Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Michael Rudolf**

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: **Parameter tuning for numerical optimization algorithms**

Guidelines:

1. Study the principles and methods used for offline tuning of parameters of numerical optimization algorithms, with emphasis on the following algorithms: irace, ParamILS, SPOT, and SMAC.
2. Compare the chosen tuning algorithms on a set of benchmark functions, e.g. found in COCO framework, as well as on some mixed continuous-discrete problems.
3. Use the chosen tuning procedure to optimize parameters of at least 2 different optimization algorithms for numerical problems. Compare their results with default and tuned parameter values.
4. Assess the contributions of parameter tuning.

Bibliography/Sources:

[1] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stutzle. ParamILS: An Automatic Algorithm Configuration Framework. Journal of Artificial Research 2009, volume 36, pages 267-306.
[2] Manuel Lopez-Ibanez, Jeremie Dubois-Lacoste, Thomas Stützle, Mauro Birattari: The irace Package: Iterated Racing for Automatic Algorithm Configuration. IRIDIA Technical Report TR/IRIDIA/2011-004, 2011.
[3] Thomas Bartz-Beielstein: SPOT: An R Package For Automatic and Interactive Tuning of Optimization Algorithms by Sequential Parameter Optimization. 2010, arXiv:1006.4645v1
[4] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In proceedings of LION-5, 2011
[5] COCO: Comparing Continuous Optimizers framework, webpage: http://coco.gforge.inria.fr/

Diploma Thesis Supervisor: Ing. Petr Pošík, Ph.D.

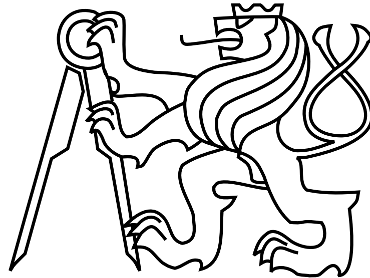
Valid until the end of the summer semester of academic year 2016/2017

prof. Ing. Filip Železný, Ph.D.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 7, 2016

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science



Master's Thesis

Parameter tuning for numerical optimization algorithms

Bc. Michael Rudolf

Supervisor: Ing. Petr Pošík, Ph.D.

Study Programme: Open Informatics

Field of Study: Artificial Intelligence

May 26, 2017

Aknowledgements

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated. Also, access to the CERIT-SC computing and storage facilities provided by the CERIT-SC Center, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CERIT Scientific Cloud LM2015085), is greatly appreciated. I would like to thank my family and friends for their support and my supervisor Ing. Petr Pošík, Ph.D., for his help, patience and much advice he provided me with during the work on this thesis.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

V Praze dne 15. 9. 2016 ....

Abstract

As there is an increasing trend in generation of new optimization problems about which often imperfect or incomplete information is known, researchers came up with many optimizing algorithms that can tackle large variety of specific problem scenarios. Because *No Free Lunch Theorem* states there is no algorithm that could outperform all other algorithms on all problems, the optimizing algorithms are parametrized to somewhat adapt to different scenarios and perform reasonably well on wider range of problems. Search for ideal algorithm setting can be viewed as another optimization problem called meta-optimization.

This thesis compares several meta-optimization techniques on public benchmark and examines the possibilities of single algorithm setting that would generalize each of the tuned optimizers to perform well on whole class of problems, rather than solving instances of just one problem. Optimization results of several variants of tuned optimizers are presented. They exhibit a range from none to substantial performance increase.

Abstrakt

Vzhledem k narůstajícímu trendu ve vytváření a objevování nových optimalizačních problémů, o kterých často nemáme dokonalou či úplnou znalost, přišli výzkumníci s mnoha optimalizačními algoritmy, které mohou čelit široké škále těchto problémů. Protože *No Free Lunch Theorem* říká, že neexistuje algoritmus, který by předčil všechny ostatní pro všechny problémy, bývají algoritmy parametrizované tak, aby se nějakým způsobem přizpůsobily různým scénářům a měly rozumnou výkonnost pro větší škálu problémů. Na hledání ideálního nastavení algoritmu můžeme nahlížet jako na další optimalizační problém kterému se říká metaoptimalizace.

Tato práce porovnává několik metaoptimalizačních technik na veřejně dostupném výkonnostním testu a zkoumá možnosti jediného nastavení algoritmů, které by zobecňovalo každý z laděných optimalizačních nástrojů tak, aby měl dobrou výkonnost na celých třídách problémů, spíše než na instancích pouze jednoho problému. Jsou zde prezentovány výsledky optimalizace několika variant laděných optimalizátorů. Vykazují rozpětí od žádného k významnému zvýšení výkonnosti.

Contents

Introduction **1**

1 Parameter tuning **3**
1.1 Problem Outline . . . 3
1.2 Motivation . . . 4
1.3 Goals and Hypotheses . . . 5

2 State of art in automatic parameter tuning **7**
2.1 Meta-evolution . . . 7
2.2 Sequential Parameter Optimization (SPO) . . . 7
2.3 Estimation of Distribution (EDA) . . . 8
2.4 Racing . . . 8
2.5 Sharpening . . . 8
2.6 Local Search . . . 8
2.7 Adaptive Capping . . . 9

3 Examined tuning methods **11**
3.1 Overview . . . 11
3.2 Algorithms . . . 12
3.2.1 ParamILS . . . 12
3.2.2 SMAC . . . 13
3.2.3 Irace . . . 13
3.2.4 SPOT . . . 14
3.2.5 Spearmint . . . 14
3.3 Performance metrics . . . 14

4 COCO benchmarking Framework **15**
4.1 Overview . . . 15
4.2 Purpose . . . 15
4.3 Testing functions and their benefits . . . 16
4.4 Benchmarking results and their meaning . . . 17

5 Conducted Experiments **21**
5.1 Experiment analysis . . . 21
5.2 Realization . . . 23
5.2.1 Used resources . . . 23

5.2.2 Optimization experiment structure 23
5.2.3 Optimization experiment execution 23
5.2.4 Parameter tuning structure 25
5.2.5 Parameter tuning execution 25

6 Results 29
6.1 Optimization experiment results 29
6.1.1 Experiments with negative results 29
6.1.2 Expected run length comparison 31
6.1.3 Algorithm performance comparison 31
6.2 Parameter tuning experiment results 34
6.2.1 GA unimodal tests 34
6.2.2 GA multimodal tests 36
6.2.3 DE unimodal tests 37
6.2.4 DE multimodal tests 39

7 Discussion 41
7.1 Optimization benchmark experiments 41
7.2 Parameter tuning experiments 42
7.2.1 GA testing 42
7.2.2 DE testing 42
7.3 Improvements 43

8 Conclusion 45

Bibliography 47

A Technical details about examined frameworks 51
A.1 Implementation Language 51
A.2 Algorithm usage 52
A.3 Parameter handling 53

B List of used abbreviations 57

C CD Content 59

List of Figures

1.1 Parameter tuning concept . . . 3

4.1 Scatter plot comparing 2 algorithms run lengths . . . 17
4.2 Run length distribution of two algorithms . . . 18
4.3 Expected/observed run length loss ratio . . . 18
4.4 Run length distribution of many algorithms . . . 19

5.1 Area under curve performance metric . . . 28

6.1 Expected runtime per dimension comparison . . . 30
6.2 Parameter tuner optimizing abilities comparison in 5D . . . 32
6.3 Parameter tuner optimizing abilities comparison in 20D . . . 33
6.4 GA unimodal training performance . . . 34
6.5 GA unimodal testing performance . . . 35
6.6 GA unimodal different class performance . . . 35
6.7 GA multimodal training performance . . . 36
6.8 GA multimodal testing performance . . . 36
6.9 GA multimodal different class performance . . . 37
6.10 DE unimodal training performance . . . 37
6.11 DE unimodal testing performance . . . 38
6.12 DE unimodal different class performance . . . 38
6.13 DE multimodal training performance . . . 39
6.14 DE multimodal testing performance . . . 39
6.15 DE multimodal different class performance . . . 40

Introduction

Optimization in strict mathematical view is finding extrema of some function but can be also viewed as a search with goal of finding the best (optimal) solution to some problem. Many real life optimization problems tend to have large search spaces, and it turns out, for increasing number of optimized variables there is exponential increase in problem complexity. This is called the curse of dimensionality [21] and it is being dealt with by algorithms that expect arbitrary number of input parameters for each optimization problem.

Many optimizers can be adjusted to increase their performance on different problems with different dimensionality by choosing the right input parameters. These parameters change the behavior of the optimizer and can have positive effect on the overall solution quality if chosen properly. Choosing algorithm parameters by hand can be inefficient and difficult as we often do not know the full extend of parameter relations. Looking for an efficient parameter initialization of the optimizing algorithm can also be viewed as an optimizing problem, often called meta-optimization.

Meta-optimization which is also known as super-optimization, parameter tuning, automated parameter calibration or even hyper-heuristics is therefore a research field of searching the right behavioral parameters for some underlying optimizer. One can easily deduce that meta-optimization is computationally expensive, as it requires at least several computations of the underlying optimization problem in order to determine performance of each parameter instantiation. Therefore, it is beneficial to have as few parameters as possible and a good performance measure to distinguish only the promising sets of parameters. Also, different approaches to meta-optimization work with different success depending on whether the tuned parameters are real-valued or discrete (with finite number of choices).

Meta-optimization is no new concept as it was already used in the late 1970s by Mercer and Sampson [38] for optimizing evolutionary algorithm. The used method is known as *Meta-evolutionary algorithm*, basically any evolutionary algorithm can be used as meta-evolutionary as long as it can use numeric vectors representing parameter instances as its individuals.

Besides meta-evolution there are several main courses which to take when implementing parameter tuning. There are *Estimation of Distribution* (EDA) based algorithms such as REVAC [40, 39], *Sequential Parameter Optimization* methods (SPO) which usually construct models of the parameter space, and try to search for good instantiations, sampling the underlying models [33, 19, 17]. There are also various techniques that can be added to already existing algorithms such as *racing* [37, 36] that aims to decrease the number of optimization evaluations by comparing parameter efficiency on as few instances as possible, or *sharpening* [20, 44] that ensures the promising configurations are evaluated more thoroughly than those that perform poorly.

Interesting overview of parameter tuning was further provided in the work of Smit and Eiben [44], Hoos and Holger [29] and Dobslaw [25].

This thesis is structured in the following manner: The Chapter 1 describes the topic of parameter tuning in general. It states the goals and hypotheses of this thesis and explains the motivation driving this area of research. Chapter 2 presents a brief overview of related work in the area as well as the current state-of-art methods. Chapter 3 provides detailed examination of used methods. Their usage and implementation is explained and commented. Chapter 4 describes the used benchmarking framework [3]. Its goals, examples of its results and their interpretation. Chapter 5 is focused entirely on conducted experiments. The requirements for the optimization scenarios are outlined as well as the details of realization based on these requirements. Chapter 6 presents the results of parameter tuning on benchmark [3] functions as well as the results of 2 optimizing algorithms with tuned parameters on functions from 2 classes of problems. Chapter 7 discusses the achieved results relative to the assigned hypotheses and gives an outlook on possible improvements for the future. Chapter 8 concludes the work and its relevance.

Chapter 1

Parameter tuning

1.1 Problem Outline

Parameter tuning is very similar to classical black-box function optimization. The precise gradient of the optimization problem is unknown and the fitness function only gives some performance measure of a candidate solution. We may know some constraints on each dimension parameter, but we do not know all the dependencies the particular dimensions pose on each other and the resulting value space. The basic view of the problem has the following structure [41]:

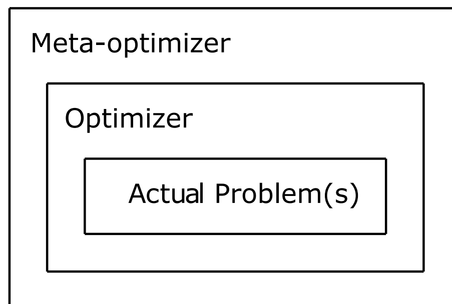


Figure 1.1: Parameter tuning concept. Overlaying black-box optimizer is used to set up input parameters for another optimization method.

Meta-optimizer is searching for the optimal (or at least somewhat well performing) set of behavioral parameters for the chosen optimizing method. The parameter tuning method has no specific knowledge about the underlying problem. The only important information are domains of the desired parameters and stopping criterion.

The meta-optimization is computationally demanding as it has to run the underlying optimizer for each candidate configuration. Due to the high problem complexity and high computation times, meta-optimizers are often limited to a certain number of the underlying algorithm runs, time limit or maximum number of candidate configurations.

1.2 Motivation

Despite the fact that there are several (if not many) ways to let automatically calibrate almost any optimizing algorithm because meta-optimizers can usually deal with parameters of both continuous and discrete domains, vast majority of researchers still rely on expertly chosen input parameters for their methods. The importance of using one or a few unified meta-optimizing frameworks can be explained in 3 main reasons [34]:

- **Algorithm comparison.** When evaluating several optimizing methods the important question is whether one algorithm or heuristic outperforms the other because it is fundamentally superior or because the authors were more successful in optimizing the input parameters. The automatic parameter tuning would eliminate this kind of uncertainty as well as provide common ground for algorithm evaluation thus generating more meaningful comparative studies.
- **Practical usage of optimizers.** The overall performance or ability to find feasible solution on difficult and computationally hard problems greatly depends on using suitable parameter configurations. End users of optimizing frameworks often have little or no knowledge about the impact of an algorithm's parameter settings on its performance thus using the default settings. Even if the used method has been carefully optimized on a standard benchmark set, default configuration may not perform well on the real-life problem instances. Automatic parameter tuning methods can be used to improve performance without any expert knowledge or experience requirements in a principled and convenient way.
- **Algorithm development.** When using automatic parameter configuration on some optimizing method, the researchers can focus more on the general idea and the fundamental principles that should lead the algorithm progress towards optimal solutions rather than spending large fraction of the development time searching for the best settings. Apart from parameter configuration time savings, the automated tuning can potentially achieve better results than manual, ad-hoc methods.

1.3 Goals and Hypotheses

This thesis aims to compare the optimizing skills of several current parameter tuning algorithms and choose some to check the overall performance enhancing capabilities for traditional optimizing algorithms. The work does not aim to find the best meta-optimizer of them all but rather to show that automatic parameter tuning is relatively easy to incorporate and it has advantages that could provide more meaningful comparative studies in the future.

Another question that this work tries to challenge is whether the optimization algorithms can be trained to perform well on a whole class of optimization instances with just one or only a few parameter configurations. Essentially whether it is possible to generalize said optimizer with new default settings, specific for the problem class, thus saving on parameter tuning over each scenario in that class.

In order to accomplish the above objectives it is necessary to provide the overview of currently used methods. From these methods it is desirable to choose some representatives, understand their implementation and usage and find a way to compare them in common fashion.

The desired outcome is a meaningful representation of computed data that would clearly support or disprove the goals and hypotheses in mind, preferably in a form of graphs and data tables. Based on the data, decisions and answers upon the individual questions are presented.

Chapter 2

State of art in automatic parameter tuning

This chapter should serve as a brief summary listing the main courses of parameter tuning. Current methods as well as methods used in the past are presented to lay down some basics of the researched field.

2.1 Meta-evolution

Used almost 40 years ago for the first time by Mercer and Sampson [38], meta-evolutionary algorithm is just another evolutionary algorithm on top of the baseline evolutionary algorithm. The only condition the overlaying algorithm must satisfy is that the population individuals are vectors of parameter values later used for the main optimization. Each population based algorithm can be used as meta-optimizer, as long as it has the right individual representation. The recent meta-evolutionary algorithms use Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [26] as it has good reputation as a numeric optimizer.

2.2 Sequential Parameter Optimization (SPO)

SPO as introduced by Bartz-Beielstein et al. [18] shows some similarities with the meta-evolutionary algorithm as it is also population-based. In each iteration of the algorithm a population of configurations is evaluated and based on the evaluation a (regression) model representing the parameter utility dependency is created. The current model is then used to test and filter new configurations, from which only the promising are added to the population. There is a wide range of SPO methods which differ mainly in the used models, from the traditional regression models used in the past to the stochastic models such as Gaussian processes used today [19, 33].

2.3 Estimation of Distribution (EDA)

Parameter Relevance Estimation and Value Calibration (REVAC) introduced by Nannen and Eiben [39, 40] is based on the same general idea as the EDA algorithms [42] because it estimates the distribution of promising values over each parameter. The result of this method is a set of distributions, representing the utility landscape. The candidate configurations are then chosen according to each parameter distribution. Such simple model does not count with the mutual parameter affection, but gives at least some insight to the relevance and sensitivity of the respective parameters.

2.4 Racing

The main idea of racing is to focus only on the well preforming configurations and discarding those that do not perform well enough. This is possible through variable number of algorithm runs. The number of tests for each configuration greatly depends on the performance of the underlying algorithm. The more promising configuration is found, the more tests it has been run against. Usually new configuration is run against small subset of the testing (or training) instances and it gets chosen for the subsequent runs if it is not substantially worse than the best yet found solution. The discarding of the nonperforming configurations saves substantial part of the computational resources and enables the algorithm to further explore the search space.

In order to effectively compare the candidate configuration statistical tests such as Friedman's test or pairwise t-test [36] are commonly used.

2.5 Sharpening

Sharpening [44] means that the promising parameter configurations are tested more thoroughly than the ones that are not as perspective. It basically starts with small number of tests, and when certain threshold is met, the number of tests doubles. It can be viewed as an opposing force to the racing process as it tries to incorporate more tests for the promising configurations where racing tries to do as few tests as possible.

When combined with racing the resulting algorithm starts with only few tests for each individual in population and the best performing configurations are further sharpened to be comparable by means of racing.

2.6 Local Search

Local search methods simply sample the search space of all parameter settings and explore the local neighborhoods (configurations differing in one parameter value) of promising solutions with decreasing sampling range to ensure convergence to local optimum. Methods of this family differ mostly in computing the *sampling range decreasing factor* which is affected by the current number of non-improving solutions and the problem dimensionality (in this case number of optimized parameters).

2.7 Adaptive Capping

Adaptive capping [34] is another method used to cut off the unpromising configurations and thus providing more time to focus on other regions of the search space which leads to searching through more configurations and possible quality improvement in comparison to the basic variant of the search. It is primarily designed for iterative search methods and the authors consider two of its variants [34].

For the first approach, called trajectory preserving, each new configuration setting has its computing budget set by the previous candidate (best found) solution. So if after same number of evaluations the candidate configuration is not equally good or better than the previous one (in terms of dominance), the search continues with another parameter setting. Each iteration thus starts slowly by initial or perturbed solution configuration and then, as the algorithm progresses, it gets tougher and tougher to top the better solutions. This approach preserves the trajectory of the search as it visits same configurations as would the basic variant only with significant speedup.

The second approach the author call aggressive adaptive capping because the computational limit depends on overall best found solution performance times weight α which is being adaptively adjusted throughout the algorithm run.

Chapter 3

Examined tuning methods

This chapter captures the main features and specifics of the examined parameter-tuning algorithms.

3.1 Overview

ParamILS

ParamILS [34] is a method for parameter tuning and algorithm configuration. It has been used in dozens of academic applications to improve state-of-the-art solvers for more then ten hard computational problems (see its 250+ Google scholar citations). ParamILS has yielded substantial speedups of state-of-the-art solvers for hard combinatorial problems, such as propositional satisfiability (SAT) [30], mixed integer programming (MIP) [31], AI planning [46], answer set programming (ASP) [35], and timetabling [23]. By doing so, ParamILS has helped several systems win solver competitions. ParamILS belongs to the family of local search algorithms hence the name ILS (Iterated Local Search) and it uses adaptive capping as mentioned in section 2.7. It can optimize all kinds of parameter settings as long as their domain is enumerated.

SPOT

Sequential Parameter Optimization Toolbox (SPOT) [17] is a R [16] package that serves as a toolbox for tuning and understanding simulation and optimization algorithms. SPOT includes methods for tuning based on classical regression and analysis of variance techniques:

- **tree-based models** such as CART and random forest
- **Gaussian process models** (Kriging)
- and **combinations** of different meta-modeling approaches

As the name suggests SPOT is also a representative of the SPO group with many models that could sufficiently fill a whole case study on parameter tuning, this thesis uses SPOT's (at the time) default random forest model.

SMAC

SMAC means Sequential Model-based Algorithm Configuration [33]. It comes from the same university and authors participating in ParamILS project, that's why it is very similar in usage and interface. SMAC is a member of SPO method group and it can also optimize all kinds of problems without the need of sampling the real value parameters.

Spearmint

Spearmint [14] is a software package to perform Bayesian optimization. It is based on work of Snoek et al. [45]. It iteratively adjusts a number of parameters modeled as a sample from a Gaussian process (GP) to minimize some objective in as few runs as possible. From this point of view it fulfills the definition of SPO method.

IRACE

Iterated Racing for Automatic Algorithm Configuration (IRACE) [36] is an extension of F-race method for automatic configuration of optimizing algorithms introduced by Birattari et al. [22] and implemented in R [16]. It is no surprise that IRACE represents the group of *racing* methods as well as *sharpening* because their combination has even better effect on the result.

3.2 Algorithms

3.2.1 ParamILS

ParamILS is implemented as traditional local search (Iterative Local Search - ILS) with several variants and heuristics. *Basic* algorithm variant executes following loop of orders:

1. Initialization / Perturbation - finds new search starting point
2. Local search - (in one-exchange neighborhood)
3. Acceptance criteria

In the first phase the algorithm initializes a new search either from the beginning or by means of deviation from current best result (the perturbation step is well-defined with determined number of steps), and then proceeds with the search.

Then it searches the parameter space in one-step neighborhood meaning it iterates all possible configurations differing in one parameter from the current candidate solution. If it finds configuration that is better it stops the search and checks whether the new candidate solution comply with the acceptance criteria and optionally accepts it to the next iteration.

The algorithm then runs in described manner until termination criterion (e.g. timeout, number of iterations, finding optimum etc.) is met.

Focused variant is different only in comparing the candidate configurations, as it incorporates adaptive capping mentioned in section 2.7.

3.2.2 SMAC

As its name suggests, SMAC uses a model to capture the solution space, the basic loop of the algorithm looks like this:

1. Fitting model
2. Selecting configurations
3. Intensify - configuration evaluation on problem instances and result saving

SMAC uses Random Forest as a prediction model, which is sampled for new configurations. Algorithm then proceeds similarly as ParamILS by means of introducing method the authors call Random Online Aggressive Racing (ROAR), which is very similar to the FocusedILS of the previous algorithm. The cornerstone of this method lies in that every configuration is always computed on several randomly chosen instances. The configuration is then compared with the the others based on mutually computed instances. Every configuration is computed on random instances until it is dominated by other configuration (based on statistic made from the mutually computed instance results), which stops all its further computations. Every configuration follows subsequent computations as long as it keeps up with others, which is why the author call it racing procedure.

3.2.3 Irace

Irace assumes for every parameter a probability distribution function and the final configuration is sampled from composed distribution over all parameters. Basic loop of the algorithm is very easy to grasp:

1. Configuration sampling (according to initialized / learned distributions)
2. Choosing the best configurations by means of racing
3. Distribution functions update

Every parameter has independent probability distribution function over its value spans, discrete for categorical parameters (restricted number of values), and normal distribution for real-valued parameters.

Distribution update step updates mean and variation for numeric parameters as well as probabilities of all values for discrete parameters in a way that the yet best found configurations have higher probability to be sampled than the others.

The racing itself works that in every step new configurations are initialized, statistic of their performance is then measured on some randomly chosen sample of instances and according to this statistic it is decided whether given configuration is perspective or not. More precisely whether it competes with others and if not it is no longer evaluated.

To compare the configurations, some type of statistical test is used, usually Friedman test or T-test. Algorithm then runs until it runs out of time, computational budget or there is no quality gain in several iterations.

3.2.4 SPOT

SPOT uses populations of parameter configurations to build and improve meta models used to predict new configuration populations, all in iterative manner. Number of computed evaluations is used to improve confidence over predicted estimated utility. Main loop is:

1. Build meta model(s) f based on current population X
2. Sample model and generate configuration set $X^{'}$
3. Calculate predicted utilities $f(x^{'})$ for each $x^{'} \in X^{'}$
4. Take set $X^{''}$ of d best configurations from $X^{'}$ where $(d \ll l)$
5. Run A with best individual from X and update estimated utility (improve confidence)
6. Let $k = k + 1$
7. Run A with each $x \in X^{''}$ k times to determine estimated utility
8. Extend population $X = X \cup X^{''}$

It runs the tuned algorithm A with the candidate configurations to determine the estimated utility of the model. The value of k is usually initially set to 10 and it keeps track of how many iterations were computed for easy and consistent (confident) comparability.

The algorithm finishes after termination condition is met, which can be user specified.

3.2.5 Spearmint

The algorithm basically builds some probabilistic model using Gaussian processes as source of prior and *Acquisition function* that constructs utility function from the model posterior. It then tries to optimize the values of expected improvement or rather speed of expected improvement as the algorithm tries to do as few iterations as possible.

3.3 Performance metrics

The performance metrics form a vital part of the parameter tuning process. User of some parameter tuner should analyze the requirements and expected outcome in order to choose metric that best suits the problem.

User can choose according which quality metric to optimize for each algorithm run (runtime, run length, absolute quality, approximate quality (ratio of the best found solution and optimum), speedup, specific function) and overall performance over whole set of instances (average mean, geometric mean, median, chosen percentile, geometric median).

Luckily for the user, all of the above mentioned metrics can be chosen in ParamILS settings. SMAC provides only the choice between QUALITY or RUNTIME, more sophisticated measures must user provide explicitly. IRACE, SPOT and Spearmint only work with single objective value (usually as a real number), so the user must adjust the underlying algorithm according to the desired measurement.

Chapter 4

COCO benchmarking Framework

Comparing Continuous Optimizers (COCO) [3] is a platform for systematic and sound comparisons of global optimizers on real-valued functions. It provides benchmark function testbeds as well as tools for processing and visualizing data generated by one or several optimizers. The COCO platform has been used for the Black-Box-Optimization-Benchmarking (BBOB) workshops that took place during the GECCO conference in 2009 [5], 2010 [6], 2012 [7], 2013 [8], and in 2015-2017 [9, 10, 11]. It was also used at the IEEE Congress on Evolutionary Computation [1] (CEC'2015) in Sendai, Japan.

4.1 Overview

COCO is a tool for benchmarking algorithms for black-box optimization. Overall about 100 different algorithms and articles have been benchmarked and written respectively. COCO provides:

- **experimental framework** used for algorithm testing and benchmarking
- **post-processing tools** generating publication quality figures and tables from gathered data
- **LaTeX article templates** to present the figures and tables in a single document

The practitioner in continuous optimization who wants to benchmark one or many algorithms has to download COCO, plug the algorithm(s) into the provided experimental template and use the post-processing tools for generating figures and tables that can be used in provided LaTeX templates.

4.2 Purpose

Quantifying and comparing performance of numerical optimization algorithms is one important aspect of research in search and optimization. However, this task turns out to be tedious and difficult to realize even in the single-objective case — at least if one is willing to accomplish it in a scientifically decent and rigorous way.

COCO provides tools for most of this tedious task:

- **choice and implementation of benchmark** with single-objective function testbed
- **design of an experimental set-up**
- **generation of data** output for post-processing
- **presentation of the results** in graphs, tables or even whole articles

4.3 Testing functions and their benefits

COCO has been used in several workshops during the GECCO conference since 2009 (Black-Box Optimization Benchmarking (BBOB) 2009) [5]. For these workshops, a testbeds of 24 noiseless functions and 30 noisy functions are provided.

All functions can be instantiated in different "versions" (with different location of the global optimum and different optimal function value). Each "version" of each function is differently rotated and scaled.

The functions come in different variants for several dimensions. The current function testbed has dimensions $D = \{2, 3, 5, 10, 20, 40\}$. All functions of BBOB are defined everywhere in $\mathbb{R}^D$ and have their global optimum in $[-5, 5]^D$. Most BBOB functions have their global optimum in the range $[-4, 4]^D$ which can be a reasonable setting for initial solutions. User can choose the testbed under consideration, i.e. different algorithms and/or parameter settings can be used for the noise-free and the noisy testbed. The final target precision has been set $\Delta f = 10^{-8}$ in order to implement effective termination and restart mechanisms (which should also prevent early termination). User can also set the overall number of evaluations, in order to reduce the overall CPU requirements.

In order to provide thorough testing of the benchmarked algorithm the testbed functions [27] are chosen with variety of different properties. Some properties are ensured by the rotation and scale variance in each of the functions 15 instances. The function testbed is composed to differ in several key features and their combinations:

- **modality :** Functions can be unimodal (with one prominent mode of their probability density function) or multi-modal which tend to have many local optima thus testing the ability to avoid premature convergence
- **separability :** Some functions are separable so the optimum can be searched by solving several disjoint problems with lower dimensionality, testing whether the optimizer can exploit such behavior
- **conditioning :** Testing functions are also equally divided into well-conditioned(small change of input parameters have small effect on the function value) and ill-conditioned (even small change of input parameters greatly affects the result). The typical well-established technique to generate non-separable functions from separable ones is the application of a rotation matrix R as it is commonly used in this benchmark.

- **regularity :** In order to create irregularities in functions created from simple formulas some non-linear transformations are applied. The testbed also contains some highly irregular functions. This is done to test more real-life examples and to disrupt strictly gradient methods.
- **symmetry :** Better part of the testbed is also asymmetric either by nature or by introducing asymmetric transformation to otherwise symmetric functions because it has been argued that symmetric benchmark functions could be in favor of Stochastic search procedures as these operators often rely on Gaussian distributions to generate new solutions.
- **structure :** Several function examples have intricate structure that can be somehow exploited but it rather makes the search for optimum more difficult. Functions with high degree of ruggedness, repetitive landscape (multi-modality or plateaus) weak or no global structure or some degree of non-differentiability. The testbed even has artificially generated functions with many optima with random height and position, with uneven conditioning in different regions of the function space.

4.4 Benchmarking results and their meaning

COCO framework post-processing feature generates several kinds of graphs [4]. Figure 4.1 shows a scatter plot of run lengths (log_{10} scale) comparing two algorithms in different dimensions on a single function.

Figure 4.2 shows the run length distribution of two algorithms for different target difficulties $\{10, 10^{-1}, 10^{-4}, 10^{-8}\}$ on a set of 24 noiseless functions. The expected run length (in COCO it is called Expected Run Time (ERT)) distribution is a measure defined by Hansen et al. in [28] and it basically shows the proportion of problems solved within target precision given the evaluation count. Figure 4.4 shows ERT distribution for more algorithms [28].

Figure 4.3 shows box-whisker plots of the loss ratios of the expected run length compared to best (shortest) observed ERT in BBOB-2009.

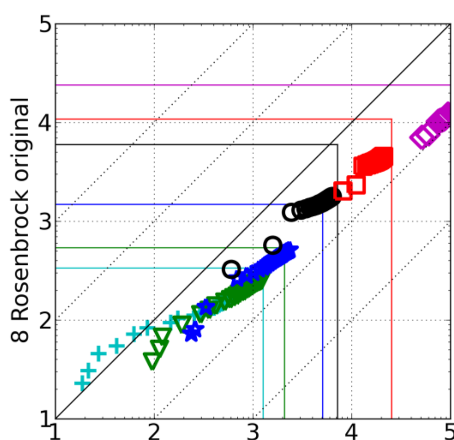


Figure 4.1: Scatter plot comparing two algorithms run lengths (log_{10} scale) on a single function in different dimensions

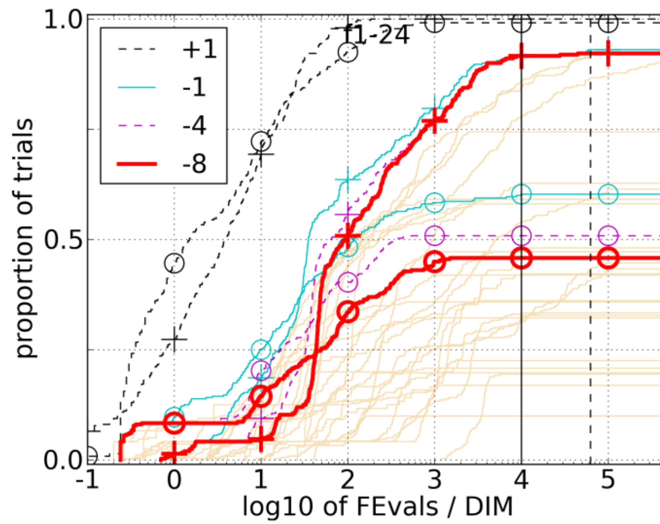


Figure 4.2: Run length distribution of two algorithms for different target difficulties $\{10, 10^{-1}, 10^{-4}, 10^{-8}\}$

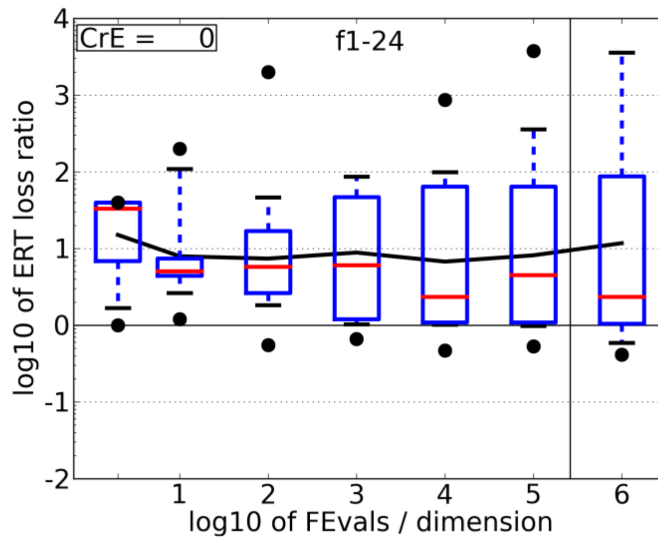


Figure 4.3: Box-whiskers plots of the loss ratios of the expected run length compared to best (shortest) observed expected run length in BBOB-2009

Figure 4.4: Empirical runtime distributions (runtime in number of function evaluations divided by dimension) on all functions with target precision in 100, ... , 1e-8 in dimension 10. The cross indicates the maximum number of function evaluations. A decline in steepness right after the cross (e.g. for IPOP-SEP-CMA-ES) indicates that the maximum number of function evaluations should have been chosen larger. A steep increase right after the cross (e.g. for simple GA) indicates that a restart should have been invoked earlier.

Chapter 5

Conducted Experiments

This chapter is dedicated to the performed measurements and their relation to the specified goals. It analyzes the capabilities of chosen tuning frameworks and methods and their interoperability with the chosen benchmark. It also presents the reasoning behind individual algorithm usages and settings.

5.1 Experiment analysis

This section briefly presents the factors and circumstances that had to be taken into account before the experiments themselves were conducted.

Implementation requirements

Several frameworks were implemented in different computer languages. In order to compare and test them in a one environment, it was necessary to connect them through the language implementations of COCO framework. COCO provides several language interfaces, specifically MatLab, R, C, Java and Python. More on algorithm implementation languages is in appendix A.1.

COCO experiment requirements

There are two main requirements that need to be fulfilled in order to successfully run and compare the optimization methods inside the COCO framework.

1. The COCO framework serves as a benchmark for continuous optimization so it is important that the optimizer can work with real valued parameters.

2. In order to execute the chosen methods on benchmark instances, the interoperability between each method and the framework must be achieved. This ideally means to wrap the tuner function to comply with the COCO benchmarking interface. If that would be time-consuming or not easily done, it is possible to generate the benchmark results by directly calling the benchmark functions from the specific optimizer and then group the results to match the structure needed for post-processing and figure generation.

Benchmark testbed

As we know from section 4.3 COCO provides 24 noiseless functions and 30 noisy functions to test the strengths and weaknesses of each optimizer. For the purposes of this thesis only the noiseless testbed was used as it provided enough data and consumed enough resources to prevent another computations to be finished in time. To test the parameter tuning abilities and the optimizers generalization for different problem classes, the testbed was divided in two groups. The first class of the noiseless testbed is composed from unimodal functions and the second class forms all the multi-modal functions in the testbed.

Optimizing capabilities

To test the optimizing capabilities of the investigated frameworks and methods the tuners were connected to the benchmark framework and used as regular optimizers.

Parameter tuning capabilities

To test the meta-optimizing abilities of presented tuners on the COCO benchmark functions it was needed to run the parameter tuner with some standard optimizer on the COCO benchmark instances. Two widely known and well established algorithms were used as the tuned optimizers, Differential Evolution (DE) and Genetic Algorithm (GA). Both population-based and pretty efficient on their own they proved themselves as ideal candidates because of decent number of parameters to optimize and publicly available implementations used in the research community [15, 43].

Suitability

Each parameter tuner had to be confronted with the benchmarking requirements to be tested or not. Although the ParamILS algorithm meets the requirement of handling real values as parameters, all the parameters have to be sampled and enumerated as possible parameter values. That is rather infeasible setting because it solves different problem than the one presented in the benchmark testbed thus making the results more or less incomparable. That is the reason the ParamILS was not tested against the benchmark.

5.2 Realization

This section focuses on the details of experiment execution. Its structure, resources, unusual procedures that were employed, various settings used and the way they were used.

5.2.1 Used resources

Most of the experiments were conducted on several laptops and computers of similar specification.

- **CPU** - Intel Core i7 2.50 GHz
- **RAM** - 8.00 GB
- **OS** - Windows 10 64-bit

Some experiments were conducted using joined resources of National Grid Infrastructure MetaCentrum [13] and CERIT Scientific Cloud (CERIT-SC) [2] on which similar computation resources were allocated for each computation (with the exception of the operating system as the cloud infrastructure was accessible through linux based front-end machines).

5.2.2 Optimization experiment structure

For each of the parameter tuning methods the following steps were executed to test them as optimizers.

1. **Connect the method to benchmark.**
   This usually meant to write a wrapper function that accepts COCO input parameters such as the optimized function, number of evaluations or dimension bounds and performs the optimization based on the called function values.

2. **Benchmark the method.**
   This usually meant to let the COCO framework benchmark the wrapper function on the entire testbed and gather the computed data.

3. **Post-processing.**
   To gather some data from the benchmark is not enough because the post-processing tool is rather sensitive to the data structure and location. After several trial-and-error method executions the benchmarked data were ready to be processed and shown.

5.2.3 Optimization experiment execution

The instantiation and execution of the examined methods were following:

- **Number of evaluations.** All the instances of all the benchmark functions had the maximum number of evaluations set to $100D$, where D is the dimension of the problem. Such limit was chosen from the practical point of view as it was computationally demanding especially for higher dimension versions of the tested functions. It corresponds to 120000 evaluations for each of the 24 noiseless functions making it 2880000 evaluations per tuning method.

- **Number of functions.** As the whole noiseless test bed was used to benchmark the optimizing abilities, all 24 functions were used.
- **Number of instances.** Each benchmark function has 15 different instances to check out the target function, all of them were used to test the tuners.

SMAC

The SMAC method was rather tricky to use as it was available only as compiled executable that requires wrapper script to run the desired optimizer (in this case the benchmark function). Also parameters of the underlying algorithm must be specified in separate file carried as command line parameter. To run the SMAC method a Java program that executed SMAC with benchmark function wrapping script as parameter was created plus another program that parsed SMAC output to generate consistent COCO data. The SMAC executing program also provided the parameter definition for each of the examined function instances.

SPOT

SPOT toolbox contains function *spotOptim* that more or less fulfills the standard interface for optimization function in R. As the COCO interface in R requires the wrapper function in the said form, the only thing the *spotOptim* function needed was prediction model specification. The (at the time of computation) default model was random forest.

Although the experiment was easy to setup it was rather lengthy in terms of computation times. It was necessary to divide it into several separate computations. The easiest way was to divide the computation into 6 parts for each dimension in $\{2, 3, 5, 10, 20, 40\}$. The parallel computation was conducted using the Metacentrum grid [13].

IRACE

The IRACE R package unfortunately does not have the standard R *optim* interface so the parameters had to be specified once again, with the benefit of providing them either in the form of parameter file that IRACE package can parse or as named parameter list in R. Besides the parameter definition the IRACE main function needs the problem to be specified in a named list called *tunerConfig* (in later versions this is called *scenario*) in which the maximum number of evaluations as well as the optimized function is specified.

Spearmint

The Spearmint software package was quite difficult to benchmark. The problem lies within the optimized problem definition. The spearmint method requires each problem instance to be in its own directory, specified by *config.json* file within that directory. Within the same folder a script that serves as the tuned optimizer (benchmark function in this case) must be placed as well. For 6 dimensions, 24 functions and 15 instances of every function that gives us 2160 folders and twice as many files that must be generated specifically to fit each benchmark definition. The spearmint experiments were also conducted in Metacentrum grid to save the running time, as every evaluation took quite some time with the accompanying database synchronization and the model fitting.

5.2.4 Parameter tuning structure

The parameter tuning procedure, that was designed and executed for this thesis has the following steps:

1. **Connecting the optimizer with parameter tuner.**
   It is required that the parameter tuner can call the underlying optimizer with specific parameters and acquire the optimized result value.

2. **Connecting the optimizer with benchmark functions.**
   It is required that the optimizer can call the benchmark function and optimize its value.

3. **Testbed initialization.**
   In the first phase a part of the benchmark testbed that belongs to the specific problem class is chosen and divided in two separate sets of training and testing functions.

4. **Training the optimizer.**
   The training phase is focused on searching the best performing set of input parameters for the optimizer. The parameter tuner thus tries different parameter configurations and returns one or a few configurations with the best performance statistic over the training functions.

5. **Testing the optimizer.**
   The testing phase takes the tuned optimizer and evaluates its performance against itself with the default setting. The comparison tests of trained and default optimizer variants are run over the set of training functions, testing functions within the same problem class and another set of previously unseen functions outside the investigated function class.

5.2.5 Parameter tuning execution

The following paragraphs describe the experimental setup used for parameter tuning.

Choice parameter tuning framework

Due to rather demanding implementation requirements on the communication layers between both the optimizers and the testing framework as well as the layers between each optimizer and all the parameter tuning methods only one parameter tuning procedure was tested.

The tested package was chosen to be IRACE mostly because the there was the least amount of problems regarding its interoperability with the testing framework as well as the fact that there already are public R packages of the tested global optimizers which made their inclusion easier.

Parameter tuning experiment setup

Each of the examined optimizers was trained on unimodal and multimodal functions respectively. Both the unimodal-trained and multimodal-trained version of each optimizer was tested against the algorithm's default setting. Tests were computed on 3 sets of functions.

- **Training set** was represented by the optimizer's training set. This was done to compare the performance enhancing abilities of the tuner against the algorithm default setting.
- **Testing set** functions belong to same class as the training ones but the algorithm was not trained on them. This was done to explore the generalization capabilities of the parameter tuner.
- **Other set** consisted of different class functions. This was done to check whether the trained algorithm variant preforms worse than the default one on different problem class.

Instantiation of parameter tuner

For each of the 2 optimizing problem classes randomly chosen half of the functions was marked as training and the other half as testing. For each function, two instances in each dimension were randomly chosen to represent the function mostly due to the computational demands. Because there were 2 instances per function in each of the 6 dimensions and half of the 12 functions within the same unimodal/multi-modal class were marked as training, each training and testing procedure was conducted on 72 instances.

The parameter tuner was chosen to have maximum number of optimizer calls 100 times the number of instances which means that it will generate at least 100 parameter configurations that are equivalent in the means of racing over the whole set of training instances. As the minimum number of instances to optimize in order to decide whether one configuration is outperformed and its optimization is stopped was left to be 5 (default), one can safely assume that there will be more than 100 tested configurations.

To have a better understanding of how many evaluations are made in the training part of the parameter tuning it is good to know that each optimizer call runs $1000D$ evaluations of the specified function instance of dimension D, where $D \in \{2, 3, 5, 10, 20, 40\}$. We know that there are 72 instances from which 12 dwell in each dimension because there are 6 functions with 2 instances per dimension. That means there can be up to 960 thousand evaluations per one configuration. Knowing that there can be 100 fully successful configurations we can safely guess that the training part of parameter tuning can make 96 million evaluations in search for the optimal parameter setting.

Parameters of the optimizers

The algorithms were optimized in these parameters:

- **GA**
  - **populationSize**, integer, $\in [5, 200]$, default is 50
    Number of candidate solutions in every generation (iteration)
  - **crossoverProbability**, real, $\in \langle 0.5; 1 \rangle$, default is 0.8
    The probability of crossover between pairs of chromosomes.
  - **mutationProbability**, real, $\in \langle 0; 0.5 \rangle$, default is 0.1
    The probability of mutation in a parent chromosome. Usually mutation occurs with a small probability.
- **DE**
  - **populationSize**, integer, $\in [5, 200]$, default is 50
    Number of candidate solutions in every generation (iteration)
  - **crossoverProbability**, real, $\in \langle 0; 1 \rangle$, default is 0.5
    The probability of crossover between candidate triplet and the parental chromosome.
  - **weightFactor**, real, $\in \langle 0; 2 \rangle$, default is 0.8
    Step size used in Differential Evolution strategy.
  - **crossoverSpeed**, real, $\in \langle 0; 1 \rangle$, default is 0
    It controls the speed of the crossover adaptation. Higher values of *crossoverSpeed* give more weight to the current successful mutations.
  - **strategy**, integer, $\in [1, 7]$, default is 2
    It defines the Differential Evolution strategy used for crossover

Enhanced Performance measure

Two different parameter configurations may yield similar results using the same computational budget. Especially when the optimizer found optimum in both cases it is difficult to determine whether one configuration performed better based only on the function objective value.

In figure 5.1 we can see 2 sets of data points representing 2 differently configured optimizer runs. Both of them achieved the same objective value, but we can see that one of them did converge faster, generating better results in the process of optimization. That is why the quality measure of one configuration was changed to the area under the optimizers convergence curve. It reflects the whole optimization process and serves for better distinction between well and poorly-performing parameter settings.

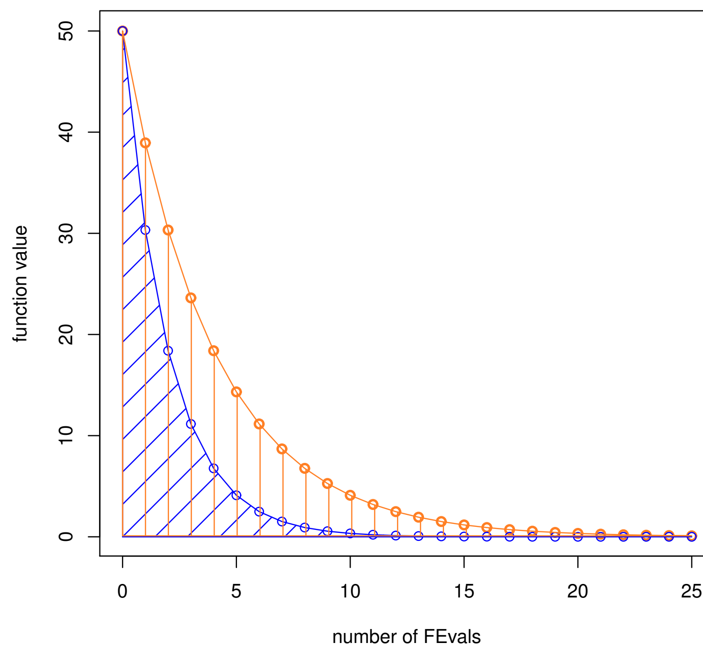


Figure 5.1: *Area under curve performance metric.* Although both optimizer configurations achieved the same minimal function value, we can see that the one represented by the blue data points made the optimizer to converge faster thus having better performance.

Chapter 6

Results

This chapter contains all the relevant computed data. The first part shows the results of benchmarking the parameter tuners as standalone optimizers. The second part is aimed at tuning parameters of GA and DE optimizers using the IRACE tuning method.

6.1 Optimization experiment results

Here are the optimization benchmark results of the presented tuning methods. Some datasets were not obtained due to the difficult or unfinished evaluation or implementation. One dataset was added for the SMAC algorithm available at [3] and presented by Hutter et al. [32].

6.1.1 Experiments with negative results

Apart from disqualifying the ParamILS algorithm from the benchmark testing, some experiments were not evaluated entirely and ended up unfinished.

SPOT
Computations of SPOT benchmarking were evaluated up until the 20-dimensional version of the function testbed. The 40-dimensional case was tried several times and was always ultimately killed as its computation exceeded the assigned 4 weeks of cpu-time.

Spearmint
Spearmint optimization was rather slow to compute all the instances sequentially thus it was necessary to run them in parallel. The original idea was to assign all of the 2160 subproblems to the cloud computing queue [2], but as it turned out transferring the necessary packages to each of the computing nodes generated rather large traffic on planning server side so the assigned jobs were automatically killed.
The second attempt in benchmarking the spearmint tuner would use one computing node as master to get the benchmarks from the planning server, distribute them on many slaves which would evaluate the benchmark instances and then gather data from the slaves and return them all at once. Again as some of the benchmarking nodes failed to deliver their computations within the assigned time frame, the master node did also fail to deliver the computed data.

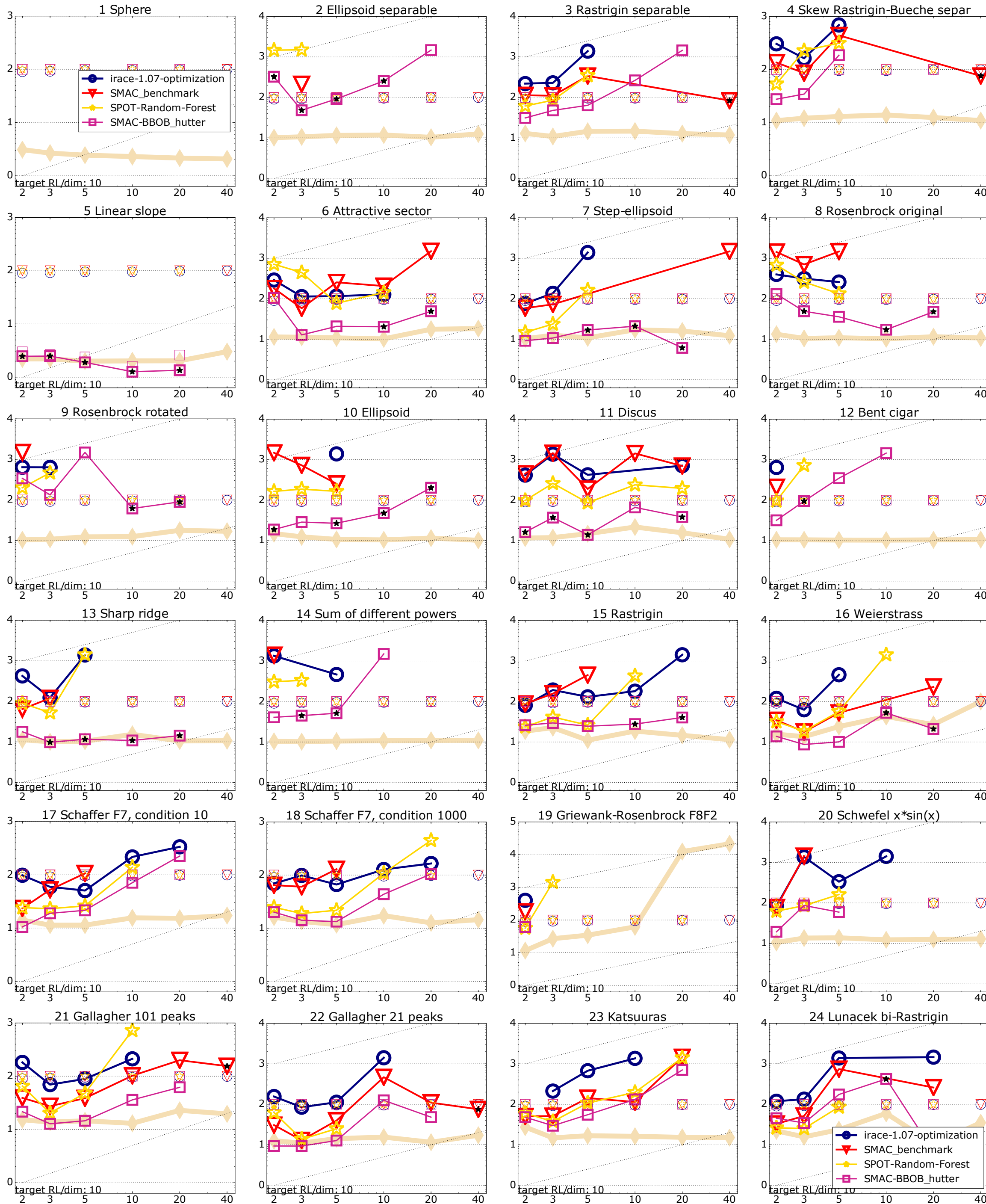


Figure 6.1: Expected running time (ERT in number of f-evaluations as $\log_{10}$ value) divided by dimension versus dimension. The target function value is chosen such that the bestGECCO2009 artificial algorithm just failed to achieve an ERT of $10 \times \mathrm{DIM}$. Different symbols correspond to different algorithms given in the legend of f_1 and f_{24}. Light symbols give the maximum number of function evaluations from the longest trial divided by dimension. Black stars indicate a statistically better result compared to all other algorithms with $p < 0.01$ and Bonferroni correction number of dimensions (six). Legend: ○:irace-1.07-optimization, ▽:SMAC_benchmark, ⋆:SPOT-Random-Forest, □:SMAC-BBOB_hutter_noiseless.

6.1.2 Expected run length comparison

The figure 6.1 shows the expected run length (ERT) for each tuner for each function over each dimension. It basically states how many evaluations (dimension lengths) are needed to achieve the objective value with the desired precision, all on logarithmic scale. Values that appear to be missing are above the bounds of the presented figures. Light-colored points show the number of evaluations performed ($100D$ for all experiments). We can see that the SMAC dataset by Hutter et al. [32] usually shows the best results.

6.1.3 Algorithm performance comparison

In figures 6.2 and 6.3 the empirical cumulative distribution of ERT is used as a measure for algorithm comparison. It basically shows the proportion of problems solved within target precision given the evaluation count. We can see that the SMAC dataset by Hutter et al. [32] shows the best performance.

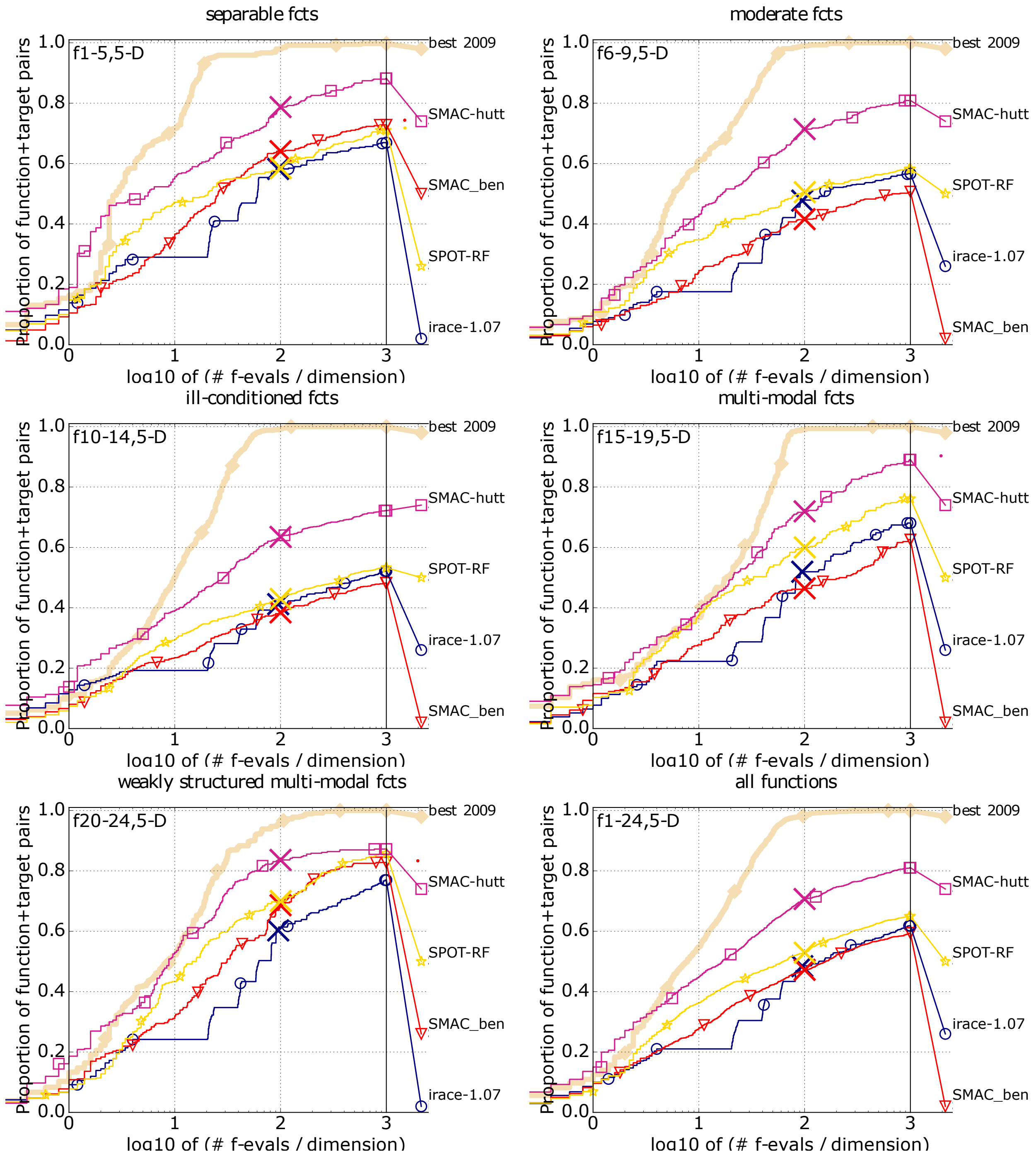


Figure 6.2: Bootstrapped empirical cumulative distribution of the number of objective function evaluations divided by dimension (FEvals/DIM) for all functions and subgroups in 5-D. The targets are chosen from $10^{[-8..2]}$ such that the bestGECCO2009 artificial algorithm just not reached them within a given budget of k × DIM, with k ∈{0.5, 1.2, 3, 10, 50}. The "best 2009" line corresponds to the best ERT observed during BBOB 2009 for each selected target.

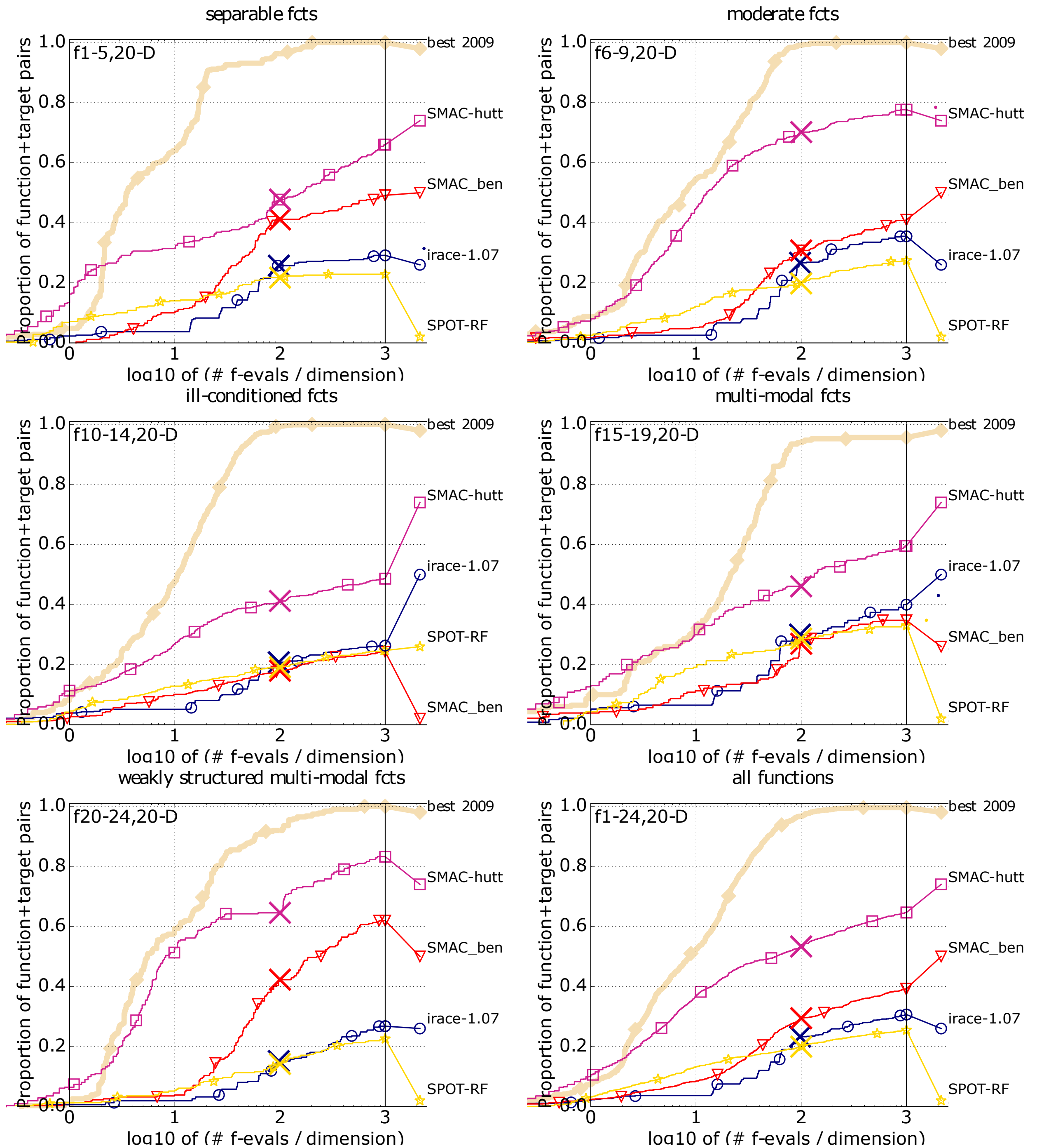


Figure 6.3: Bootstrapped empirical cumulative distribution of the number of objective function evaluations divided by dimension (FEvals/DIM) for all functions and subgroups in 20-D. The targets are chosen from $10^{[-8..2]}$ such that the bestGECCO2009 artificial algorithm just not reached them within a given budget of k × DIM, with k ∈{0.5, 1.2, 3, 10, 50}. The "best 2009" line corresponds to the best ERT observed during BBOB 2009 for each selected target.

6.2 Parameter tuning experiment results

Here are the results of parameter tuning of GA and DE algorithms on unimodal and multimodal function classes with *training*, *testing*, and *other* testbeds (see 5.2.5).

The presented results are generated by the post-processing tools of COCO showing lower and higher dimensional performance in 5 and 20 dimensions respectively. The crosses in each dataset represent the maximum number of evaluations for the given algorithm run.

6.2.1 GA unimodal tests

Figures 6.4, 6.5 and 6.6 show the cumulative distribution of ERT for GA. They compare the best configurations IRACE has trained on the set of unimodal function testbed with the default configuration. Figure 6.4 shows tests that were evaluated on the same training testbed. Figure 6.5 shows tests that were evaluated on unseen unimodal functions. Figure 6.6 shows tests that were evaluated on multimodal function testbed.
On the multimodal testbed the default algorithm configuration is best as expected. On the training and testing set of unimodal functions all the configurations performance is similar although the default one seems slightly better.

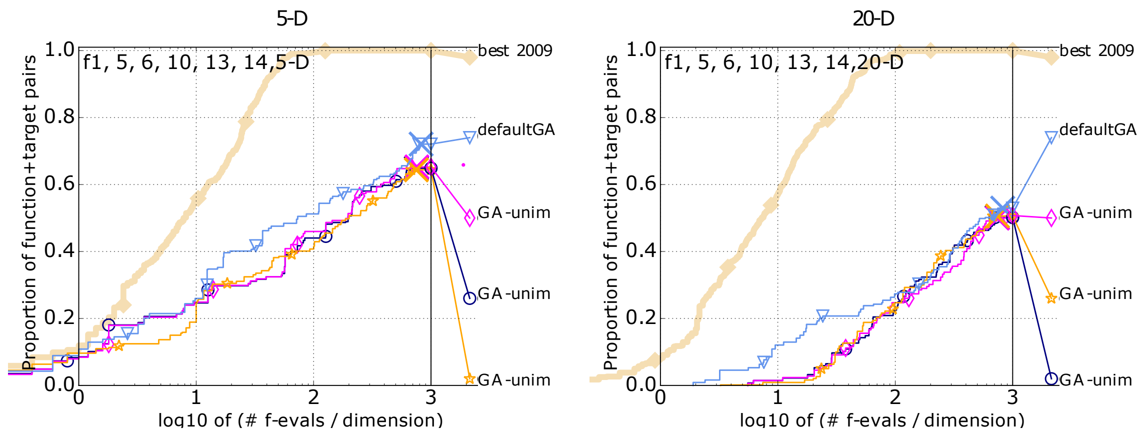


Figure 6.4: Bootstrapped empirical cumulative distribution of the number of objective function evaluations divided by dimension (FEvals/DIM) for all functions and subgroups in 5 and 20-D. The targets are chosen from $10^{[-8..2]}$ such that the bestGECCO2009 artificial algorithm just not reached them within a given budget of k × DIM, with k ∈{0.5, 1.2, 3, 10, 50}. The "best 2009" line corresponds to the best ERT observed during BBOB 2009 for each selected target.

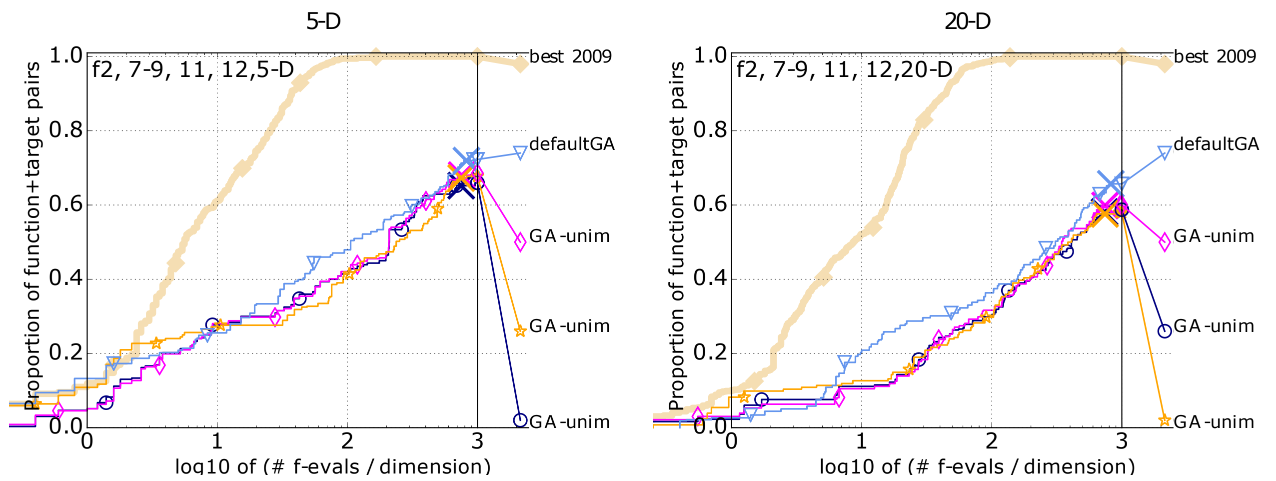


Figure 6.5: Bootstrapped empirical cumulative distribution of the number of objective function evaluations divided by dimension (FEvals/DIM) for all functions and subgroups in 5 and 20-D. The targets are chosen from $10^{[-8..2]}$ such that the bestGECCO2009 artificial algorithm just not reached them within a given budget of k × DIM, with k ∈{0.5, 1.2, 3, 10, 50}. The "best 2009" line corresponds to the best ERT observed during BBOB 2009 for each selected target.

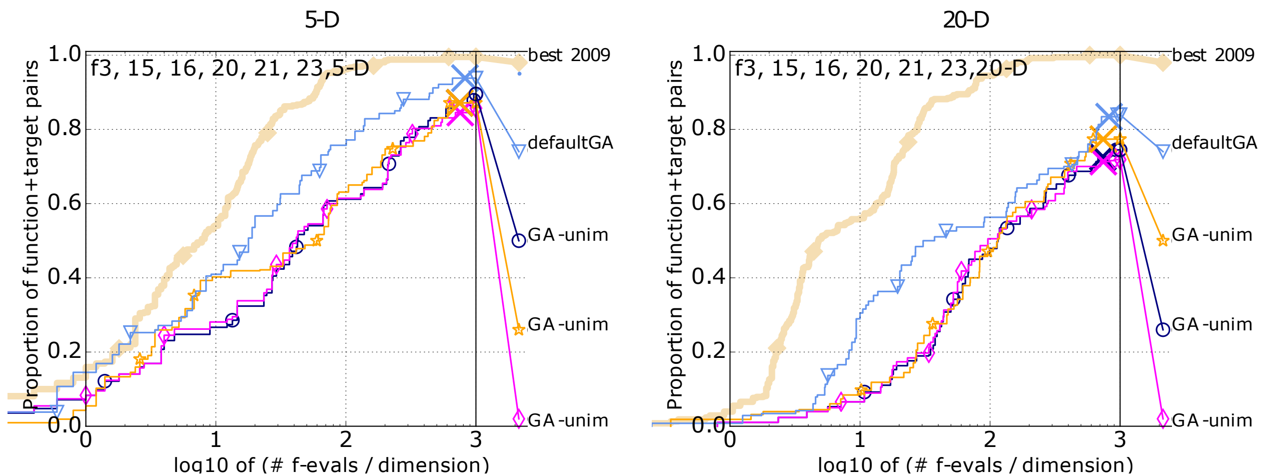


Figure 6.6: Bootstrapped empirical cumulative distribution of the number of objective function evaluations divided by dimension (FEvals/DIM) for all functions and subgroups in 5 and 20-D. The targets are chosen from $10^{[-8..2]}$ such that the bestGECCO2009 artificial algorithm just not reached them within a given budget of k × DIM, with k ∈{0.5, 1.2, 3, 10, 50}. The "best 2009" line corresponds to the best ERT observed during BBOB 2009 for each selected target.

6.2.2 GA multimodal tests

Figures 6.7, 6.8 and 6.9 show the cumulative distribution of ERT for GA. They compare the best configurations IRACE has trained on the set of multimodal function testbed with the default configuration. Figure 6.7 shows tests that were evaluated on the same training testbed. Figure 6.8 shows tests that were evaluated on previously unseen multimodal functions. Figure 6.9 shows tests that were evaluated on unimodal function testbed.
Although all the performances are rather similar in the 5-dimensional scenario, the default configuration is better in all 20-dimensional cases.

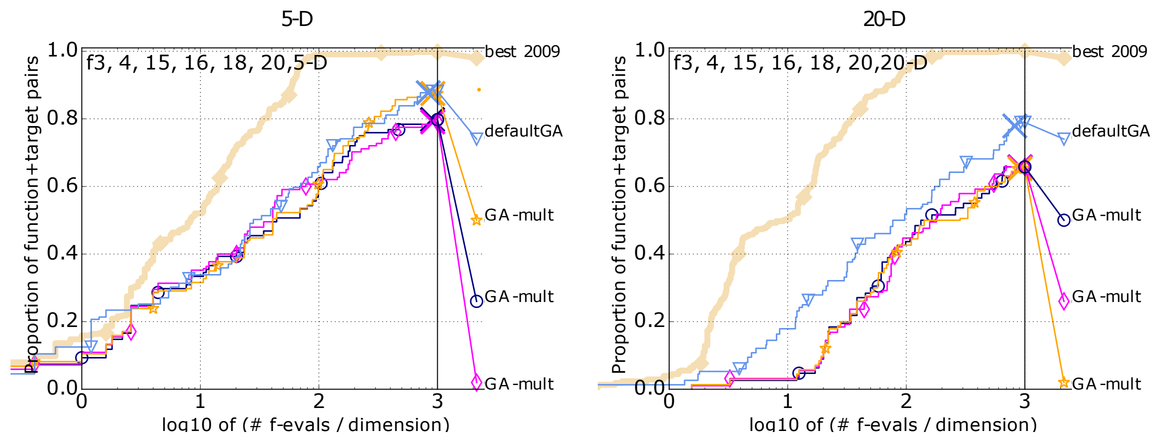


Figure 6.7: Bootstrapped empirical cumulative distribution of the number of objective function evaluations divided by dimension (FEvals/DIM) for all functions and subgroups in 5 and 20-D. The targets are chosen from $10^{[-8..2]}$ such that the bestGECCO2009 artif cial algorithm just not reached them within a given budget of k × DIM, with k ∈{0.5, 1.2, 3, 10, 50}. The "best 2009" line corresponds to the best ERT observed during BBOB 2009 for each selected target.

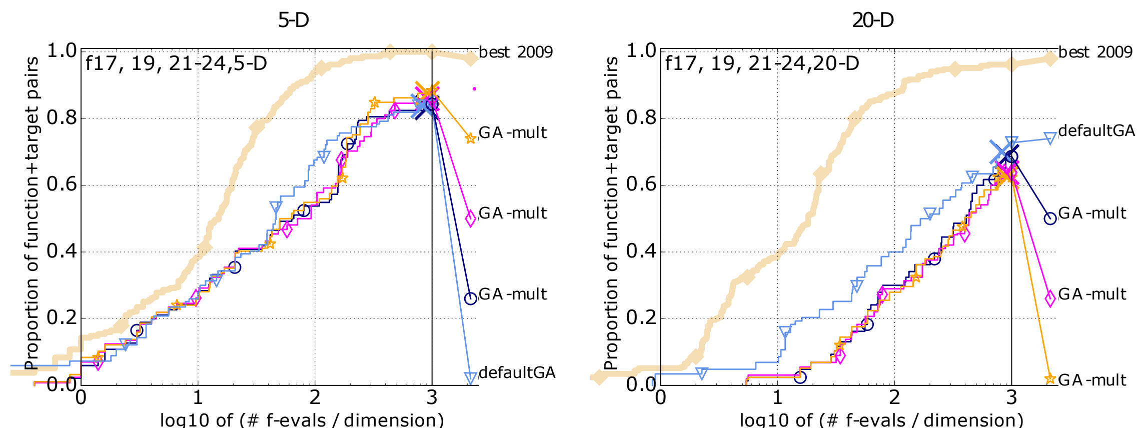


Figure 6.8: Bootstrapped empirical cumulative distribution of the number of objective function evaluations divided by dimension (FEvals/DIM) for all functions and subgroups in 5 and 20-D. The targets are chosen from $10^{[-8..2]}$ such that the bestGECCO2009 artif cial algorithm just not reached them within a given budget of k × DIM, with k ∈{0.5, 1.2, 3, 10, 50}. The "best 2009" line corresponds to the best ERT observed during BBOB 2009 for each selected target.

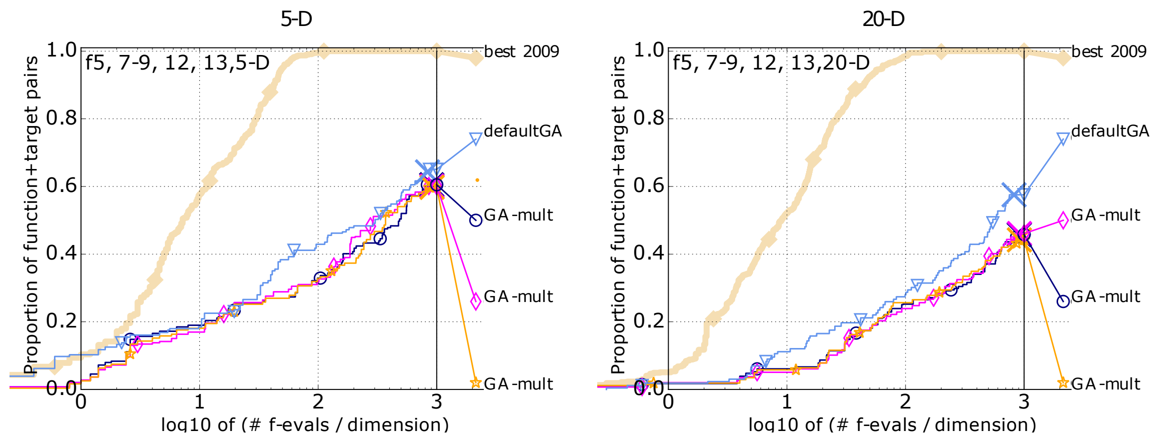


Figure 6.9: Bootstrapped empirical cumulative distribution of the number of objective function evaluations divided by dimension (FEvals/DIM) for all functions and subgroups in 5 and 20-D. The targets are chosen from $10^{[-8..2]}$ such that the bestGECCO2009 artificial algorithm just not reached them within a given budget of k × DIM, with k ∈{0.5, 1.2, 3, 10, 50}. The "best 2009" line corresponds to the best ERT observed during BBOB 2009 for each selected target.

6.2.3 DE unimodal tests

Figures 6.10, 6.11 and 6.12 show the cumulative distribution of ERT for DE. They compare the best configurations IRACE has trained on the set of unimodal function testbed with the default configuration. Figure 6.10 shows tests that were evaluated on the same training testbed. Figure 6.11 shows tests that were evaluated on unseen unimodal functions. Figure 6.12 shows tests that were evaluated on multimodal function testbed.
The trained configurations showed better performance on the training and testing testbed. On the unimodal testbed all the configurations showed similar performance.

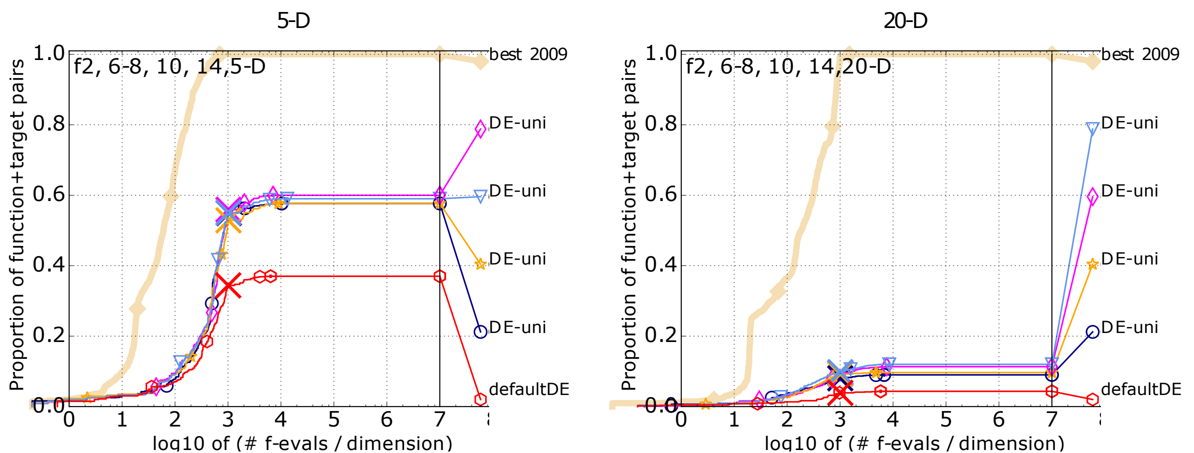


Figure 6.10: Bootstrapped empirical cumulative distribution of the number of objective function evaluations divided by dimension (FEvals/DIM) for 50 targets in $10^{[-8..2]}$ for all functions and subgroups in 5 and 20-D. The "best 2009" line corresponds to the best ERT observed during BBOB 2009 for each single target.

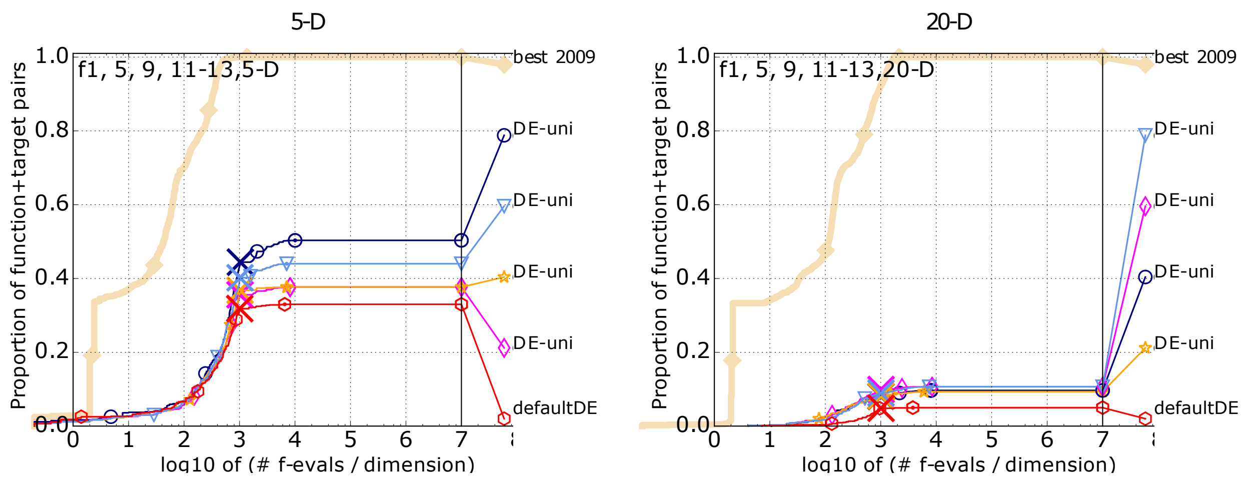


Figure 6.11: Bootstrapped empirical cumulative distribution of the number of objective function evaluations divided by dimension (FEvals/DIM) for 50 targets in $10^{[-8..2]}$ for all functions and subgroups in 5 and 20-D. The "best 2009" line corresponds to the best ERT observed during BBOB 2009 for each single target.

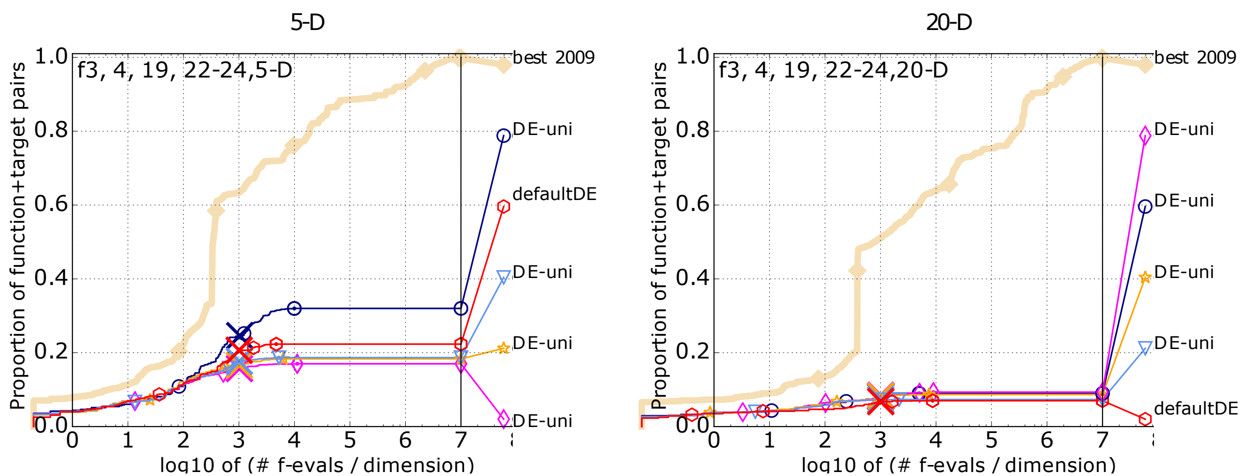


Figure 6.12: Bootstrapped empirical cumulative distribution of the number of objective function evaluations divided by dimension (FEvals/DIM) for 50 targets in $10^{[-8..2]}$ for all functions and subgroups in 5 and 20-D. The "best 2009" line corresponds to the best ERT observed during BBOB 2009 for each single target.

6.2.4 DE multimodal tests

Figures 6.13, 6.14 and 6.15 show the cumulative distribution of ERT for DE algorithm. They compare the best configurations IRACE has trained on the set of multimodal function testbed with the default configuration. Figure 6.13 shows tests that were evaluated on the same training testbed. Figure 6.14 shows tests that were evaluated on previously unseen multimodal functions. Figure 6.15 shows tests that were evaluated on unimodal function testbed.
The trained configurations performed better on all instances, showing only minimal differences in the 20-dimensional cases.

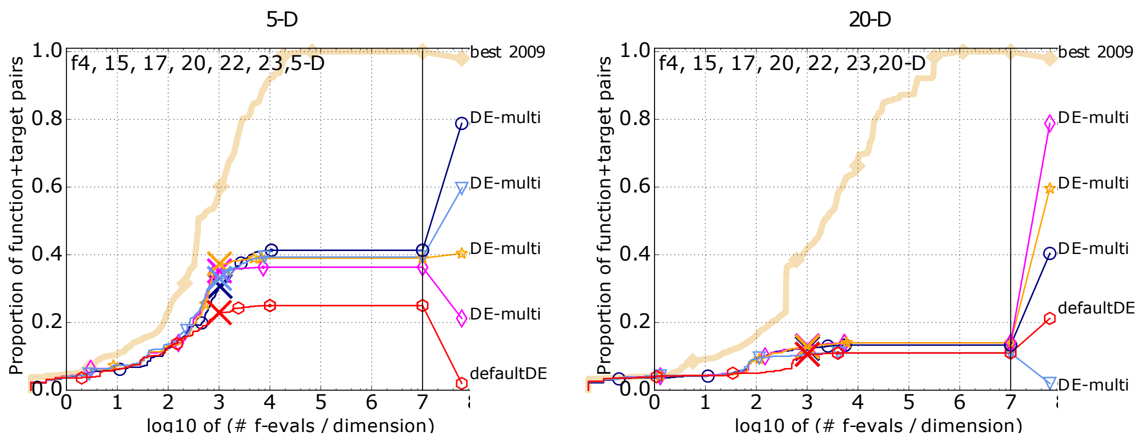


Figure 6.13: Bootstrapped empirical cumulative distribution of the number of objective function evaluations divided by dimension (FEvals/DIM) for 50 targets in $10^{[-8..2]}$ for all functions and subgroups in 5 and 20-D. The "best 2009" line corresponds to the best ERT observed during BBOB 2009 for each single target.

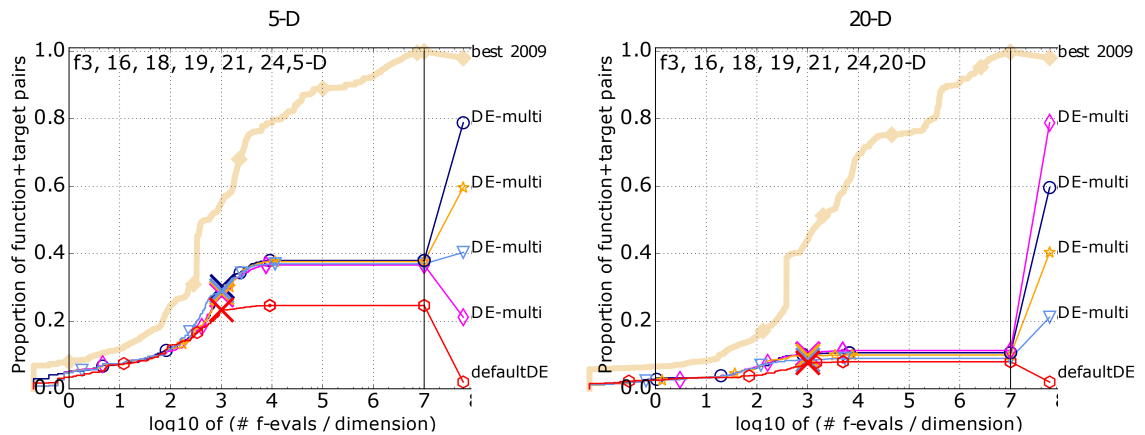


Figure 6.14: Bootstrapped empirical cumulative distribution of the number of objective function evaluations divided by dimension (FEvals/DIM) for 50 targets in $10^{[-8..2]}$ for all functions and subgroups in 5 and 20-D. The "best 2009" line corresponds to the best ERT observed during BBOB 2009 for each single target.

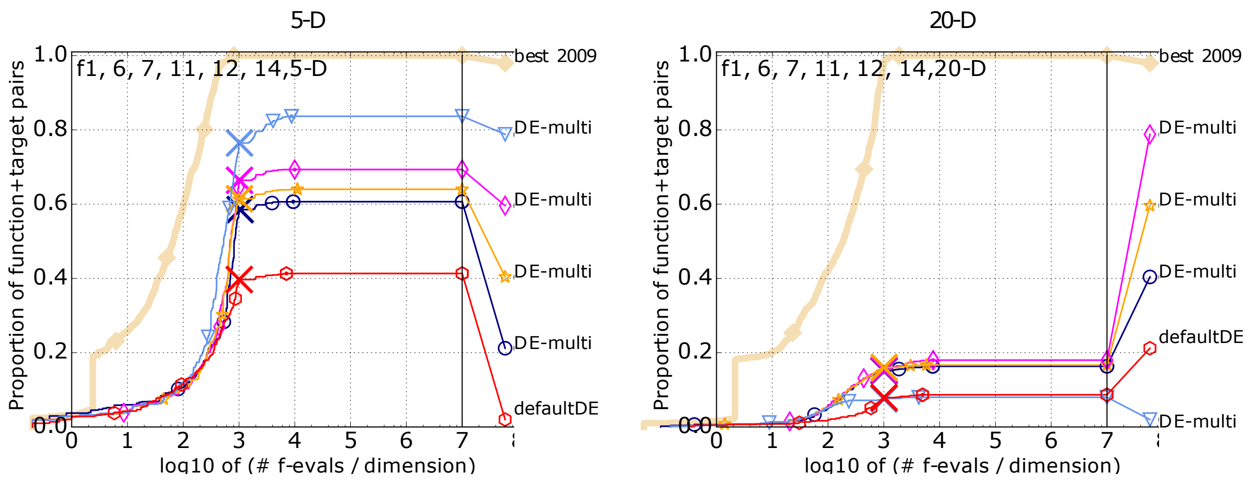


Figure 6.15: Bootstrapped empirical cumulative distribution of the number of objective function evaluations divided by dimension (FEvals/DIM) for 50 targets in $10^{[-8..2]}$ for all functions and subgroups in 5 and 20-D. The "best 2009" line corresponds to the best ERT observed during BBOB 2009 for each single target.

Chapter 7

Discussion

7.1 Optimization benchmark experiments

ERT

As for the ERT comparison the prominent success of the SMAC dataset by Hutter et al. [32] was probably ensured by better algorithm settings (number of restarts, number of principal components for the model etc.). The default SMAC version computed for this thesis usually copied the trend of needed evaluations, but one order of magnitude higher than the version in [32] which could easily mean e.g. 10 times more restarts or another parameter difference.

Most of the tuning procedures showed that it would suffice them to evaluate the objective function $1000D$ times before reaching the desired precision supposing dimension $D \leq 10$.

ERT cumulative distribution

Apart from the provided SMAC dataset dominance we can see rather well-performing SPOT procedure in the lower dimensional problems, but rather poor results in the 20-dimensional cases.

The slight decline in steepness after the evaluation cap (crosses in the figures) for the unimodal (separable, moderate, even the ill-conditioned) functions would suggest that more function evaluations were needed for better performance, especially in the high dimensional cases.

On the other hand the steep increase of the distribution function after reaching the maximum evaluations point for multimodal functions would suggest that more or earlier restarts would increase the overall performance.

Benchmark outcome

Three of the examined parameter tuners were benchmarked on the continuous noiseless function testbed and compared with one already existing benchmark. Although the tuners default configurations did not show any substantial optimizing capabilities and proving them to be rather basic level optimizers, comparison of several different methods was made that can improve the view on each of the investigated tuners.

7.2 Parameter tuning experiments

7.2.1 GA testing

GA unimodal test

The trained optimizer variants showed similar performance as the default variant on both training and testing testbeds. This is probably due to low number of optimized parameters.

Tests on the different class functions were in favor of the default configuration as expected. The unimodal-trained versions of the optimizer exhibited surprisingly good performance as well. This is probably also caused by the lower number of tuned parameters.

GA multimodal test

The multimodal-trained versions of GA demonstrated very similar performance as the default on all training, testing, and unimodal set of functions. This shows that GA trained on multimodal functions has decent performance for the unimodal testbed as well. The default variant was usually better in the 20D cases, as it is probably chosen to generalize well.

GA test outcome

GA parameter tuning did not show any gain against the default variant. It was probably optimized in too few parameters in too narrow domains.

7.2.2 DE testing

DE unimodal test

All unimodal-trained variants of the DE algorithm outperformed the default variant on training and testing set of functions. This shows that the parameters were well optimized to improve the algorithm performance. Even on the multimodal set the best trained variant outperformed the default one.

All the tests exhibit steepness decline after the point of maximum evaluations, which means that the performance would most likely further increase with increased number function evaluations as explained by Hansen et al. [28].

The higher-dimensional cases seem to be rather poorly performing for both default and optimized variants, meaning that DE is likely not good in generalizing over multiple dimensions.

DE multimodal test

Multimodal-trained variants of the DE algorithm outperformed the default variant on all training, testing and unimodal set of functions. This once again shows that the parameters were well optimized to improve the algorithm performance as well as the generalization of the multimodal-trained DE over the set of unimodal functions.

The tests exhibit steepness decline after the maximum evaluations point once again recommending higher number of function evaluations for better results [28]. Higher dimensional cases were once again optimized rather poorly.

DE test outcome

Parameter tuning improved the DE optimizer in all testing cases showing good optimizing enhancing abilities. Further research shows that higher number of evaluations would improve the performance even more. As the parameters outperformed default configurations on previously unseen cases, it would suggest that the DE parameters can be trained on specific set of functions.

7.3 Improvements

Possible improvements originating from this thesis would most likely include enhancements in:

- **SPOT**
  1. SPOT with random forest model 40D evaluation.
  2. Benchmark of SPOT's different model versions.
- **Spearmint**
  Successful benchmark of the Spearmint method, by further improving the current experiment design or by the use of the recent parameter configuration libraries that include spearmint interface such as Hyper-Parameter Optimization Library (HPOLib) [24, 12].
- **Mixed Continuous/Discrete benchmark functions**
  Benchmarking all the methods on mixed continuous-discrete benchmark functions. The only mixed functions the IRACE method optimized in this thesis were the GA and DE parameter spaces.
- **Parameter tuning extension**
  Tune the parameters of GA and DE by all the investigated parameter tuners.
- **Benchmarking extension - noise**
  Benchmark the tuners on the noisy function testbed.

Chapter 8

Conclusion

Brief overview of the current state of the art methods and methodologies in parameter tuning was given as a base to choose several comparable parameter tuning method representatives.

Three of the investigated parameter tuners were benchmarked and compared in continuous optimizers benchmarking framework [3] showing their strengths and weaknesses against other methods and frameworks that are already evaluated under same benchmark.

Two traditional optimizers (GA and DE) were chosen for parameter tuning by using the IRACE [36] procedure. Each of them was trained and tested on 2 different classes of functions. The GA did not exhibit any performance improvement after its parameter tuning. This was probably due to the lack of the optimized parameters and their restricted domain as well as the possibly well-tuned default parameter setting.

On the contrary the DE algorithm showed quite an improvement against its default settings showing the importance of parameter tuning as well as its (however limited) problem generalizing capabilities.

The results were presented in the form of graphs, generated by the COCO framework [3] that are consistently used within its research community providing clear view over the compared algorithm qualities and drawbacks.

For the results to be even sounder several improvement ideas were presented concerning the benchmark testing and parameter tuning completion and extension.

Bibliography

[1] 2015 IEEE Congress on Evolutionary Computation.
http://sites.ieee.org/cec2015/, state from 14. 5.,2017.

[2] Cerit scientific cloud. https://www.cerit-sc.cz.

[3] COmparing Continuous Optimisers: COCO.
http://coco.gforge.inria.fr/, state from 14. 5.,2017.

[4] COmparing Continuous Optimisers: Introduction.
http://coco.lri.fr/COCOdoc/introduction.html, state from 14. 5.,2017.

[5] Genetic and Evolutionary Computation Conference: GECCO.
http://www.sigevo.org/gecco-2009, state from 14. 5.,2017.

[6] Genetic and Evolutionary Computation Conference: GECCO.
http://www.sigevo.org/gecco-2010, state from 14. 5.,2017.

[7] Genetic and Evolutionary Computation Conference: GECCO.
http://www.sigevo.org/gecco-2012, state from 14. 5.,2017.

[8] Genetic and Evolutionary Computation Conference: GECCO.
http://www.sigevo.org/gecco-2013, state from 14. 5.,2017.

[9] Genetic and Evolutionary Computation Conference: GECCO.
http://www.sigevo.org/gecco-2015, state from 14. 5.,2017.

[10] Genetic and Evolutionary Computation Conference: GECCO.
http://gecco-2016.sigevo.org/index.html/HomePage, state from 14. 5.,2017.

[11] Genetic and Evolutionary Computation Conference: GECCO.
http://www.sigevo.org/gecco-2017, state from 14. 5.,2017.

[12] Hyperparameter Optimization Library.
http://www.automl.org/hpolib.html, state from 14. 5.,2017.

[13] National grid infrastructure metacentrum. https://www.metacentrum.cz/.

[14] R. P. Adams, M. Gelbart, J. Snoek, K. Swersky, and H. Larochelle. Spearmint bayesian optimization code base, 2013. https://github.com/HIPS/Spearmint.

[15] D. Ardia, K. Mullen, B. Peterson, J. Ulrich, and K. Boudt. Deoptim: Global optimization by differential evolution, 2016. `https://CRAN.R-project.org/package=DEoptim`.

[16] J. C. at al. The r project for statistical computing, 1997. `https://www.r-project.org/`.

[17] T. Bartz-Beielstein. SPOT: an R package for automatic and interactive tuning of optimization algorithms by sequential parameter optimization. *CoRR*, abs/1006.4645, 2010.

[18] T. Bartz-Beielstein, C. W. G. Lasarczyk, and M. Preuss. Sequential parameter optimization. In *2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 773–780 Vol.1, Sept 2005.

[19] T. Bartz-Beielstein and S. Markon. *Tuning Search Algorithms for Real-world Applications: A Regression Tree Based Approach*. 2004.

[20] T. Bartz-Beielstein, K. Parsopoulos, and M. Vrahatis. Analysis of Particle Swarm Optimization Using Computational Statistics. In Chalkis, editor, *Proceedings of the International Conference of Numerical Analysis and Applied Mathematics (ICNAAM 2004)*, pages 34–37, 2004.

[21] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[22] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. *F-Race and Iterated F-Race: An Overview*, pages 311–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[23] M. Chiarandini, C. Fawcett, and H. Hoos. A modular multiphase heuristic solver for post enrollment course timetabling. In *Proc. PATAT*, 2008.

[24] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, 2013.

[25] F.Dobslaw. Recent development in automatic parameter tuning for metaheuristics. In *WDS 2010 - Proceedings of Contributed Papers*, page 54–63, 2010.

[26] N. Hansen. *The CMA Evolution Strategy: A Comparing Review*, pages 75–102. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[27] N. Hansen, A. Auger, S. Finck, and R. Ros. Real-Parameter Black-Box Optimization Benchmarking 2010: Experimental Setup. Technical report, Centre de recherche INRIA Saclay – Île-de-France Parc Orsay Université, 2010.

[28] N. Hansen, A. Auger, R. Ros, S. Finck, and P. Pošík. Comparing results of 31 algorithms from the black-box optimization benchmarking bbob-2009. In *Proceedings of the 12th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '10, pages 1689–1696, New York, NY, USA, 2010. ACM.

[29] H. H. Hoos. *Automated Algorithm Configuration and Parameter Tuning*, pages 37–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[30] F. Hutter, D. Babic, H. H. Hoos, and A. J. Hu. Boosting verification by automatic tuning of decision procedures. In *Formal Methods in Computer Aided Design, 2007. FMCAD '07*, pages 27 –34, nov. 2007.

[31] F. Hutter, H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In A. Lodi, M. Milano, and P. Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6140 of *Lecture Notes in Computer Science*, pages 186–202. Springer Berlin Heidelberg, 2010.

[32] F. Hutter, H. Hoos, and K. Leyton-Brown. An evaluation of sequential model-based optimization for expensive blackbox functions. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '13 Companion, pages 1209–1216, New York, NY, USA, 2013. ACM.

[33] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION-5*, page 507–523, 2011.

[34] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, October 2009.

[35] Y. Lierler and P. Schüller. Parsing combinatory categorial grammar via planning in answer set programming. In E. Erdem, J. Lee, Y. Lierler, and D. Pearce, editors, *Correct Reasoning*, volume 7265 of *Lecture Notes in Computer Science*, pages 436–453. Springer Berlin Heidelberg, 2012.

[36] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari. The irace package, iterated race for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.

[37] O. Maron and A. W. Moore. The racing algorithm: Model selection for lazy learners. *Artif. Intell. Rev.*, 11(1-5):193–225, Feb. 1997.

[38] R. MERCER and J. SAMPSON. Adaptive search using a reproductive meta,Äźplan. *Kybernetes*, 7(3):215–228, 1978.

[39] V. Nannen and A. Eiben. A method for parameter calibration and relevance estimation in evolutionary algorithms. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 183–190, New York, NY, USA, 2006. ACM.

[40] V. Nannen and A. E. Eiben. Relevance estimation and value calibration of evolutionary algorithm parameters. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 975–980, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[41] M. E. H. Pedersen. *Tuning & Simplifying Heuristical Optimization.* PhD thesis, University of Southampton, Shool of Engineering Science, 2010.

[42] J. M. Peña, J. A. Lozano, and P. Larrañaga. *Estimation of Distribution Algorithms. A New Tool for Evolutionary Computation*, chapter Benefits of data clustering in multimodal function optimization via EDAs, pages 99–124. Kluwer Academic Publishers, Boston/Dordrecht/London, 2002.

[43] L. Scrucca. Ga: Genetic algorithms, 2016. `https://cran.r-project.org/package=GA`.

[44] S. K. Smit and A. E. Eiben. Comparing parameter tuning methods for evolutionary algorithms. In *2009 IEEE Congress on Evolutionary Computation*, pages 399–406, May 2009.

[45] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems 25*, 12/2012 2012.

[46] M. Vallati, C. Fawcett, A. Gerevini, H. Hoos, and A. Saetti. Generating fast domain-specific planners by automatically configuring a generic parameterised planner. 2011.

Appendix A

Technical details about examined frameworks

A.1 Implementation Language

ParamILS

Algorithm is implemented v language Ruby, which is script language similar to Perl, but it is object-oriented and cleaner. Not every person working with ParamILS needs to understand Ruby as the package also contains windows executable generated by RubyScript2Exe.

Spearmint

Spearmint is implemented in the Python language (version 2.7), using some plugins for numeric computation and stores the information in MongoDB database, which is NoSQL database with document-oriented data model.

IRACE

Irace package is implemented in R, which is environment meant for statistical data analysis and their graphical depiction, used mostly for academical purposes.

SPOT

SPOT is also implemented as R package, making it once again usable from every system with R shell or IDE.

SMAC

SMAC Framework is implemented in JAVA, compiled to be executable in Linux or some Linux-like variable environment for windows such as cygwin or bash for windows 10. The optimizing wrapper can be implemented in users language of choice, but it must be executable from the command line of the system and comply with the SMAC API (e.g. python, shell, java and others).

A.2 Algorithm usage

Spearmint

The spearmint usage is quite simple.

1. Start up a MongoDB daemon instance:

```
mongod --logpath <path_to_logfile> --dbpath <path_to_dbfolder>
```

2. Run spearmint main file:

```
python main.py <path_to_experiment_directory>
```

The results for each iteration are then printed to files the `out` directory, created in the `experiment_directory` as the algorithm starts.

SPOT

For all the computations with SPOT, function *spotOptim* was sufficient. It has optim-like interface common for many optimization functions in R. It has following signature:

```
spotOptim(par, fun, lower, upper, method, control)
```

where fun is function to be optimized, *lower* resp. *upper* are vectors of bounds on the parameter vector in variable *par* and *control* is a list of additional parameters for the optimizing algorithm such as maximum number of evaluations. SPOT is by design meant to tune noisy function solvers, to avoid that one can add following 3 lines to the *control* list of the *spotOptim* function:

```
spot.ocba=FALSE           #parameters for
seq.design.maxRepeats=1   #non-noisy
init.design.repeats=1     #functions
```

SPOT uses large variety of meta models to perform the parameter tuning such as random forests, multivariate regression spline, neural networks, treed Gaussian processes, prediction trees and others. One could easily explore only the possibilities of SPOT framework on COCO framework functions, but for the purposes of this work the default random forest option was selected.

IRACE

Irace framework has many different functions a usages, but for our purposes it sufficed only function *irace*, wrapped to the standard signature of optimizing function for R language:

```
optimize(params, optimized_function, lower_bounds, upper_bounds, max_eval)
```

ParamILS

User can run the ParamILS in the following way:

```
ParamILS -parameter0 value0 -scenariofile filename -parameterN valueN
```

Above usage is combined as it has standalone parameters as well as the scenario file name with the rest of algorithm configuration. Among the parameters there must be also the name of the executable optimizing wrapper which takes following input:

```
filename <instance> <instance_info> <cutoff_time>
         <cutoff_length> <seed> <params>
```

And prints out the following output:

```
Result for ParamILS: <solved>, <runtime>, <runlength>, <best_sol>, <seed>
```

Where the `<solved>` is either „SAT“, „UNSAT“ or „TIMEOUT“. The other parameters are numeric and self-explanatory.

SMAC

The usage is again same as in the previous case, with small differences in the names of the parameters. The program wrapper has again the same interface as in case of ParamILS, only instead of the last parameter with the name of file with parameters of the optimizing algorithm it also accepts couples

```
-parameter_name value
```

A.3 Parameter handling

SPOT

SPOT has many parameters to cover wide range of problems to represent and optimize. The parameters are traditionally specified in several files with different extensions. Files with extension *.roi* specify the region of interest for the tuned algorithm. There the user can specify algorithm parameters in the form:

```
<NAME> <low> <high> <TYPE>
```

The *low* and *high* parameters define the bounds for the parameter. The *type* parameter can be one of $\{INT, FLOAT, FACTOR\}$, where $FACTOR$ stands for categorical parameters, which are given in the form of an integer number which the algorithm (or its executing wrapper) must interpret itself, hence the obligatory *low* and *high* restrictions. Files with extension *.apd* specify the algorithm design, resp the parameters of tuning algorithm e.g. problem dimension, objective function, initial seed etc. Files with extension $.conf$ specify SPOT specific parameters, such as prediction model or initial design size (k).

Spearmint

Parameters

Spearmint has parameters stored inside *config.json* file which defines parameters for the tuning as well as the search/solving/optimizing algorithm. typical example of such file is following:

```
{
    "language": "PYTHON",
    "experiment-name": "Name",
    "main-file": "algorithm_executable.py",
    "max-finished-jobs": 200,
    "likelihood": "NOISELESS",
    "variables": {
        "var_name1": {
            "type": "ENUM",
            "options": [
                first, second
            ],
            "size": 1
        },
        "var_name2": {
            "max": 5,
            "min": -5,
            "type": "FLOAT",
            "size": 2
        }
    }
}
```

This configuration file would define one experiment with algorithm that uses 1 categorical parameter, 2 floating point parameters, is written in python, optimizes noiseless function and is permitted to run 200 times. Variable type can be also INT for integer parameters.

IRACE

Similarly as in previous examples, the parameters are specified as ordinal (o), categorical (c), integer (i) and real (r). The main function is called with following signature:

```
irace( tunerConfig = list(tuner_parameters),
       parameters = tuned_parameters)
```

And the result is a map of parameter names as keys and their values. Parameter tunerConfig is list parameters, which usually look similarly to this:

```
tunerConfig = list( hookRun = hook.run,  #irace inner function
                    instances = number_of_instances,
                    maxExperiments = number_of_evaluations,
                    logFile = log_filename)
```

SMAC

Basic parameters of SMAC framework are in fact same as in the case of ParamILS, with little less number of performance metrics and unspecified order of the parameters and their types. Remarkable change against ParamILS is the possibility to specify the numeric parameters as an interval or union of intervals, which enables optimization of parameters with continuous domain. The parameter definition is following:

```
<name> <type> {list_of_values} [initial_value]
```

The only different thing is the optional argument for initial/default value. The numeric parameters can be also defined in this way:

```
<name> <integer/real> [minimum, maximum] [initial_value]
```

So it suffices to determine closed interval on which all the possible values are defined.

ParamILS

ParamILS gets various parameters which can be passed from command line or preferably specified in one configuration file which name can be then passed as the only argument. ParmaILS can also take combination of CLI and configuration file. Apart from the configuration file parameter, user can specify parameters such as the problem specific folder, path to the optimizing executable, performance metric to consider, timeouts, algorithm variant (basic/ focused), number of iterations, number of evaluations, logging file path, instance files or reference solution files. Apart from ParamILS parameters there are also parameters for the optimizing algorithm wrapper, which must comply with ParamILS specific API. These parameters include optimizing algorithm name, random number generator seed, optimization timeouts and optimizing algorithm parameter configuration file. This file has 3 parts, first the basic parameters are specified (enough for most of the cases), user then can specify conditional dependencies and lastly the forbidden combinations of parameter values. Basic parameters can be numeric, ordinal (strings), and categorical (e.g. used heuristics, can be specified as ordinal). Every parameter has following representation:

```
<name> <type> {list_of_values}
```

This type of representation is very unfortunate because it does not support continuous optimization as all the possible floating point numeric parameters must be specified. This is a major issue for aforementioned COCO framework, because all the parameters of the optimized functions are floating point numbers in the span of $[-5, 5]$ which makes it very difficult to accomplish the desired accuracy of 10^{-8}.

Appendix B

List of used abbreviations

5D Five-Dimensional

20D Twenty-Dimensional

BBOB Black-Box Optimization Benchmarking

CART Classification And Regression Tree

COCO Comparing Continuous Optimizers

CPU Central Processing Unit

DE Differential Evolution

EDA Estimation of Distribution Algorithm

ERT Expected Run Time

GECCO Genetic and Evolutionary Computation Conference

GA Genetic Algorithm

HPOLib Hyper-Parameter Optimization Library

IRACE Iterated Racing for Automatic Algorithm Configuration extension

OS Operating system

RAM Random access memory

SMAC Sequential Model-based Algorithm Configuration

SPO Sequential Parameter Optimization

SPOT Sequential Parameter Optimization Toolbox

Appendix C

CD Content

- **COCO:** directory with COCO framework project and optimization experiment comparison
- **COCO/4comparison:** directory with 4 algorithm datasets comparison graphs and templates
- **SPOTEvaluation:** directory with SPOT benchmark and SPOT running scripts
- **IraceEvaluation:** directory with IRACE benchmark and IRACE running script
- **Parameter Tuning:** directory with DE and GA parameter tuning tests, also with data generating and processing script
- **SMAC:** directory with SMAC benchmark dataset, SMAC dataset by Hutter at al. [32] and SMAC code-base
- **Spearmint:** directory with examined spearmint project
- **thesis:** directory with the thesis together with its latex source codes