances of hard computational problems. We plan to employ predictive models of runtime, such as those introduced in Chapter 13 and visualized in Section 13.5, to construct *surrogate configuration scenarios*, in which we optimize *predicted*

algorithm performance instead of actual performance. In the surrogate version of a given configuration scenario, we would simply replace each call of the target algorithm (with a specified combination of parameter configuration, instance, and seed) by a sample from the predictive distribution of the model (for the requested combination of parameter configuration and instance, using the seed to draw the sample). We expect that the use of such surrogate configuration scenarios will facilitate the development of configuration procedures since it (1) substantially reduces the runtime necessary for each configuration run and (2) provides "ground truth" data to evaluate performance of the configurator. Surrogate scenarios that more closely resemble the underlying actual configuration scenarios can be expected to be more realistic, and we would thus like to improve model quality further. Of course, if a configurator was to be developed chiefly using surrogate configuration scenarios, its performance would need to be verified on a suite of actual configuration scenarios.

Empirical Analysis of Configuration Scenarios

Algorithm configuration is a complex problem and we expect that many more insights can be be gained by extending the empirical analysis we provided in Chapter 4. In particular, we plan to *quantify* the qualitative measures we introduced in that chapter. Once quantified, such measures can be used to characterize configuration scenarios and to study which types of configuration procedures work well for which types of configuration scenarios (and, eventually, *why* they work well).

We also plan to apply our predictive models to gather a matrix of *predicted* runtimes and apply our analysis techniques on that matrix. Since predictive models can be built online (during the runtime of a configurator) without the need for any additional runs of the target algorithm, this promises to yield an approximate online version of our offline empirical analysis tools. Approximations of quantitative versions of our measures can then be used during algorithm configuration to make online decisions about which configuration procedures to use.

14.3.2 Search Strategy

In this thesis, we have focused on model-free techniques based on iterated local search and model-based techniques based on Gaussian process and random forest models. Other approaches and combinations of methods could be applied beneficially for algorithm configuration.

Empirical Evaluation of Different Configuration Procedures

In this thesis, we have compared our new configuration procedures to some existing ones, namely the CALIBRA system (see Section 5.4), Sequential Kriging Optimization (SKO, see Chapter 9), and Sequential Parameter Optimization (SPO, see Chapters 9 and 10). We have not compared it to other configuration procedures since, to our best knowledge, no other procedures exist that can handle the very complex configuration scenarios we studied (optimizing target algorithms with many categorical parameters, such as CPLEX). However,

for algorithms with less parameters, or with continuous parameters, other procedures *do* exist. In particular, we would like to compare the methods studied in this thesis to racing algorithms, such as F-Race (Birattari et al., 2002; Birattari, 2004) and iterated F-Race (Balaprakash et al., 2007). For optimizing target algorithms with only numerical parameters, we would also like to compare to prominent model-free blackbox optimization approaches, such as CMA-ES (Hansen and Ostermeier, 1996; Hansen and Kern, 2004).

Mixed Categorical/Numerical Optimization

In practice, most algorithms with categorical parameters also have some numerical parameters. Throughout this thesis, in such cases we discretized numerical parameters, thereby restricting the search space to potentially exclude the optimal configuration. In the case of SAPS, we showed in Section 12.3.3 that the loss incurred by such a discretization was rather substantial (greater than the performance differences due to using different configuration procedures).

In future work, we would like to extend our model-free and model-based methods to the optimization of mixed categorical/numerical parameters to remove the need for a discretization. In the model-free case, one may consider hybrids of continuous optimization algorithms, such as CMA-ES, for optimizing the numerical parameters, and local search for optimizing the discrete parameters. In the model-based case, our random forest and approximate Gaussian process models can already handle mixed numerical/categorical inputs. What is missing in the current version of ACTIVECONFIGURATOR is a method for optimizing the expected improvement criterion (EIC) in mixed categorical/numerical configuration spaces. For this subproblem, we could apply a similar method as just discussed for the model-free case.

Hybrids of Model-free and Model-based Search

We believe that there is significant room for combining aspects of the methods studied here with concepts from related work in algorithm configuration and related problems. In particular, we believe it would be fruitful to integrate statistical testing methods—as used, *e.g.*, in F-Race—into PARAMILS or ACTIVECONFIGURATOR. Such statistical tests could, for example, be applied in the intensification procedure to decide how many runs should be used to evaluate each configuration. An effective intensification mechanism would quickly discard poor configurations and yield good performance if the process used for selecting promising configurations to be evaluated returns good configurations at least every now and then. Given such an intensification mechanism, we could easily hybridize model-free and model-based approaches, taking advantage of their respective strengths.

Automatic Self-Configuration

We have already performed an experiment for the automatic self-configuration of PARAMILS (see Section 8.2). That experiment, while automatically yielding a parameter configuration comparable to the one identified manually, did not result in large performance improvements. In the case of ACTIVECONFIGURATOR or hybrid versions of algorithms, we would expect this to change. ACTIVECONFIGURATOR has a large number of parameters, for many of which

we simply used a default based on preliminary experiments. Optimizing these parameters on a set of interesting configuration scenarios is hard since each single configuration run already requires substantial time. In this context, we could use the surrogate configuration scenarios discussed in Section 14.3.1 to substantially reduce the time required for each configuration run. We plan to apply this approach in the future and expect that, especially when optimizing performance for a homogeneous set of configuration scenarios, significant performance improvements are possible.

14.3.3 Adaptive Approaches for Selecting Captime and Instances

Our predictive models of runtime can be applied to construct more sophisticated methods for adaptively selecting the instances and captimes to use in each run of the target algorithm.

Adaptive Capping in the Model-based Framework

So far, we have not applied adaptive capping in the model-based framework even though it is straightforward to implement the adaptive capping mechanism introduced in Chapter 7. In fact, compared to the PARAMILS framework this is *easier* in our model-based framework since there—as in RANDOMSEARCH—all pairwise comparisons of parameter configurations involve the incumbent; thus, the trajectory-preserving and aggressive capping variants we introduce for PARAMILS in Section 7.2 are identical in the model-based framework.

The part that is not straightforward is to build models for training data that includes unsuccessful runs terminated before cutoff time κ_{max}. We plan to apply methods from the survival analysis literature to build models for regression under such *partly right-censored* data. In particular, we are experimenting with an adaptation of our random forest model along the lines of work by Segal (1988). We are also experimenting with Gaussian process models under censoring (Ertin, 2007) and with more generic methods to handle censored data in regression models (Schmee and Hahn, 1979). The capability of dealing with partially censored data also opens up the possibility to initialize the model based on a large number of runs with a low captime.

Active Selection of Problem Instances

As discussed in Section 4.3, a mechanism to detect very hard instances and reduce the time spent with them could be a useful component of both model-free and model-based procedures. In particular, in our case study for configuring SPEAR on benchmark set BMC (see Section 6.3.3), we manually eliminated too hard instances from the training set in order to speed up FOCUSEDILS; we would like to eliminate the need for such manual steps.

Predictive models of algorithm runtime for $\langle$parameter configuration, instance$\rangle$ combinations can, in principle, be used for this purpose. One could simply predict a matrix of runtimes as we did in Section 13.5 and avoid instances predicted to be hard for every configuration. However, when using this information in a heuristic mechanism for instance selection, we would need to take care not to bias the search away from configurations that perform well on instances (wrongly) predicted to be uniformly hard.

We also plan to experiment with an existing approach for selecting the most informative instance for each run. In Section 13.2.1, we reviewed work by Williams et al. (2000), in which they used Gaussian process models to optimize a function across the setting of environmental variables—in our case, problem instances. In that work, they also suggested an approach for actively selecting the most informative problem instance for each run. Applied to algorithm configuration, their mechanism would first choose a parameter configuration to evaluate and then select the problem instance which, on expectation (over the predicted distribution of runtime), most reduces the predictive variance for that chosen configuration. We plan to implement this approach in our RF model framework and extend it with a criterion that takes into account the expected time for runs on each problem instance. We could then, *e.g.*, select the instance promising the highest information gain per time spent.

It might also be possible to apply a similar criterion to select the next parameter configuration, thereby replacing the expected improvement criterion (EIC). The problem with EIC is robustness: it can fail miserably if runs for the configuration with maximal EIC score basically do not change the model (see Section 11.2.2). Including an "expected change of model" term in the EIC might help avoid this problem.

Active Selection of Captime

In some scenarios, it is important to terminate unsuccessful runs before the maximal captime κ_{max} is over. Our current techniques always use the maximal captime κ_{max} unless unsuccessful runs with a lower captime $\kappa \leq \kappa_{max}$ would already render the current configuration provably worse than the incumbent (based on the N runs used in the comparison). In our current implementation, every run for the incumbent uses the full captime, κ_{max}. With too large (user-defined) maximal captimes (*e.g.*, $\kappa_{max} = \infty$), this approach can yield poor results. In the extreme, if κ_{max} is set to ∞ and the first target algorithm run (using the default) takes longer than the total time budget, then the procedure simply returns that default, even if every single other configuration is clearly better. This simple example demonstrates the need to terminate some runs before the maximal captime, κ_{max}, is over. One heuristic approach could be to internally set κ'_{max} differently from the problem definition's κ_{max}, and increase κ'_{max} over time.

Alternatively, an approach similar to that described in the previous section could be used to *actively* select the captime for each run. In such an approach, one would select the captime that yields the highest expected information gain, normalized by the expected time spent. This could be combined with the selection of instances (to select the most promising $\langle$instance, captime$\rangle$ combination to use for evaluating a given configuration), or even with the selection of both the next configuration and the instance to be used. Models that explicitly integrate information about which runs were terminated unsuccessfully might also be able to extrapolate performance beyond the internally-used maximal captime, κ'_{max}.

14.3.4 Scientific Studies of Target Algorithm Performance

Response surface models can not only be used to determine promising configurations in a sequential optimization framework. Another important aspect of predictive models of

algorithm performance, is to provide feedback to the algorithm designer. For example, they can be used to determine how much algorithm response depends on the setting of single parameters, and to which degree parameters interact (Jones et al., 1998; Santner et al., 2003; Bartz-Beielstein, 2006).

Compared to typical response surface models, we added the additional dimension of instance features; this increased the potential of scientific studies of algorithm performance. Assuming that the response surface model fits the data well (which can be verified by simple cross-validation), we can use a joint model of instance features and parameter values to detect interactions between parameters and features. The automated determination of such interactions can suggest and aid scientific studies of what makes a problem hard and which approaches make a hard problem easy. We thus hope that the use of advanced response surface models will lead to insights into algorithms and eventually to algorithm improvements. It would be fruitful to provide general tools to support this type of analysis.

14.3.5 Per-Instance Approaches

In this thesis, we have concentrated on finding a single fixed configuration with overall good performance across a set or distribution of instances. This type of algorithm configuration is relevant in many real-world applications of algorithm configuration, in which all instances to be solved typically come from a single problem domain (*e.g.*, our application to industrial verification problems in Chapter 6).

In some applications, however, instances originate from a variety of sources and have rather different characteristics. In particular, the best parameter configuration for one instance might perform poorly for another, such that there is potential in using a different configuration for each instance. The decision about which configuration to use for a new problem instance needs to be comparably cheap since it is part of the time for solving the instance. In this context, the polytime-computable instance features we discussed in Sections 13.1.1 and 13.1.2 can be used.

There are at least two different possibilities for the instance-based selection of parameter configurations. The first approach works as follows. In an offline stage, carry out a large number of target algorithm runs on many instances; this step gathers data for learning a model that predicts algorithm performance for combinations of parameter configurations and instances. Then, given a previously-unseen problem instance, compute the instance's features (a polytime operation), and determine a parameter configuration with low predicted runtime for that instance. In large configuration spaces, $\boldsymbol{\Theta}$, we cannot afford to evaluate model predictions for every $\boldsymbol{\theta} \in \boldsymbol{\Theta}$, but rather have to apply a search through $\boldsymbol{\Theta}$. This is very similar to our optimization of the expected improvement criterion (see Section 12.3.1). Hence, while at first glance it seems rather infeasible to perform an optimization in parameter configuration space for each single instance to be solved, in practice this optimization step (for which we do not require optimality) can be performed efficiently. Based on the good performance of some of our models (see, *e.g.*, Figure 13.8(d) on page 228), we believe there is much promise in this approach and plan to pursue it further in the future.

The second approach works as follows. In an offline stage, cluster the training instances into k clusters with respect to their features, and employ some automated algorithm config-

uration procedure to find a good parameter configuration for each of the k instance clusters. This process outputs k parameter configurations, which can be seen as k algorithms. Thus, the portfolio-based algorithm selection scheme of SATzilla (Xu et al., 2008) directly applies. At its core, this scheme is based on a separate predictive model of runtime for each of the k candidate algorithms. Given a new, previously-unseen, problem instance, this approach would then select the candidate algorithm predicted to perform best on the instance.

We have already applied an early version of the first approach for automatically setting two of SAPS's continuous parameters, as well as the single parameter of the WalkSAT variant Novelty$^+$, on a per-instance basis (Hutter et al., 2006). For a mix of random and structured instances that required very different settings of the Novelty$^+$ parameter, we achieved a speedup factor of 10 over the best fixed configuration. However, in that work, the learning problem was rather easy (2 continuous parameters for SAPS, 1 for Novelty$^+$), we only considered few possible parameter configurations (30 for SAPS, 6 for Novelty$^+$), and we had access to a large training data set (1 000 algorithm runs for each parameter configuration). Having scaled up the models to handle problems with many categorical parameters, we now plan to revisit this problem for more strongly parameterized algorithms. We are actively working on both this and the second approach to instance-based algorithm configuration; which of the two will perform better (and under which circumstances) is yet an open question.

14.4 Outlook

As stated earlier, there are more opportunities for research than one person or even a single group can handle alone. In the computer science department of the University of British Columbia, we have started an initiative involving 2 professors, 3 PhD students and 2 MSc students working on algorithm configuration and per-instance algorithm selection. This group is partially funded by the MITACS project "Automated Design of Heuristic Algorithms from Components" (PI Dr. Hoos, co-PI Dr. Leyton-Brown), which was largely conceived as a result of the work reported in this thesis. The author of this thesis will remain in this group for a postdoctoral fellowship, following many of the lines of future research identified above.

In the coming years, we expect the area of algorithm configuration to mature further. Currently mostly a collection of configuration procedures (each of which is independently developed and evaluated), we hope that algorithm configuration will grow into a more established area of research with standard benchmarks, reference implementations, standard interfaces and problem descriptions, as well as readily-available collections of experimental data. We plan to join forces with other groups to construct a comprehensive benchmark suite for algorithm configuration (see Section 14.3.1) and compare existing configuration procedures (see Section 14.3.2). We are also working towards simplifying the application of algorithm configuration procedures as much as possible to enable practitioners to easily use them "out-of-the-box". In particular, work in our group has already commenced on a new web-based interface that will enable researchers and practitioners to easily employ algorithm configuration procedures for their purposes. This interface will enable the use of PARAMILS and ACTIVECONFIGURATOR, as well as a variety of other configuration and general optimization procedures, such as F-Race and CMA-ES. We are also reaching out to researchers in a variety of fields to popularize

the use of algorithm configuration by demonstrating that it can help to quickly construct algorithms that challenge the state-of-the-art in the respective field of study.

In the medium term, we firmly believe that automated algorithm configuration methods, such as those introduced in this thesis, will play an increasingly prominent role in the development of high-performance algorithms and their applications, and will be widely used in academia and industry. As we have demonstrated in this thesis, automated configuration procedures often require only a fraction of the time human experts need to establish good parameter settings, while at the same yielding much better results. Especially in an industrial context, where human expert time is costly, this combination of reduced development times and improved performance is likely to attract the attention of decision makers. In the context of the MITACS project mentioned above, we are closely collaborating with an industrial partner, Actenum Corporations[4], a company that focuses on industrial decision support and scheduling. Actenum already routinely applies our PARAMILS framework to reduce algorithm development times and improve algorithm performance. In an academic context, several research groups are also already using PARAMILS, and once the interface to apply algorithm configuration has been simplified and other configuration procedures have been integrated, we expect interest to grow substantially.

In the long term, we see automated algorithm configuration procedures as an important part of the emerging field of *meta-algorithmics*, the study of algorithms that execute and reason about other algorithms. Other examples of meta-algorithmic approaches are algorithm portfolios, per-instance algorithm selection techniques, reactive search algorithms, algorithm synthesis, and work in genetic programming that evolves programs. We believe that much progress could be derived from a unified view of the problems faced in these different areas of research. We also advocate joint initiatives integrating existing expertise from various areas, such as machine learning, statistics, artificial intelligence, constraint programming, computer science, operations research, and evolutionary algorithms. Finally, we believe that response surface models, such as those we constructed in the context of algorithm configuration, can be used in a variety of meta-algorithmic contexts. In particular, per-instance algorithm selection techniques, such as SATzilla (Xu et al., 2008), and dynamic algorithm portfolios (Gagliolo and Schmidhuber, 2006) already use similar predictive models, but do not yet reason about the setting of algorithm parameters. It is also possible to integrate features into these models that are acquired during algorithm runtime. Such models can then be used to change algorithms or parameter configurations in an online setting, *while* solving a new problem instance. Thus, we firmly believe that the study of response surface models in a meta-algorithmic context is a rich and fruitful research area with many interesting questions remaining to be explored.

[4] http://www.actenum.com/

Bibliography

Adenso-Diaz, B. and Laguna, M. (2006). Fine-tuning of algorithms using fractional experimental design and local search. *Operations Research*, 54(1):99–114.

Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F. P., Thomson, J., Toussaint, M., and Williams, C. K. I. (2006). Using machine learning to focus iterative optimization. In *Proceedings of the Fourth Annual International Symposium on Code Generation and Optimization (CGO-06)*, pages 295–305, Washington, DC, USA. IEEE Computer Society.

Aktürk, S. M., Atamtürk, A., and Gürel, S. (2007). A strong conic quadratic reformulation for machine-job assignment with controllable processing times. Research Report BCOL.07.01, University of California-Berkeley.

Andronescu, M., Condon, A., Hoos, H. H., Mathews, D. H., and Murphy, K. P. (2007). Efficient parameter estimation for RNA secondary structure prediction. *Bioinformatics*, 23:i19–i28.

Atamtürk, A. (2003). On the facets of the mixed–integer knapsack polyhedron. *Mathematical Programming*, 98:145–175.

Atamtürk, A. and Muñoz, J. C. (2004). A study of the lot-sizing polytope. *Mathematical Programming*, 99:443–465.

Audet, C. and Orban, D. (2006). Finding optimal algorithmic parameters using the mesh adaptive direct search algorithm. *SIAM Journal on Optimization*, 17(3):642–664.

Babić, D. (2008). *Exploiting Structure for Scalable Software Verification*. PhD thesis, University of British Columbia, Vancouver, Canada.

Babić, D. and Hu, A. J. (2007a). Exploiting Shared Structure in Software Verification Conditions. In Yorav, K., editor, *Proceedings of Haifa Verification Conference (HVC'07)*, volume 4899 of *Lecture Notes in Computer Science*, pages 169–184. Springer.

Babić, D. and Hu, A. J. (2007b). Structural Abstraction of Software Verification Conditions. In W. Damm, H. H., editor, *Computer Aided Verification: 19th International Conference, CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 366–378. Springer Verlag, Berlin, Germany.

Babić, D. and Hutter, F. (2007). Spear theorem prover. Solver description, SAT competition 2007.

Babić, D. and Musuvathi, M. (2005). Modular Arithmetic Decision Procedure. Technical Report TR-2005-114, Microsoft Research Redmond.

Balaprakash, P., Birattari, M., and Stützle, T. (2007). Improvement strategies for the F-Race algorithm: Sampling design and iterative refinement. In Bartz-Beielstein, T., Aguilera, M. J. B., Blum, C., Naujoks, B., Roli, A., Rudolph, G., and Sampels, M., editors, *4th International Workshop on Hybrid Metaheuristics (MH'07)*, pages 108–122.

Bartz-Beielstein, T. (2006). *Experimental Research in Evolutionary Computation: The New Experimentalism*. Natural Computing Series. Springer Verlag, Berlin, Germany.

Bartz-Beielstein, T., Lasarczyk, C., and Preuss, M. (2005). Sequential parameter optimization. In et al., B. M., editor, *Proc. of CEC-05*, pages 773–780. IEEE Press.

Bartz-Beielstein, T., Lasarczyk, C., and Preuss, M. (2008). Sequential parameter optimization toolbox. Manual version 0.5, September 2008, available at
`http://www.gm.fh-koeln.de/imperia/md/content/personen/lehrende/bartz_beielstein_thomas/spotdoc.pdf`.

Bartz-Beielstein, T. and Markon, S. (2004). Tuning search algorithms for real-world applications: A regression tree based approach. In Greenwood, G. W., editor, *Proc. of CEC-04*, pages 1111–1118. IEEE Press.

Bartz-Beielstein, T. and Preuss, M. (2006). Considerations of budget allocation for sequential parameter optimization (SPO). In et al., L. P., editor, *Proc. of EMAA-06*, pages 35–40.

Battiti, R., Brunato, M., and Mascia, F. (2008). *Reactive Search and Intelligent Optimization*, volume 45 of *Operations research/Computer Science Interfaces*. Springer Verlag. In press; available online at http://reactive-search.org/thebook.

Beachkofski, B. and Grandhi, R. (2002). Improved distributed hypercube sampling. American Institute of Aeronautics and Astronautics Paper 2002-1274.

Beasley, J. (1990). Or-library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072.

Bentley, J. L. and McIlroy, M. D. (1993). Engineering a sort function. *Software–Practice and Experience*, 23:1249–1265.

Bhalla, A., Lynce, I., de Sousa, J., and Marques-Silva, J. (2003). Heuristic backtracking algorithms for SAT. In *MTV '03: Proc. of the 4th International Workshop on Microprocessor Test and Verification: Common Challenges and Solutions*, pages 69–74.

Birattari, M. (2004). *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective*. PhD thesis, Université Libre de Bruxelles, Brussels, Belgium. Available online at http://iridia.ulb.ac.be/~mbiro/thesis/.

Birattari, M. (2005). *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective*. DISKI 292, Infix/Aka, Berlin, Germany.

Birattari, M., Stützle, T., Paquete, L., and Varrentrapp, K. (2002). A racing algorithm for configuring metaheuristics. In Langdon, W. B., Cantu-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M. A., Schultz, A. C., Miller, J. F., Burke, E., and Jonoska, N., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, pages 11–18. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Bölte, A. and Thonemann, U. W. (1996). Optimizing simulated annealing schedules with genetic programming. *European Journal of Operational Research*, 92(2):402–416.

Borrett, J. E., Tsang, E. P. K., and Walsh, N. R. (1995). Adaptive constraint satisfaction: The quickest first principle. Technical Report CSM-256, Dept. of Computer Science, University of Essex, Colchester, UK.

Boyan, J. A. and Moore, A. W. (2000). Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1:77–112.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

Brewer, E. A. (1994). *Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization*. PhD thesis, Massachusetts Institute of Technology.

Brewer, E. A. (1995). High-level optimization via automated statistical modeling. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 80–91, New York, NY, USA. ACM.

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691.

Burke, E., Kendall, G., Newall, J., Hart, E., Ross, P., and Schulenburg, S. (2003). *Handbook of metaheuristics*, chapter 16, Hyper-heuristics: an emerging direction in modern search technology, pages 457–474. Kluwer Academic Publishers.

Carchrae, T. and Beck, J. C. (2004). Low knowledge algorithm control. In McGuinness, D. and Ferguson, G., editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI'04)*, pages 49–54. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Carchrae, T. and Beck, J. C. (2005). Applying machine learning to low-knowledge control of optimization algorithms. *Computational Intelligence*, 21(4):372–387.

Cavazos, J. and O'Boyle, M. F. P. (2005). Automatic tuning of inlining heuristics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC-05)*, pages 321–353.

Cavazos, J. and O'Boyle, M. F. P. (2006). Method-specific dynamic compilation using logistic regression. In Cook, W. R., editor, *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-06)*, pages 229–240, New York, NY, USA. ACM Press.

Chaloner, K. and Verdinelli, I. (1995). Bayesian experimental design: A review. *Statistical Science*, 10:273–304.

Chen, C., Lin, J., Yücesan, E., and Chick, S. E. (2000). Simulation budget allocation for further enhancing the efficiency of ordinal optimization. *Discrete Event Dynamic Systems*, 10(3):251–270.

Cicirello, V. A. and Smith, S. F. (2004). Heuristic selection for stochastic search optimization: Modeling solution quality by extreme value theory. In *Tenth Internatioal Conference on Principles and Practice of Constraint Programming (CP'04)*, Lecture Notes in Computer Science, pages 197–211. Springer Verlag, Berlin, Germany.

Cicirello, V. A. and Smith, S. F. (2005). The max k-armed bandit: A new model of exploration applied to search heuristic selection. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI'05)*, pages 1355–1361. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Couto, J. (2005). Kernel k-means for categorical data. In *Advances in Intelligent Data Analysis VI (IDA-05)*, volume 3646 of *Lecture Notes in Computer Science*, pages 46–56. Springer Verlag.

Cowling, P., Kendall, G., and Soubeiga, E. (2002). Hyperheuristics: A tool for rapid prototyping in scheduling and optimisation. In *Applications of Evolutionary Computing*, volume 2279 of *Lecture Notes in Computer Science*, pages 1–10. Springer.

Coy, S. P., Golden, B. L., Runger, G. C., and Wasil, E. A. (2001). Using experimental design to find effective parameter settings for heuristics. *Journal of Heuristics*, 7(1):77–97.

Crary, S. B. and Spera, C. (1996). Optimal experimental design for combinatorial problems. *Comput. Econ.*, 9(3):241–255.

Csat, L. and Opper, M. (2003). Sparse gaussian processes: inference, subspace identification and model selection. In Van der Hof, W., editor, *13th IFAC Symposium on System Identifiaction*, pages 1–6, The Netherlands.

Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397.

Dechter, R. and Rish, I. (2003). Mini-buckets: A general scheme for bounded inference. *Journal of the ACM*, 50(2):107–153.

Di Gaspero, L. and Schaerf, A. (2007). Easysyn++: A tool for automatic synthesis of stochastic local search algorithms. In Stützle, T., Birattari, M., and Hoos, H. H., editors, *Proc. International Workshop on Engineering Stochastic Local Search Algorithms (SLS 2007)*, pages 177–181.

Diao, Y., Eskesen, F., Froehlich, S., Hellerstein, J. L., Spainhower, L., and Surendra, M. (2003). Generic online optimization of multiple configuration parameters with application to a database server. In Brunner, M. and Keller, A., editors, *14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM-03)*, volume 2867 of *Lecture Notes in Computer Science*, pages 3–15. Springer Verlag, Berlin, Germany.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

Eén, N. and Biere, A. (2005). Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the 8th Intl. Conf. on Theory and Applications of Satisfiability Testing, LNCS*, volume 3569, pages 61–75.

Eén, N. and Sörensson, N. (2003). An extensible SAT solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, pages 502–518.

Eén, N. and Sörensson, N. (2004). An extensible SAT-solver. In *Proceedings of the 6th Intl. Conf. on Theory and Applications of Satisfiability Testing, LNCS*, volume 2919, pages 502–518.

Eppstein, S. L. and Freuder, E. C. (2001). Collaborative learning for constraint solving. In *Seventh Internatioal Conference on Principles and Practice of Constraint Programming (CP'01)*, Lecture Notes in Computer Science. Springer Verlag, Berlin, Germany.

Epstein, S. L., Freuder, E. C., Wallace, R. J., Morozov, A., and Samuels, B. (2002). The adaptive constraint engine. In Hentenryck, P. V., editor, *Eighth Internatioal Conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 525 – 540. Springer Verlag, Berlin, Germany.

Ertin, E. (2007). Gaussian process models for censored sensor readings. In *Proceedings of the IEEE Statistical Signal Processing Workshop 2007 (SSP'07)*, pages 665–669.

Etzioni, O. and Etzioni, R. (1994). Statistical methods for analyzing speedup learning experiments. *Journal of Machine Learning*, 14(3):333–347.

Fawcett, C., Hoos, H., and Chiarandini, M. (2009). An automatically configured modular algorithm for post enrollment course timetabling. Under review at Principles and Practice of Constraint Programming (CP'09).

Fukunaga, A. S. (2008). Automated discovery of local search heuristics for satisfiability testing. *Evolutionary Computation*, 16(1):31–61.

Gagliolo, M. and Schmidhuber, J. (2006). Dynamic algorithm portfolios. In Amato, C., Bernstein, D., and Zilberstein, S., editors, *Ninth International Symposium on Artificial Intelligence and Mathematics (AI-MATH-06)*.

Gagliolo, M. and Schmidhuber, J. (2007). Learning restart strategies. In Veloso, M. M., editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, volume 1, pages 792–797. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York.

Gebruers, C., Hnich, B., Bridge, D., and Freuder, E. (2005). Using CBR to select solution strategies in constraint programming. In Muñoz-Avila, H. and Ricci, F., editors, *Proceedings of the 6th International Conference on Case Based Reasoning (ICCBR'05)*, volume 3620 of *Lecture Notes in Computer Science*, pages 222–236. Springer Verlag, Berlin, Germany.

Gent, I. P., Hoos, H. H., Prosser, P., and Walsh, T. (1999). Morphing: Combining structure and randomness. In Hendler, J. and Subramanian, D., editors, *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)*, pages 654–660, Orlando, Florida. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Gomes, C. P. and Selman, B. (1997). Problem structure in the presence of perturbations. In Kuipers, B. and Webber, B., editors, *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 221–226. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Gomes, C. P. and Selman, B. (2001). Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62.

Gould, N. I. M., Orban, D., and Toint, P. L. (2004). Cuter (and sifdec), a constrained and unconstrained testing environment, revisited. Technical Report TR/PA/01/04, Cerfacs.

Gratch, J. and Chien, S. A. (1996). Adaptive problem-solving for large-scale scheduling problems: A case study. *Journal of Artificial Intelligence Research*, 4:365–396.

Gratch, J. and Dejong, G. (1992). Composer: A probabilistic solution to the utility problem in speed-up learning. In Rosenbloom, P. and Szolovits, P., editors, *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 235–240. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182.

Hansen, N. (2006). The CMA evolution strategy: a comparing review. In Lozano, J., Larranaga, P., Inza, I., and Bengoetxea, E., editors, *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*, pages 75–102. Springer.

Hansen, N. and Kern, S. (2004). Evaluating the CMA evolution strategy on multimodal test functions. In Yao, X. et al., editors, *Parallel Problem Solving from Nature PPSN VIII*, volume 3242 of *LNCS*, pages 282–291. Springer.

Hansen, N. and Ostermeier, A. (1996). Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In *Proc. of CEC-96*, pages 312–317. Morgan Kaufmann.

Hastie, T., Tibshirani, R., and Friedman, J. H. (2001). *The Elements of Statistical Learning*. Springer Series in Statistics. Springer Verlag.

Hastie, T., Tibshirani, R., and Friedman, J. H. (2009). *The Elements of Statistical Learning*. Springer Series in Statistics. Springer Verlag, 2nd edition.

Hedar, A. (2009). Test functions for unconstrained global optimization. http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO_files/Page364.htm. Version last visited on July 19, 2009.

Hentenryck, P. V. and Michel, L. (2005). *Constraint-Based Local Search*. The MIT Press, Cambridge, MA, USA.

Hoos, H. (1999a). *Stochastic Local Search – Methods, Models, Applications*. infix-Verlag, Sankt Augustin, Germany.

Hoos, H. H. (1999b). On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)*, pages 661–666. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Hoos, H. H. (2002a). An adaptive noise mechanism for WalkSAT. In Dechter, R., Kearns, M., and Sutton, R., editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI'02)*, pages 655–660. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Hoos, H. H. (2002b). A mixture-model for the behaviour of SLS algorithms for SAT. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02)*, pages 661–667, Edmonton, Alberta, Canada.

Hoos, H. H. (2008). Computer-aided design of high-performance algorithms. Technical Report TR-2008-16, University of British Columbia, Department of Computer Science.

Hoos, H. H. and Stützle, T. (2000). Local search algorithms for SAT: An empirical evaluation. *Journal of Automated Reasoning*, 24(4):421–481.

Hoos, H. H. and Stützle, T. (2005). *Stochastic Local Search – Foundations & Applications*. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Horvitz, E., Ruan, Y., Gomes, C. P., Kautz, H., Selman, B., and Chickering, D. M. (2001). A Bayesian approach to tackling hard computational problems. In Breese, J. S. and Koller, D.,

editors, *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence (UAI'01)*, pages 235–244. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Huang, D., Allen, T. T., Notz, W. I., and Zeng, N. (2006). Global optimization of stochastic black-box systems via sequential kriging meta-models. *Journal of Global Optimization*, 34(3):441–466.

Hutter, F. (2004). Stochastic local search for solving the most probable explanation problem in Bayesian networks. Master's thesis, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany.

Hutter, F. (2007). On the potential of automatic algorithm configuration. In *SLS-DS2007: Doctoral Symposium on Engineering Stochastic Local Search Algorithms*, pages 36–40. Technical report TR/IRIDIA/2007-014, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium.

Hutter, F., Babić, D., Hoos, H. H., and Hu, A. J. (2007a). Boosting Verification by Automatic Tuning of Decision Procedures. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD'07)*, pages 27–34, Washington, DC, USA. IEEE Computer Society.

Hutter, F., Bartz-Beielstein, T., Hoos, H. H., Leyton-Brown, K., and Murphy, K. P. (2009a). Sequential model-based parameter optimisation: an experimental investigation of automated and interactive approaches. In *Empirical Methods for the Analysis of Optimization Algorithms*. Springer Verlag. To appear.

Hutter, F. and Hamadi, Y. (2005). Parameter adjustment based on performance prediction: towards an instance-aware problem solver. Technical Report MSR-TR-2005-125, Microsoft Research, Cambridge, UK.

Hutter, F., Hamadi, Y., Hoos, H. H., and Leyton-Brown, K. (2006). Performance prediction and automated tuning of randomized and parametric algorithms. In Benhamou, F., editor, *Twelfth Internatioal Conference on Principles and Practice of Constraint Programming (CP'06)*, volume 4204 of *Lecture Notes in Computer Science*, pages 213–228. Springer Verlag, Berlin, Germany.

Hutter, F., Hoos, H., Leyton-Brown, K., and Stützle, T. (2009b). PARAMILS: An automatic algorithm configuration framework. Technical Report TR-2009-01, University of British Columbia, Department of Computer Science.

Hutter, F., Hoos, H., Leyton-Brown, K., and Stützle, T. (2009c). ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*. Accepted for publication.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2009d). Tradeoffs in the empirical evaluation of competing algorithm designs. Technical report, University of British Columbia, Department of Computer Science.

Hutter, F., Hoos, H. H., Leyton-Brown, K., and Murphy, K. P. (2009e). An experimental investigation of model-based parameter optimisation: SPO and beyond. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2009)*, pages 271–278.

Hutter, F., Hoos, H. H., and Stützle, T. (2005). Efficient stochastic local search for MPE solving. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 169–174.

Hutter, F., Hoos, H. H., and Stützle, T. (2007b). Automatic algorithm configuration based on local search. In Howe, A. and Holte, R. C., editors, *Proceedings of the Twentysecond National Conference on Artificial Intelligence (AAAI'07)*, pages 1152–1157. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Hutter, F., Tompkins, D. A. D., and Hoos, H. H. (2002). Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In Hentenryck, P. V., editor, *Eighth Internatioal Conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 233–248. Springer Verlag, Berlin, Germany.

Jackson, D. (2000). Automating first-order relational logic. *SIGSOFT Softw. Eng. Notes*, 25(6):130–139.

Johnson, D. S. (2002). A theoretician's guide to the experimental analysis of algorithms. In Goldwasser, M. H., Johnson, D. S., and McGeoch, C. C., editors, *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, pages 215–250. American Mathematical Society, Providence, RI, USA.

Jones, D. R., Perttunen, C. D., and Stuckman, B. E. (1993). Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181.

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

Kask, K. and Dechter, R. (1999). Stochastic local search for Bayesian networks. In *International Workshop on Artificial Intelligence and Statistics (AISTATS-99)*.

Kautz, H., Horvitz, E., Ruan, Y., Gomes, C. P., and Selman, B. (2002). Dynamic restart policies. In Dechter, R., Kearns, M., and Sutton, R., editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI'02)*, pages 674–681. AAAI Press / The MIT Press, Menlo Park, CA, USA.

KhudaBukhsh, A., Xu, L., Hoos, H. H., and Leyton-Brown, K. (2009). SATenstein: Automatically building local search sat solvers from components. In *Proceedings of the Twentyfirst International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 517–524.

Kilby, P., Slaney, J., Thiebaux, S., and Walsh, T. (2006). Estimating search tree size. In *Proceedings of the Twentyfirst National Conference on Artificial Intelligence (AAAI'06)*, pages 1014–1019. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Knuth, D. (1975). Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136.

Kohavi, R. and John, G. H. (1995). Automatic parameter selection by minimizing estimated error. In *Proceedings of the Twelfth International Conference on Machine Learning (ICML-95)*.

Lagoudakis, M. G. and Littman, M. L. (2000). Algorithm selection using reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning (ICML-00)*, pages 511–518.

Lagoudakis, M. G. and Littman, M. L. (2001). Learning to select branching rules in the DPLL procedure for satisfiability. In *Electronic Notes in Discrete Mathematics (ENDM)*.

Lasarczyk, C. W. G. (2007). *Genetische Programmierung einer algorithmischen Chemie*. PhD thesis, Technische Universität Dortmund.

Leyton-Brown, K. (2003). *Resource Allocation in Competitive Multiagent Systems*. PhD thesis, Stanford University.

Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., and Shoham, Y. (2003a). Boosting as a metaphor for algorithm design. In *Ninth Internatioal Conference on Principles and Practice of Constraint Programming (CP'03)*, pages 899–903.

Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., and Shoham, Y. (2003b). A portfolio approach to algorithm selection. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 1542–1543.

Leyton-Brown, K., Nudelman, E., and Shoham, Y. (2002). Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In Hentenryck, P. V., editor, *Eighth Internatioal Conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 556–572. Springer Verlag, Berlin, Germany.

Leyton-Brown, K., Nudelman, E., and Shoham, Y. (2009). Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4):1–52.

Leyton-Brown, K., Pearson, M., and Shoham, Y. (2000). Towards a universal test suite for combinatorial auction algorithms. In Jhingran, A., Mason, J. M., and Tygar, D., editors, *EC '00: Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76, New York, NY, USA. ACM.

Li, C. and Huang, W. (2005). Diversification and determinism in local search for satisfiability. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, pages 158–172.

Li, X., Garzarán, M. J., and Padua, D. (2005). Optimizing sorting with genetic algorithms. In *Proc. International Symposium on Code Generation and Optimization*, pages 99–110. IEEE Computer Society.

Lobjois, L. and Lemaître, M. (1998). Branch and bound algorithm selection by performance prediction. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Lophaven, S. N., Nielsen, H. B., and Sondergaard, J. (2002). Aspects of the Matlab toolbox DACE. Technical Report IMM-REP-2002-13, Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Kongens Lyngby, Denmark.

Lourenço, H. R., Martin, O., and Stützle, T. (2002). Iterated local search. In Glover, F. and Kochenberger, G., editors, *Handbook of Metaheuristics*, pages 321–353. Kluwer Academic Publishers, Norwell, MA, USA.

Lu, F., Wang, L.-C., Cheng, K.-T. T., Moondanos, J., and Hanna, Z. (2003). A signal correlation guided ATPG solver and its applications for solving difficult industrial cases. In *DAC '03: Proc. of the 40th conference on Design automation*, pages 436–441. ACM Press.

Maron, O. and Moore, A. (1994). Hoeffding races: Accelerating model selection search for classification and function approximation. In Cowan, J. D., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems 7 (NIPS-94)*, volume 6, pages 59–66. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Maturana, J., Fialho, A., Saubion, F., Schoenauer, M., and Sebag, M. (2009). Extreme compass and dynamic multi-armed bandits for adaptive operator selection. In *CEC'09: Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 365–372. IEEE Press.

McAllester, D., Selman, B., and Kautz, H. (1997). Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 321–326. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Meinshausen, N. (2006). Quantile regression forests. *Journal of Machine Learning Research*, 7:983–999.

Mengshoel, O. J. (2008). Understanding the role of noise in stochastic local search: Analysis and experiments. *Artificial Intelligence*, 172(8-9):955–990.

Minton, S. (1993). An analytic learning system for specializing heuristics. In Bajcsy, R., editor, *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 922–929. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Minton, S. (1996). Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1):1–40.

Minton, S., Johnston, M. D., Philips, A. B., and Laird, P. (1992). Minimizing conflicts: A heuristic repair method for constraint-satisfaction and scheduling problems. *Artificial Intelligence*, 58(1):161–205.

Mockus, J., Tiesis, V., and Zilinskas, A. (1978). The application of bayesian methods for seeking the extremum. *Towards Global Optimisation*, 2:117–129. North Holland, Amsterdam.

Monette, J., Deville, Y., and Van Hentenryck, P. (2009). Aeon: Synthesizing scheduling algorithms from high-level models. In *Proc. 11th INFORMS Computing Society Conference (to appear)*.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: engineering an efficient SAT solver. In *DAC '01: Proc. of the 38th conference on Design automation*, pages 530–535. ACM Press.

Muja, M. and Lowe, D. G. (2009). Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Applications (VISAPP-09)*.

Nelder, J. A. and Mead, R. (1965). A simplex method for function minimization. *The Computer Journal*, 7(4):308–313.

Nelson, G. (1979). *Techniques for program verification*. PhD thesis, Stanford University.

Nudelman, E., Leyton-Brown, K., Hoos, H. H., Devkar, A., and Shoham, Y. (2004). Understanding random SAT: Beyond the clauses-to-variables ratio. In *Tenth Internatioal Conference on Principles and Practice of Constraint Programming (CP'04)*, Lecture Notes in Computer Science. Springer Verlag, Berlin, Germany.

Oltean, M. (2005). Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410.

Patterson, D. J. and Kautz, H. (2001). Auto-WalkSAT: a self-tuning implementation of WalkSAT. In *Electronic Notes in Discrete Mathematics (ENDM), 9*.

Poli, R., Langdon, W. B., and McPhee, N. F. (2008). *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

Prasad, M. R., Biere, A., and Gupta, A. (2005). A survey of recent advances in SAT-based formal verification. *STTT: International Journal on Software Tools for Technology Transfer*, 7(2):156–173.

Quinonero-Candela, J., Rasmussen, C. E., and Williams, C. K. (2007). Approximation methods for gaussian process regression. In *Large-Scale Kernel Machines*, Neural Information Processing, pages 203–223. MIT Press, Cambridge, MA, USA.

Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning*. The MIT Press, Cambridge, MA, USA.

Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, 15:65–118.

Ridge, E. and Kudenko, D. (2006). Sequential experiment designs for screening and tuning parameters of stochastic heuristics. In Paquete, L., Chiarandini, M., and Basso, D., editors, *Workshop on Empirical Methods for the Analysis of Algorithms at the Ninth International Conference on Parallel Problem Solving from Nature (PPSN)*, pages 27–34.

Ridge, E. and Kudenko, D. (2007). Tuning the performance of the mmas heuristic. In Stützle, T., Birattari, M., and Hoos, H. H., editors, *Proc. International Workshop on Engineering Stochastic Local Search Algorithms (SLS 2007)*, pages 46–60.

Rothkopf, M., Pekec, A., and Harstad, R. (1998). Computationally manageable combinational auctions. *Management Science*, 44(8):1131–1147.

Sacks, J., Welch, W. J., Welch, T. J., and Wynn, H. P. (1989). Design and analysis of computer experiments. *Statistical Science*, 4(4):409–423.

Samulowitz, H. and Memisevic, R. (2007). Learning to solve QBF. In *Proceedings of the Twentysecond National Conference on Artificial Intelligence (AAAI'07)*, pages 255–260.

Sanders, P. and Schultes, D. (2007). Engineering fast route planning algorithms. In *Proc. 6th International Workshop on Experimental Algorithms (WEA-2007)*, volume 4525 of *Lecture Notes in Computer Science*, pages 23–36. Springer.

Santner, T. J., Williams, B. J., and Notz, W. I. (2003). *The Design and Analysis of Computer Experiments*. Springer Verlag, New York.

Saxena, A. (2008). Benchmark Instances in .mps format. http://www.andrew.cmu.edu/user/anureets/mpsInstances.htm. Version last visited on April 29, 2009.

Schmee, J. and Hahn, G. J. (1979). A simple method for regression analysis with censored data. *Technometrics*, 21(4):417–432.

Schonlau, M., Welch, W. J., and Jones, D. R. (1998). Global versus local search in constrained optimization of computer models. In Flournoy, N., Rosenberger, W., and Wong, W., editors, *New Developments and Applications in Experimental Design*, volume 34, pages 11–25. Institute of Mathematical Statistics, Hayward, California.

Schuurmans, D. and Southey, F. (2001). Local search characteristics of incomplete SAT procedures. *Artificial Intelligence*, 132(2):121–150.

Segal, M. R. (1988). Regression trees for censored data. *Biometrics*, 44(1):35–47.

Segre, A., Elkan, C., and Russell, A. (1991). A critical look at experimental evaluations of EBL. *Journal of Machine Learning*, 6(2):183–195.

Selman, B., Kautz, H., and Cohen, B. (1996). Local search strategies for satisfiability testing. In Johnson, D. S. and Trick, M. A., editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*.

Selman, B., Levesque, H. J., and Mitchell, D. (1992). A new method for solving hard satisfiability problems. In Rosenbloom, P. and Szolovits, P., editors, *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 440–446. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Seshia, S. A. (2005). Adaptive eager boolean encoding for arithmetic reasoning in verification. Technical Report CMU-CS-05-134, School of Computer Science, Carnegie Mellon University.

Shacham, O. and Zarpas, E. (2003). Tuning the VSIDS Decision Heuristic for Bounded Model Checking. In *Fourth International Workshop on Microprocessor Test and Verification, Common Challenges and Solutions (MTV 2003), May 29-30, 2003, Hyatt Town Lake Hotel, Austin, Texas, USA*, pages 75–79. IEEE Computer Society.

Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press, New York, NY, USA.

Shtrichman, O. (2000). Tuning SAT checkers for bounded model checking. In *CAV '00: Proc. of the 12th International Conference on Computer Aided Verification*, volume 1855 of *LNCS*, pages 480–494. Springer.

Silva, J. P. M. (1999). The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *EPIA '99: Proc. of the 9th Portuguese Conference on Artificial Intelligence*, volume 1695 of *LNCS*, pages 62–74. Springer.

Simon, L. and Chatalic, P. (2001). SATEx: a web-based framework for SAT experimentation. In Kautz, H. and Selman, B., editors, *Proceedings of the Fourth International Conference on Theory and Applications of Satisfiability Testing (SAT'01)*.

Snelson, E. and Ghahramani, Z. (2006). Sparse gaussian processes using pseudo-inputs. In Weiss, Y., Schölkopf, B., and Platt, J., editors, *Advances in Neural Information Processing Systems 18 (NIPS-05)*, Cambridge, MA, USA. MIT Press.

Spall, J. C. (1999). Stochastic optimization: Stochastic approximation and simulated annealing. *Encyclopedia of Electrical and Electronics Engineering*, 20:529–542.

Spall, J. C. (2003). *Introduction to Stochastic Search and Optimization*. John Wiley & Sons, Inc., New York, NY, USA.

Srivastava, B. and Mediratta, A. (2005). Domain-dependent parameter selection of search-based algorithms compatible with user performance criteria. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI'05)*, pages 1386–1391. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Stillger, M. and Spiliopoulou, M. (1996). Genetic programming in database query optimization. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 388–393, Cambridge, MA, USA. MIT Press.

Streeter, M. J. and Smith, S. F. (2006a). An asymptotically optimal algorithm for the max k-armed bandit problem. In *Proceedings of the Twentyfirst National Conference on Artificial Intelligence (AAAI'06)*, pages 135–142. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Streeter, M. J. and Smith, S. F. (2006b). A simple distribution-free approach to the max k-armed bandit problem. In Benhamou, F., editor, *Twelfth Internatioal Conference on Principles and Practice of Constraint Programming (CP'06)*, Lecture Notes in Computer Science, pages 560–574. Springer Verlag, Berlin, Germany.

Sversky, K. and Murphy, K. (2009). How to train restricted boltzmann machines and deep belief networks. Technical report, University of British Columbia, Department of Computer Science. In preparation.

Terashima-Marín, H., Ross, P., and Valenzuela-Réndon, M. (1999). Evolution of constraint satisfaction strategies in examination timetabling. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, pages 635–642. Morgan Kaufmann.

Thachuk, C., Shmygelska, A., and Hoos, H. H. (2007). A replica exchange monte carlo algorithm for protein folding in the hp model. *BMC Bioinformatics*, 8:342–342.

Tolson, B. A. and Shoemaker, C. A. (2007). Dynamically dimensioned search algorithm for computationally efficient watershed model calibration. *Water Resources Research*, 43.

Tompkins, D. A. D. and Hoos, H. H. (2004). UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT & MAX-SAT. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, volume 3542, pages 306–320. Springer Verlag, Berlin, Germany.

Van Hentenryck, P. and Michel, L. D. (2007). Synthesis of constraint-based local search algorithms from high-level models. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 273–278.

Westfold, S. J. and Smith, D. R. (2001). Synthesis of efficient constraint-satisfaction programs. *Knowl. Eng. Rev.*, 16(1):69–84.

Whaley, R. C. (2004). *Automated Empirical Optimization of High Performance Floating Point Kernels*. PhD thesis, The Florida State University of Arts & Sciences, Florida, USA.

Whaley, R. C., Petitet, A., and Dongarra, J. J. (2001). Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35.

Williams, B. J., Santner, T. J., and Notz, W. I. (2000). Sequential design of computer experiments to minimize integrated response functions. *Statistica Sinica*, 10:1133–1152.

Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

Xu, L., Hoos, H. H., and Leyton-Brown, K. (2007a). Hierarchical hardness models for SAT. In Bessiere, C., editor, *Thirteenth Internatioal Conference on Principles and Practice of Constraint Programming (CP'07)*, Lecture Notes in Computer Science, pages 696–711. Springer Verlag, Berlin, Germany.

Xu, L., Hutter, F., Hoos, H., and Leyton-Brown, K. (2007b). Satzilla-07: The design and analysis of an algorithm portfolio for SAT. In Bessiere, C., editor, *Thirteenth Internatioal Conference on Principles and Practice of Constraint Programming (CP'07)*, Lecture Notes in Computer Science, pages 712–727. Springer Verlag, Berlin, Germany.

Xu, L., Hutter, F., Hoos, H., and Leyton-Brown, K. (2009). SATzilla2009: an automatic algorithm portfolio for sat. Solver description, SAT competition 2009.

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.

Zarpas, E. (2005). Benchmarking SAT Solvers for Bounded Model Checking. In Bacchus, F. and Walsh, T., editors, *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 340–354. Springer Verlag.