ECOLE
POLYTECHNIQUE
DE BRUXELLES

UNIVERSITÉ LIBRE DE BRUXELLES
Ecole Polytechnique de Bruxelles
IRIDIA - Institut de Recherches Interdisciplinaires
et de Développements en Intelligence Artificielle

Automated Algorithm Configuration for Hard Optimization Problems

Zhi YUAN

Ph.D. Thesis

Promoteur de Thèse:
Prof. **Thomas STÜTZLE**

Co-Promoteur de Thèse:
Prof. **Mauro BIRATTARI**

Thèse présentée en vue de l'obtention du titre de
Docteur en Sciences de l'Ingénieur

Année académique 2018-2019

Abstract

Algorithms for solving hard optimization problems usually have a number of parameters that greatly influence the algorithm performance. Instead of manual or ad hoc methods to adjust the parameters, using automatic tools for configuring such parameters have been proved to be crucial for deriving high-performing algorithms. Automatic algorithm configuration can be regarded as a black-box mixed-discrete-continuous variable optimization problem. Solving such problem usually requires two components for the two subtasks: a black-box search method that generates candidate configurations, and an evaluation method that evaluate the quality of the generated configuration under stochasticity.

In this work, we define two frameworks, namely, iterated selection and post-selection, for combining search method and evaluation method for algorithm configuration. Through extensive literature review, all the established algorithm configurators can be identified to be under the iterated selection framework. We have extended an established evaluation method, the statistical racing method, with an ad-hoc iterated search method for configuring categorical and conditional parameters, as well as combined the racing method with established black-box search methods, such as MADS and CMA-ES, following the iterated selection framework. We have introduced state-of-the-art black-box optimizers such as CMA-ES, BOBYQA, and Nelder-Mead simplex to solving the problem of automatic algorithm configuration, and demonstrated how state-of-the-art configurator can be obtained by hybridizing these black-box search method with statistical racing method following the post-selection framework. The best settings for devising post-selection configurators are empirically analyzed in depth. We have identified and proposed a decreasing population CMA-ES with post-selection as the state-of-the-art algorithm configurator across extensive benchmark configuration problems with various dimensions and configuration budget.

Acknowledgement

This dissertation results from the work mainly during my stay at the research lab IRIDIA in Université Libre de Bruxelles from November 2007 to September 2012. During the almost five years at IRIDIA, I have the honor to have worked with some of the most brilliant minds, who have great influence on me in both research works as well as the way how I see the world. It goes without saying that this work is the combined influence from all the people that I interacted with.

Firstly, I sincerely thank my supervisor, Prof. Thomas Stützle. Thomas is deeply involved in all my research works. Without Thomas, many of the ideas that are presented in this dissertation wouldn't have been realized. His charisma and rigorous attitude towards scientific research has shaped me in all my research works and beyond. I could never learn enough from his immense knowledge and expertise in the design and development of optimization algorithms. Working with him is always enjoyable experience. I can always rely on him with his judgement and taste on the research work. His constant support and encouragement were crucial for the completion of my Ph.D. work, especially during the hard times. I wish to continue to work with him also in the future. I would also like to thank Prof. Mauro Birattari, my co-supervisor and mentor. Mauro is a great scientist in many research directions. Among his many important scientific contributions, his work on automatic configuration has laid the foundation that my PhD work is based upon. His passion, rigor, and creativity have shaped my work in many important ways. Mauro has constantly provided reliable critics to my work, which usually provided me with different perspective of looking into the research topic, and have always led to significant improvement of my work. There is no exception that any work he is involved is high quality work. I wish I could learn even more from him. I am also grateful to have Prof. Marco Dorigo to be my scientific advisor. Thanks to Marco, I was able to receive financial funding through various research projects, which made the research work presented here possible. He has also supported me in the applications of research fellowships, which have always turned out successful. Marco

also constantly provided comments to my work and career. His comments are usually brief, but to the point and very helpful. I also want to thank Prof. Hugues Bersini, the co-director of the IRIDIA together with Marco, for making IRIDIA an enjoyable workplace. My gratitude also goes to Prof. Marie-Éléonore Kessaci from Université de Lille, Prof. Patrick de Causmaecker from Katholieke Universiteit Leuven, for their useful comments that helped improve the quality of this dissertation.

Special thanks go to all my fellow IRIDIAns. All of them, Christos Ampatzis, Doğan Aydın, Prasanna Balaprakash, Hugues Bersini, Leonardo Bezerra, Stefano Benedettini, Navneet Bhalla, Saifullah bin Hussin, Mauro Birattari, Matteo Borrotti, Manuele Brambilla, Arne Brutschy, Alexandre Campo, Sara Ceschia, Marco Chiarandini, Anders Christensen, Muriel Decreton, Antal Decugnière, Giacomo Di Tollo, Jérémie Dubois-Lacoste, Stefan Eppe, Eliseo Ferrante, Gianpiero Francesca, Marco Frison, Matteo Gagliolo, Lorenzo Garattoni, Álvaro Gutiérrez-Martín, Dhananjay Ipparthi, Takashi Kawakami, Marie-Éléonore Kessaci, Stefanie Kritzinger, Renaud Lenne, Xiaomin Liang, Tianjun Liao, Manuel López-Ibáñez, Max Manfrin, Amin Mantrach, Bruno Marchal, Xavier Martínez-González, Franco Mascia, Nithin Mathews, Michael Maur, Roman Miletitch, Marco A. Montes de Oca, Daniel Molina, Prospero C. Naval Jr., Shervin Nouyan, Rehan O'Grady, Sabrina Oliveira, Michele Pace, Paola Pellegrini, Leslie Pérez Cáceres, Carlo Pinciroli, Giovanni Pini, Carlotta Piscopo, Gaëtan Podevijn, Andreagiovanni Reina, Andrea Roli, Fabio Rossi, Francesco Sambo, Francisco C. Santos, Alexander Scheidler, Touraj Soleymani, Paolo Spanevello, Alessandro Stranieri, Wenjie Sun, Cristina Teixeira, Vito Trianni, Roberto Tavares, Elio Tuci, Ali Emre Turgut, Colin Twomey, Thijs Urlings, Gabriele Valentini, David Weiss helped me in different ways throughout these years. Though I know this is definitely an inexhaustive list, by this list my gratitude extends to all IRIDIAns and friends that I have met in Brussels. IRIDIAns are more than colleagues to me, they are friends and even an extended family to me during my time in Brussels. They have made IRIDIA become more than a productive workspace: besides high productivity, it is at the same time an everyday playground for us. I appreciate the fun that we have had, and these experiences are always engraved in my deepest memories.

I acknowledge financial support from the following projects: COMP^{2}SYS, a Marie Curie Early Stage Research Training Site funded by the European Community's Sixth Framework Programme; the ANTS project, an *Action de Recherche Concertée* funded by the Scientific Research Directorate of the French Community of Belgium; and the

META-X project, an *Action de Recherche Concertée* funded by the Scientific Research Directorate of the French Community of Belgium. I also acknowledge support from the Belgian F.R.S.-FNRS, where I was funded through an Aspirant fellowship and a FRIA fellowship.

I couldn't thank my beloved wife, Jianqun Huang, enough, by dedicating this work solely to her. She has been with me through countless nights and weekends when I spent on the seemingly never-ending work. I am extremely grateful to her support and encouragement, and above all, to her love for me. I am also deeply indebted to my beloved parents, Lihui Yuan and Qun Xiao, for supporting my choice of life path so far away from home. I also thank all my dear friends for all their invaluable supports during these PhD years.

Zhi Yuan
October 12th, 2019
Brussels, Belgium

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Optimization is about finding the best from a set of possible choices. Life is full of optimization problems, such as picking a coin with largest value from different coins, walking to the Delirium bar[1] from home using the shortest path, buying the most tasty beer from its over 2000 available beer brands. To define the optimization problems, one needs to specify what the possible choices are, i.e., the *domain* or *solution space* such as all available coins, all possible paths from home to Delirium, and all the 2000 beer brands to offer. In addition, one needs to define how good each choice is by a so-called *objective function*, such as the value of the coin, the walking distance measure, and a personal rating of beer taste. Some optimization problem can be solved by human in the blink of an eye, like coin picking. Some optimization problem will take us quite some effort, like finding the shortest path on a map, which will be best left to be solved by a computer with a computational method, an *algorithm*.

Different classes of optimization problems pose different levels of computational difficulty, or *complexity* [88], to the present-day computers. The shortest path problem mentioned above belongs to an "easier" complexity class called $\mathcal{P}$, which can be solved in polynomial-time on a deterministic machine by, e.g., the famous Dijkstra algorithm [65]. Imagine your friend comes to visit Brussels: besides the Delirium bar also with a list of other 50 points of interest to visit, including touristic attractions such as Grand Place, Manneken Pis, Atomium, and so on. Finding a shortest tour that visits all these places of interest, also known as the *traveling salesman problem* (TSP), belongs to the more complex class of $\mathcal{NP}$-hard problems, for which no polynomial-time de-

[1] A famous bar in the city center of Brussels, Belgium. Delirium was recorded as having 2004 beer brands in the year 2004 in the Guinness Book of Records: `https://en.wikipedia.org/wiki/Delirium_Caf%C3%A9`

terministic algorithm is known.[2] For $\mathcal{NP}$-hard problems, the best exact algorithms known will take in the worst case exponential time with respect to problem size. These problems will quickly become intractable for the exact algorithms as problem size grows. Algorithms with state-of-the-art empirical performance for such computationally hard problems are usually highly heuristic. Similarly for the computationally hard continuous optimization problems where the functions are multimodal, non-separable, and high-dimensional, classic algorithms such as gradient descent and Newton's method usually fail to find optimal or near-optimal solutions, and heuristic algorithms are usually preferred. In this thesis, we focus on such computationally hard problems, and mainly on the heuristic algorithms for solving them.

Any state-of-the-art algorithms for computationally hard optimization problems, be it exact algorithms such as branch-and-cut solvers, or various heuristic algorithms such as tabu search, simulated annealing, and ant colony optimization, have a number of parameters that strongly influence the algorithm behavior and performance. For example, the CPLEX 12.6[3] solver has a total of 159 parameters [120], out of which 72 parameters influence the solver's search mechanism and can be configured by the user. How such parameters should be set to optimize performance is a challenge for algorithm designers as well as for algorithm users. Conventionally, the algorithm parameters are either set based on rules of thumb, or tuned manually by trial and error. It is even mentioned in the literature that when designing and testing a new heuristic algorithm, around 10% of the time is spent on the algorithm development, and 90% of the time is spent at fine-tuning its parameters [2].

Finding the optimal parameter setting of a target optimization algorithm automatically is itself a hard optimization problem [32]. The solution space is the parameter space of the target algorithm, which could contain a mix of variables such as real-valued parameters, e.g., the perturbation parameter of the Simplex algorithm, or the cooling rate of the simulated annealing algorithm; integer-valued parameters, e.g., the heuristic frequency of MIP solvers, or the population size of population-based algorithms; and categorical parameters, e.g., the branching strategy or cuts aggressivity of MIP solvers, or types of neighborhoods to choose for local search. The objective function, i.e., the quality of a parameter setting with respect to a class of target optimization problem, is a *stochastic black box*. Evaluating the quality of a parameter setting

[2] $\mathcal{NP}$-hard problem is polynomial-time solvable if and only if $\mathcal{NP} = \mathcal{P}$, which is currently still the most prominent open problem in theory of computation.

[3] https://www.ibm.com/bs-en/marketplace/ibm-ilog-cplex

consists in running the target algorithm with that setting on a stream of target problem instances, and measuring the quality of the obtained solutions. Such evaluation is a *black box*, and the objective function can only be queried one value at a time. No analytic expression is available to describe the objective function, thus, no gradient information and no curvature of the function can be obtained. Such evaluation is also *stochastic* due to two possible sources of stochasticity: firstly, the target optimization algorithm might be stochastic; secondly, even if the target algorithm is deterministic, the quality of a parameter setting is still stochastic, as its measure is different on different target problem instance.

To approach the automatic algorithm configuration problem, two separate but interconnected aspects need to be considered: the *search* aspect that generates high-quality candidate configurations, and the *evaluation* aspect that evaluates the candidate configurations under stochasticity and selects the best to bias the search. In the literature on automatic algorithm configuration, most of the research efforts are mainly focused on the search aspect, including fractional factorial design [2], Kriging model-based optimization (SPO) [17], mesh adaptive direct search (MADS) [8], local search [118], or genetic algorithms [5]. In these works, only little efforts are investigated on the evaluation aspect. Some other research efforts are focused on the evaluation aspect, such as statistical racing [36], which is hybridized with an iterative search mechanism for numerical parameters [14]. SPO+ [117] and SMAC [112] also significantly extends SPO on the evaluation aspect.

1.1 Objective and methodology

The main objective of the work presented in this thesis is to design and develop high-performing tools (also called *configurators*) for automatic algorithm configuration.

We first focused on studying existing algorithm configurators, and classified them into the framework of *iterated selection*. Configurators that are based on iterated selection iteratively apply a search method to generate candidate configurations, compare them with the best-so-far (or *incumbent*) configuration using an evaluation method, and select the best configurations to bias the search in the next iterations. Under the iterated selection framework, we tried to extend existing evaluation method (statistical racing) with an iterative search method, and extend existing search method with an effective evaluation method. We further surveyed the literature on black-box optimization, and were the first to introduce state-of-the-art black-box optimizer as search method

for algorithm configuration, including CMA-ES, BOBYQA and Nelder-Mead Simplex. In order for such black-box optimizers to perform effectively on algorithm configuration, we have studied and proposed the *post-selection* framework, which differs from iterated selection in preserving the incumbent configuration. Instead of comparing and identifying the incumbent from iteration to iteration, it divides the configuration procedure into two phases: a first *elite qualification* phase, where the a set of good configurations are qualified and archived. Then in a second *elite selection* phase, these elite configuration will be carefully compared and the best will be selected by an evaluation method. Such post-selection mechanism is especially effective for the model-based, black-box optimizer such as BOBYQA, which uses all generated configuration points to build a global model to approximate the search space; and for the evolutionary search method that does not include best-so-far solution in each iteration, such as CMA-ES.

1.2 Main contributions

The main contributions of this thesis are the following:

- We classified the existing algorithm configurators into an iterated selection framework, which combines a search method and an evaluation method in an iterative manner. In each iteration, a search method generates candidate configurations, and then an evaluation method is applied to select best ones from the candidate configurations to bias the search of the further iterations.
- We extended the statistical racing method with an effective iterated search method for categorical and conditional parameters, and improved the performance of the first version of iterated racing for numerical parameters. The extended version of iterated racing is the first version of a fully functioning iterated racing configurator, and it is currently one of the most widely used automatic algorithm configuration tools, also known as `irace`.
- We extended the family of iterated racing by hybridizing the statistical racing method with established search methods for continuous optimization, such as MADS and CMA-ES, by following the iterated selection framework.
- We adopted a number of state-of-the-art black-box optimizers, such as BOBYQA, CMA-ES, MADS and Nelder-Mead Simplex to configure numerical parameters. In order to achieve better performance of these continuous optimizers, we proposed simple

post-selection mechanisms to handle the stochastic nature of algorithm configuration problem. Post-selection is an alternative mechanism to iterated selection. It divides the algorithm configuration procedure into two phases: a first elite qualification phase that iteratively identifies a set of elite configurations which are then compared and selected carefully in the second elite selection phase. We analyzed the search performance of such continuous optimizers in the algorithm parameter landscape.

- We performed an in-depth analysis of the post-selection framework for automatic algorithm configuration, and studied the best settings for devising high-performing algorithm configurators. We have proposed a post-selection CMA-ES configurator with decreasing population, which outperforms state-of-the-art configurators such as iterated racing and SMAC.

Further side contributions include:

- Applying automatic algorithm configuration to various application optimization problems, such as configuring iterated greedy algorithms for a real-world train scheduling problem, optimizing operator selection strategies in memetic algorithms, and configuring CPLEX for planning flight trajectories.
- Studying various algorithm configuration scenarios, such as configuration for large instances, an analysis of offline configuration versus online adaptation.

1.3 Publications

A number of publications have been produced during the development of the research work presented in this thesis. Many of these publications, especially the ones that contribute to the core part of this thesis, have been written in collaboration with colleagues under the supervision of Dr. Thomas Stützle.

These publications are grouped into three categories: the main contributed publications to the core work for this PhD thesis, publications related to automatic algorithm configuration, and additional contributions on miscellaneous research topics.

1.3.1 Main contributed publications for the thesis

The main contributed publications are on the development of automatic algorithm configuration tools.

- M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. Automated Algorithm Tuning using F-Races: Recent Developments. In M. Caserta and S. Voß, editors, *Proceedings of MIC 2009, the 8th Metaheuristics International Conference*, page 10 pages, Hamburg, Germany, 2009.

- M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. F-Race and iterated F-Race: An overview. In T. Bartz-Beielstein et al., editors, *Experimental Methods for the Analysis of Optimization Algorithms*, Natural Computation Series, pages 311–336. Springer Verlag, Berlin, Germany, 2010.

- Z. Yuan, T. Stützle, and M. Birattari. MADS/F-Race: mesh adaptive direct search meets F-Race. In M. Ali et al., editors, *Proceedings of IEA-AIE 2010*, volume 6096 of *Lecture Notes in Artificial Intelligence*, pages 41–50. Springer Verlag, Berlin, Germany, 2010.

- Z. Yuan, M. Montes de Oca, M. Birattari, and T. Stützle. Modern continuous optimization algorithms for tuning real and integer algorithm parameters. In M. Dorigo et al., editors, *Proceedings of ANTS 2010, the Seventh International Conference on Swarm Intelligence*, volume 6234 of *Lecture Notes in Computer Science*, pages 204–215. Springer Verlag, Berlin, Germany, 2010.

- Z. Yuan, M. Montes de Oca, M. Birattari, and T. Stützle. Continuous optimization algorithms for tuning real and integer parameters of swarm intelligence algorithms. *Swarm Intelligence*, 6(1):49–75, 2012.

- Z. Yuan, T. Stützle, M. A Montes de Oca, H. C. Lau, and M. Birattari. An analysis of post-selection in automatic configuration. In *Proceedings of GECCO*, pages 1557–1564. ACM, 2013.

1.3.2 Related contributions on automatic algorithm configuration

The following publications are related to the main topic of this thesis, automatic algorithm configuration. Their topics are mainly on applying the developed algorithm configuration tools in this thesis to configuring high-performing algorithms to various application optimization problems, and studying various algorithm configuration scenarios.

- Z. Yuan, A. Fügenschuh, H. Homfeld, P. Balaprakash, T. Stützle, and M. Schoch. Iterated greedy algorithms for a real-world cyclic

train scheduling problem. In M. J. Blesa et al., editors, *Hybrid Metaheuristics, 5th International Workshop, HM 2008*, volume 5296 of *Lecture Notes in Computer Science*, pages 102–116. Springer Verlag, Berlin, Germany, 2008.

- Lindawati, Z. Yuan, H. C. Lau, and F. Zhu. Automated parameter tuning framework for heterogeneous and large instances: Case study in quadratic assignment problem. In G. Nicosia and P. Pardalos, editors, *Proc. of LION7*, volume 7997 of *Lecture Notes in Computer Science*, pages 423–437. Springer-Verlag, Berlin, Germany, 2013.
- Z. Yuan, S. D. Handoko, D. T. Nguyen, and H. C. Lau. An empirical study of off-line configuration and on-line adaptation in operator selection. In P. M. Pardalos et al., editors, *Proceeding of Learning and Intelligent OptimizatioN (LION8)*, volume 8426 of *Lecture Notes in Computer Science*, pages 62–76. Springer International Publishing, 2014.
- Z. Yuan. Automatic Configuration of MIP Solver: Case Study in Vertical Flight Planning. In *Matheuristics 2016*, pages 48–59, 2016.

1.3.3 Additional contributions

During my PhD studies, I have been involved in a number of side projects that often also led to publications. In the following I mention the publications in these additional research efforts that are, however, not necessarily strictly contributing to the core contents of this thesis.

- A. Fügenschuh, H. Homfeld, A. Huck, A. Martin, and Z. Yuan. Scheduling locomotives and car transfers in freight transport. *Transportation Science*, 42(4):478–491, 2008.
- P. Balaprakash, M. Birattari, T. Stützle, Z. Yuan, and M. Dorigo. Estimation-based ant colony optimization and local search for the probabilistic traveling salesman problem. *Swarm Intelligence*, 3(3):223–242, 2009.
- P. Varakantham, H. C. Lau, and Z. Yuan. Scalable randomized patrolling for securing rapid transit networks. In *Proc. Innovative Applications for Artificial Intelligence (IAAI)*, pages 1563–1568. AAAI Press, 2013.
- A. Gunawan, Z. Yuan, and H. C. Lau. A mathematical model and metaheuristics for time dependent orienteering problem. In *Proc. of PATAT*, pages 202–217, 2014.

- H. C. Lau, Z. Yuan, and A. Gunawan. Patrol scheduling in urban rail network. *Annals of Operations Research*, 239(1):317–342, 2014.
- Z. Yuan, A. Fügenschuh, A. Kaier, and S. Schlobach. Variable speed in vertical flight planning. In *Operations Research Proceedings*, pages 635–641. Springer, 2014.
- Z. Yuan and A. Fügenschuh. Home health care scheduling: A case study. In G. Kendall et al., editors, *Proceedings of MISTA*, pages 555–569, August 2015.
- L. Amaya Moreno, Z. Yuan, A. Fügenschuh, A. Kaier, and S. Schlobach. Combining NLP and MILP in Vertical Flight Planning. In *Operations Research Proceedings*, pages 273–278. Springer, 2015.
- Z. Yuan, L. Amaya Moreno, A. Fügenschuh, A. Kaier, and S. Schlobach. Discrete speed in vertical flight planning. In F. Corman et al., editors, *Proceedings of International Conference on Computational Logistics*, volume 9335 of *Lecture Notes in Computer Science*, pages 734–749. Springer, 2015.
- Z. Yuan, L. Amaya Moreno, A. Maolaaisha, A. Fügenschuh, A. Kaier, and S. Schlobach. Mixed integer second-order cone programming for the horizontal and vertical free-flight planning problem. Technical report, AMOS#21, Applied Mathematical Optimization Series, Helmut Schmidt University, Hamburg, Germany, 2015. Submitted to journal.
- Z. Yuan and É. Taillard. Dynamic programming for vertical flight planning. Technical Report TR/IRIDIA/2016-006, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, May 2016.
- Z. Yuan. A brief literature review on ship management in maritime transportation. Technical Report TR/IRIDIA/2016-001, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, February 2016.

1.4 Structure of the thesis

This thesis consists of seven chapters and four appendices.

In Chapter 2, we provide relevant background information for the rest of the thesis, including an introduction on various optimization problems, various optimization algorithms, the definition of the algorithm configuration problem, and types of algorithm parameters to be configured.

In Chapter 3, we present a literature survey on configuration algorithms from our point of view, where we decompose the existing configuration algorithms into a combination of search method and evaluation method, and different ways to combine them such as iterated selection and post-selection.

Chapter 4 presents our iterated selection configurators using statistical racing as evaluation method, including the extension of the iterated racing method to configure categorical and conditional parameters, as well as to improve its search capability for numerical parameters. We also extend a mesh adaptive direct search configurator by hybridizing it with a statistical racing as evaluation method in the iterated selection context.

Chapter 5 introduces the state-of-the-art black-box continuous optimizers such as BOBYQA and CMA-ES as search method for configuring numerical algorithm parameters, such as real-valued and integer-valued parameters. We extend these black-box optimizers to handle the stochasticity in the algorithm configuration problem, and compare them with the established search method used in the existing configurators.

Chapter 6 provides an in-depth analysis of the post-selection framework for algorithm configuration. We studied and extensively experimented various post-selection setting for handling the stochasticity in the algorithm configuration problem. We compare our post-selection mechanism with the state-of-the-art configurators for configuring mainly numerical parameters, as well as for configuring categorical parameters.

Chapter 7 summarizes and concludes the thesis.

Within the four appendix chapters, Appendix A applies the automatic algorithm configuration tool developed in this thesis to tune numerical parameters of iterated greedy and iterated ants algorithm for a real-world train scheduling problem. Appendix B provides a proof-of-concept feasibility study on configuring algorithms for large instances by configuring on small instances and scaling the tuned configuration to large instances. Appendix C studies offline configuration versus online parameter adaptation and various ways to combine them. Appendix D applies configurator to configure over 70 categorical parameters of a commercial MIP solver CPLEX for a real-world flight planning problem.

Chapter 2

Background

In this chapter, we explain some basic concepts about optimization problems and optimization algorithms. Then the automatic algorithm configuration problem, the problem of configuring a target optimization algorithm for a target optimization problem, is formally defined, and the types of algorithm parameters to be configured are introduced.

2.1 Optimization

Optimization problems concern finding the *best* solution in a solution space with respect to a certain objective function. Without loss of generality, an optimization problem can be defined as

$$\text{minimize}^1 \quad f(x) \mid x \in X^n, \tag{2.1}$$

where X^n is the set of feasible solutions, or *domain*, and $f : X^n \to \mathbb{R}$ is the *objective function* that maps a feasible solution $x \in X^n$ to an objective value in $\mathbb{R}$. A solution $x \in X^n$ is an n-dimensional vector that is composed of n *variables*, i.e., $x := \{x_1, x_2, \ldots, x_n\}$. Optimization problems can roughly be classified into two different categories depending on the type of the variables in the domain X^n: discrete optimization problems, where the domain consists of a finite set of n discrete variables, and continuous optimization problems, where the domain comprises a subset of n real-valued variables. n is usually referred to as the *size* of an optimization problem.

2.1.1 Discrete optimization problems

An optimization problem defined by equation 2.7 is called discrete optimization problem, if some or all of the variables take values from a

[1]Note that this definition is for minimization problem. A maximization problem can be equivalently formulated as $-\min(-f(x))$ without loss of generality.

discrete set, such as integers.

Many discrete optimization problems arise from problems on graphs or other discrete structures. Such discrete optimizations are also referred to as combinatorial optimization problems. Solutions to combinatorial optimization problems are usually combinatorial structures such as combinations, permutations, sequences or routes. In the following, two example combinatorial optimization problems are given, which will be approached in this thesis.

Traveling salesman problem

The traveling salesman problem (TSP) [58, 140, 193, 6] is one of the most prominent combinatorial optimization problems. It can be informally described as finding a minimum distance tour for a salesman that visits each city exactly once and returns to his starting city. In the TSP, a graph $G := (N, A)$ is given, where $N := \{1, 2, \ldots, n\}$ is the set of nodes and A is the set of arcs that fully connects the nodes. Each arc $(i, j) \in A$ that connects two nodes $i, j \in N$ is associated with a cost $d_{i,j}$ for traversing it. A solution of the TSP can be represented as a permutation π of the nodes in N, $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$, where $\pi(i)$ is the node index at position i. The salesman can follow the so-called Hamiltonian cycle in the sequence π and then return to $\pi(1)$. The goal of the TSP is to find a Hamiltonian cycle such that its cost defined as

$$d_{\pi(n),\pi(1)} + \sum_{i=1}^{n-1} d_{i,\pi(i)}, \tag{2.2}$$

i.e., the sum of the costs of the arcs in the Hamiltonian cycle, is minimized.

Quadratic assignment problem

The quadratic assignment problem (QAP) [136, 199, 183, 41] is another prominent combinatorial optimization problem. It is the abstract model of many real-life facility layout problems, including the layout design of campuses, hospitals and typewriter keyboards. In the QAP, are given n interrelated facilities, each of which is to be assigned to one of n locations. The distance between each pair of locations i and j is given by d_{ij}; and the flow (measuring relatedness) between each pair of facilities k and l is given by f_{kl}. A solution of the QAP can be represented by a permutation vector π, such that $\pi(i) = k$ assigns to a location i a unique facility k. The objective is to find a permutation π

that minimizes the total cost defined as:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij} \cdot f_{\pi(i)\pi(j)}, \tag{2.3}$$

that is, the total cost is to sum up for each pair of locations i and j that are assigned with facilities $\pi(i)$ and $\pi(j)$, the distance d_{ij} multiplied by the flow $f_{\pi(i)\pi(j)}$. A good QAP solution can be heuristically understood as an assignment of highly related facilities to nearby locations.

2.1.2 Continuous optimization problems

Continuous optimization problems differ from discrete ones by the domain of the variables. The variables of continuous optimization problems usually take values from a subset of the real numbers. The standard version of an unconstrained continuous optimization problem can be stated as

$$\min f(x) \quad \text{where } x \in X^n \subseteq \mathbb{R}^n \tag{2.4}$$

In the following, two continuous functions are given as examples, the sphere function and the Rastrigin function.

Sphere function

The sphere function of dimension n is defined as

$$\min \quad \sum_{i=1}^{n} x_i^2. \tag{2.5}$$

It contains only one local optimum when $x_i = 0$ for all $i = 1, 2, \ldots, n$. Therefore, the sphere function is an example of a *unimodal* function, which are functions that contain only a single local optimum, hence, this is also the global optimum. Besides, the sphere function is also an example of a *separable* function, i.e., for optimizing $f(x)$, it suffices to optimize for each variable $x_i, i = 1, 2, \ldots, n$, independently.

It is common in practice to impose a bound constraint with bound b_i to each variable x_i, such that

$$-b_i \leq x_i \leq b_i, \ i = 1, 2, \ldots, n. \tag{2.6}$$

An illustration of the sphere function of dimension $n = 2$ and bound $b_1 = b_2 = 4$ is given in Figure 2.1.

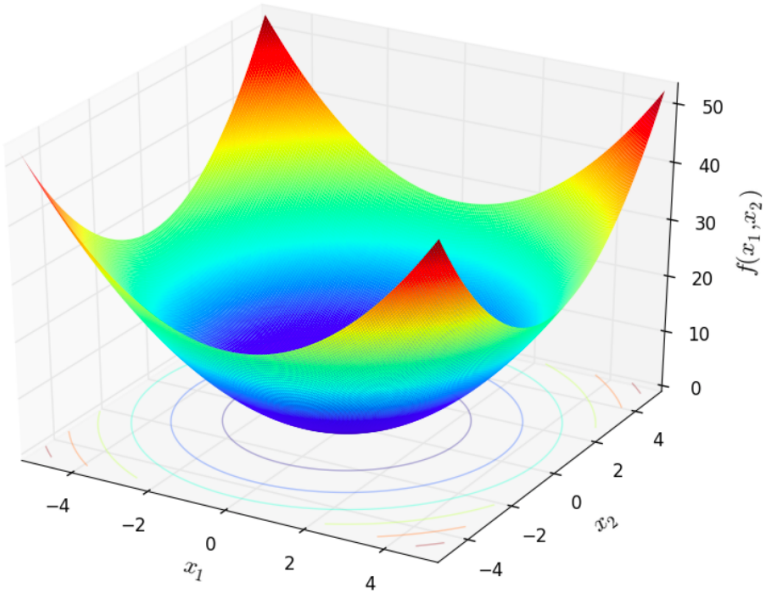


Figure 2.1: Sphere function of dimension two with bound constraint $-4 \leq x_1, x_2 \leq 4$.

Rastrigin function

The family of Rastrigin functions of dimension n can be stated as

$$\min \quad nA + \sum_{i=1}^{n} (x_i^2 - A\cos(2\pi x_i)). \tag{2.7}$$

The so-called amplitude parameter A can be adjusted to obtain different landscapes. An illustration of the Rastrigin function of dimension $n = 2$, amplitude $A = 10$, and bound $[-5.12, 5.12]^2$ is given in Figure 2.2. It clearly shows that there exist many local optima. Therefore, the Rastrigin function is an example of a *multimodal* function, which are functions that contain multiple local optima. It is clearly shown from Figure 2.2, that optimizing for each variable independently does not lead to the global optimum. Therefore, Rastrigin is not a separable function. Finding the optimal solution of a non-separable function is computationally more difficult than optimizing a separable function.

2.2 Metaheuristic algorithms

An *algorithm* is an unambiguous specification to instruct a computer, how to solve a class of problems. The general purpose of this thesis,

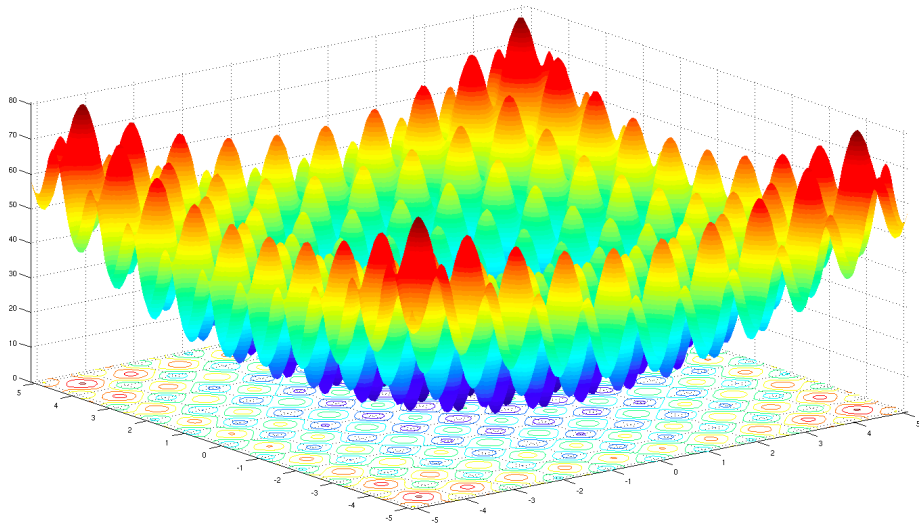


Figure 2.2: Rastrigin function of dimension 2, amplitude $A = 10$, and with bound constraint $-5.12 \leq x_1, x_2 \leq 5.12$.

as it is also the purpose of the research in optimization, is to design efficient algorithm for optimization problems.

The computational complexity of an algorithm is characterized as the time and storage space required for solving a class of problems with a Turing machine. As the complexity is analyzed for a class of problems instead of single instances, it is characterized as a function of the computation time or space in relation with the size of the problem instance being solved. Example problem size includes the number of cities in the TSP or the number of locations in the QAP. The complexity analysis usually focuses on the asymptotic worst-case behavior of the algorithm. As the computation for combinatorial optimization is usually more restrictive in time than in space, the worst-case time complexity is of particular interest.

Different classes of combinatorial optimization problems are categorized with different classes of computational complexity [88]. Complexity class $\mathcal{P}$ denotes the class of problems that can be solved by a deterministic machine in polynomial time. Many seemingly difficult combinatorial optimization problems belong to this class, e.g., the shortest path problem with positive edge weight can be solved by the Dijkstra's algorithm [65], and minimum spanning tree can be solved by the Prim's algorithm [191], etc. Another complexity class $\mathcal{NP}$ denotes the class of problems that can only be solved in polynomial time by a nondeterministic machine, a hypothetical machine model. Every problem in $\mathcal{P}$ is also contained in $\mathcal{NP}$. Whether every problem in $\mathcal{NP}$ is

also contained in $\mathcal{P}$, and thus $\mathcal{NP} = \mathcal{P}$, is one of the most prominent open problems in the theory of computation. For certain hard problems in $\mathcal{NP}$, such as the TSP and the QAP, no polynomial-time algorithm is known to solve them in general. Furthermore, many of such hard problems in $\mathcal{NP}$ such as the TSP and the QAP can be transformed into each other within polynomial time through the so-called polynomial reductions. A problem that is at least as hard as any other problem in $\mathcal{NP}$, i.e., a problem that all the hardest problems in $\mathcal{NP}$ can be polynomially transformed to, is called $\mathcal{NP}$*-hard*. In the meantime, many real-world applications belong to the $\mathcal{NP}$-hard problems. Nevertheless, all the research efforts devoted in the last decades to finding polynomial-time algorithms for $\mathcal{NP}$-hard problems have not led to any success. It is widely believed that these $\mathcal{NP}$-hard problems cannot be solved within polynomial time. This means, the computational time for solving $\mathcal{NP}$-hard problems increases exponentially with the growing problem size, which quickly makes the computation intractable. Consequently, algorithms with state-of-the-art empirical performance for such computationally hard problems are usually highly heuristic.

Similarly, for the computationally hard continuous optimization problems where the functions are multi- modal, non-separable, and high-dimensional, classic algorithms such as gradient descent and Newton's method usually fail to find optimal or near-optimal solutions, and heuristic algorithms are usually preferred.

The practical approaches for such hard problems, as will be mainly focused in this thesis, are metaheuristic algorithms [91] or stochastic local search (SLS) algorithms [108]. In particular, the two most studied algorithms in this thesis belong to the swarm intelligence algorithms [39]. In the following, we introduce in more details the metaheuristic algorithms that are used in this thesis.

2.2.1 Metaheuristics for discrete optimization problems

The metaheuristics for discrete optimization problems can be roughly distinguished into two classes, construction and local search methods.

Construction heuristics

Construction methods aim at building a feasible solution component by component, until a complete solution is obtained. There are various ways how a solution can be constructed from scratch, including the following.

- *Random construction*, which selects each next solution component uniformly at random.

- *Greedy construction*, which always chooses one solution component that has the highest desirability based on a greedy criterion.
- *Adaptive construction*, which adapts the desirability of solution components based on their historical performance learned during the execution of the algorithm.

Take the TSP for example. A complete solution of the TSP refers to a round trip that contains all the nodes of the problem graph. Each arc of the round trip can be regarded as a solution component. A construction heuristic method can start the round trip from a random first node, and iteratively choose to follow an arc from the current node to a next unvisited node, until all nodes have been visited. The three construction methods above differ in the way how to iteratively select the next node. A random construction may select a next arc to an unvisited node uniformly at random. Random construction is usually the fastest way to construct a complete solution from scratch. An example greedy construction, known as *nearest neighbor* heuristic, chooses the next arc that connects to the closest unvisited node from the current node. A greedily constructed solution is usually better than a randomly constructed one, as it exploits the heuristic information such as the cost of traversing an arc. However, it requires more time to search for the best next arc that connects to each unvisited node in each iteration. A typical speed-up is to store a *candidate list* of closest nodes to each node, and search only within such candidate list in each iteration. The greedy construction method usually does not generate solutions with near-optimal quality due to its myopic nature. In comparison, an adaptive construction method chooses the next solution component by considering not only the cost of the solution component but also the quality of the historical complete solutions that contain the solution component. The adaptive construction method can usually generate better solutions than the greedy construction. An example adaptive construction approach is adopted in the *ant colony optimization* metaheuristic algorithm detailed below.

Local search

Local search methods start from a complete solution and try to improve it by local modification operations. The set of candidate solutions that can be reached by modifying a solution is called the *neighborhood* (of a solution). The choice of the neighborhood structure is usually important for the performance of a local search algorithm, and such choice is usually problem specific. However, some classic types of neighborhoods exist as basic neighborhood template. One such classic neighborhood is

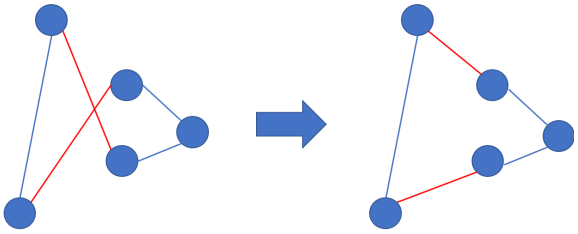


Figure 2.3: Illustration of two-exchange local search. Two edges on the left are exchanged with two other edges on the right

known as *k-exchange* neighborhoods, where two neighboring solutions differ in exactly k solution components. Figure 2.3 provides an illustration of a two-exchange neighborhood, where two solution components (edges) are replaced by two other solution components. The *size* of the k-exchange neighborhood relation, in terms of the number of neighbors for each solution, is in $O(n^k)$, where n is the problem size. As k grows, the size of the neighborhood relation increases exponentially with k. The larger the neighborhood relation, the more and potentially better neighboring solutions that each solution has, but also the longer time it takes to explore the complete neighborhood. An interesting speed-up for exploring the large neighborhoods is to limit the neighboring solution components to each solution component by using a *candidate list* of the immediately best ones based on the evaluation function, and performs neighborhood searches only within such candidate list.

A local search can start from an initial complete solution, move to a neighboring solution if it improves over the current solution. This is done until it reaches a solution that cannot be improved by any neighboring solution any more. This process is usually named *iterative improvement*, and the final solution is a *local optimum* with respect to the neighborhood relation. There are two types of iterative improvement method, namely, *best-improvement* and *first-improvement*. The *best-improvement* method searches all the neighboring solutions of the current solution exhaustively at each local search step, and moves to the neighbor with the best solution quality. The *first-improvement* method avoids a complete evaluation of all the neighboring solutions at each local search step, and accepts the first neighbor that improves the quality of the current solution. The order in which the neighboring solutions are searched can significantly affect the performance of the first-improvement method. It is typical to randomize the order of neighbors at each local search step. The main problem of iterative improvement algorithms is the stagnation at the local optimum. In order to avoid an early stagnation of the local search, more sophisticated

neighborhood relations can be applied. The both examples below systematically vary the types and sizes of neighborhood relations during the local search procedure.

Variable neighborhood descent (VND) [169, 106] is a method which applies a set of local search neighborhood relations iteratively. These neighborhoods are usually ordered by increasing size. VND starts from one neighborhood until it reaches the local optimum with respect to this neighborhood, and then applies local search by the next neighborhood to the current solution. As long as the current solution has been modified by a larger neighborhood search, VND restarts the local search procedure from the smallest neighborhood. The VND termintes when no neighborhood in the set can improve the current solution any more. Then the final solution obtained is locally optimal with respect to all local search neighborhoods. VND is shown to perform better in terms of solution quality and the computation time required to reach high-quality solutions than iterative improvement using one large neighborhood that contains all the VND-constituent neighborhoods [106].

Variable depth search (VDS) [130, 144] explores a very large neighborhood heuristically by applying a series of smaller neighborhoods. At each local search step, VDS starts to apply local search to current solution by the small neighborhood relation. Once a neighbor improves the current solution, VDS searches "deeper" on the improved neighbor by applying a further series of small neighborhood searches, which amounts to a local search of larger neighborhood size. In such way, the neighborhood size is adaptively adjusted based on the attractiveness of the current local search step, which makes VDS capable of exploring large and complex neighborhood efficiently within short computation time.

Note that both variable neighborhood descent and variable depth search start with a complete solution and return a modified solution, hence they can also be regarded as iterative improvement algorithms. Both methods belong to a broader class of local search method called *very large-scale neighborhood search* [3].

Metaheuristic algorithms

A simple local search method such as iterative improvement introduced accepts only improving neighbors until a local optimum is reached. Such method can easily stagnate in a low-quality local optimum. The term *metaheuristic* was first mentioned by Glover [92] when he proposed tabu search. In its original meaning, metaheuristic was defined to be a superimposed method to another heuristic to escape local op-

tima. A simple way to escape a local optimum in the iterative improvement approach is to restart the algorithm from another random initial solution. An alternative is to modify the local search operation to be able to move to a worsening solution from the incumbent. By doing so, the new solution is still of good quality as it is not far from the currently best solution; besides, it also saves the computation to reconstruct a solution from scratch.

The term metaheuristic in a broader sense also refers to a general-purpose heuristic algorithm template that is not problem specific and can be applied to a wide variety of optimization problems. In the following, a number of metaheuristics will be introduced.

Simulated Annealing

One typical mechanism to avoid stagnation at a low-quality local optimum is to occasionally accept worse solutions. A typical such example is to accept a neighboring solution s' from incumbent solution s with objective value $f(s')$ using the *Metropolis* condition [168] as follows:

$$P_{accept}(s, s') = \begin{cases} 1 & \text{if } f(s') \leq f(s) \\ exp(\frac{f(s)-f(s')}{T}) & \text{otherwise} \end{cases} \tag{2.8}$$

The probability function (2.8) always accepts a neighboring solution with better objective function value than the incumbent; a worsening solution may still be accepted probabilistically: the better the quality of the neighboring solution, the higher the probability that it is accepted. The temperature parameter T controls the likelihood of accepting a worsening neighbor. A high value of T makes it easier to accept a worsening neighbor, while a lower value of T makes it less likely to accept a worsening neighbor.

Simulated Annealing (SA) [131, 45] has considered this Metropolis condition and combined it with a temperature control scheme to device an optimization method. SA is analogous to the physical process of annealing for obtaining solids. The annealing procedure heats up a material and then reduces the temperature very slowly in order to arrive at a perfect crystal structure. The analogy in the SA algorithm is the following. SA starts with an initial high temperature T_0, and gradually reduces its temperature by a *cooling rate* α at each iteration, e.g., $T_{i+1} = \alpha \cdot T_i$, where $0 < \alpha < 1$. The cooling rate α is a parameter that influences the performance of SA. The number of local search steps to be performed at each temperature is a parameter that can be set as a multiple of the neighborhood size. SA can be terminated under various conditions, for example, when the incumbent solution is not

changed in a given number of search steps, or the acceptance ratio has dropped under a given threshold.

Tabu Search

In order to escape from a local optimum, *Tabu Search* (TS) [92, 93, 94, 105] exploits search history and makes less use of randomization. TS typically uses a best-improvement local search, and keeps a short-term memory to remember the recent search path. The recently changed solution components in the recent search path are then declared tabu to forbid the local search to immediately return to the visited solutions. The duration in the number of search steps that such solution components are kept tabu is a parameter of the TS algorithm called tabu tenure. The tabu tenure is the most important parameter of TS. A small value of the tabu tenure may lead to search stagnation, while a large value of the tabu tenure will highly restrict the search and, thus, may miss good-quality solutions. A good setting of tabu tenure parameter is usually problem dependent and requires fine-tuning.

There exist different schemes to make the setting of tabu tenure more robust. For example, robust tabu search [220] defines a value range instead of a single value for the tabu tenure parameter, and randomly selects a tabu tenure value from the defined range at each iteration. In another alternative, reactive tabu search [23] adapts the value of the tabu tenure dynamically based on the search history. More specifically, if repeated solutions occur frequently, it is a sign of stagnation and will, thus, increase the tabu tenure; if no solutions are repeated, this indicates that decreasing the tabu tenure is deemed appropriate.

Iterated Local Search

Iterated local search (ILS) [153, 154] is a simple yet effective, general-purpose metaheuristic. ILS contains four components: a construction heuristic method, a subsidiary local search mechanism, a perturbation mechanism, and an acceptance criterion. ILS starts with the construction method, e.g., a random construction, to generate an initial solution, which is improved by the subsidiary local search, e.g., an iterative improvement method, until a local optimum is reached. Then, ILS iterates until the termination criterion is met the three components: firstly, a perturbation to modify the incumbent local optimum to another solution; then the subsidiary local search is applied to improve the perturbed solution to a local optimum; the new local optimum is compared to the incumbent local optimum by the acceptance criterion

to decide whether the new local optimum can replace the incumbent one or not.

These components of ILS should be complementary to each other to achieve a good balance of intensification and diversification. The quality of ILS heavily depends on the subsidiary local search method. Besides the iterative improvement method, more sophisticated local search methods can also be applied, for example, variable neighborhood decent, variable depth search, or tabu search. The perturbation method should modify the incumbent in such a way that it can not be immediately reverted by the subsequent local search steps, and it should ideally lead the subsequent local search to a new local optimum. The strength of perturbation is an important parameter for the ILS method. A weak perturbation may fail to escape from the incumbent local optimum, while too strong perturbation will be amount to a random restart of the method, which may jump too far from the promising region of the solution space and may have very low chance of reaching a good local optimum. The acceptance criterion also have a strong impact to the algorithm. Always accepting the better solution will induce a strong intensification to the algorithm, while always accepting the new solution will result in a strong diversification. A good acceptance criterion should achieve a good balance. A new candidate solution is usually accepted as the new incumbent, if it is at least as good as the old one. If the new candidate solution is worse than the incumbent, occasionally accepting a worsening solution as incumbent, e.g., by applying the Metropolis rule introduced in simulated annealing, may still improve the algorithm performance by diversifying the search to escape from a local optimum.

Ant Colony Optimization

Ant Colony Optimization (ACO) [69, 68, 71, 72], inspired by the collaborative and adaptive foraging behavior of some natural ant species, has become a successful and widely used metaheuristic for solving hard optimization problems. Its success has been proved not only by the large number of problems to which it has been applied, but also by the very good performance ACO algorithms have achieved in many fields, especially for routing problems with complex features (e.g., [63]).

An ACO algorithm is a population-based, iterative construction procedure. In each iteration, a population of ants starts to construct solutions component by component, until each ant has built a complete solution. The definition of solution component is problem-dependent. In the TSP problem, a solution component can refer to an arc in the graph; while in the QAP problem, a solution component can refer to

an assignment of an facility to a location. Each solution component is assigned with two measures: the pheromone trail that reflects the historical desirability of the solution component in the previous constructions; and the greedy information that reflects the immediate desirability of the solution component. Two parameters α and β are used to weigh the relative importance of the two measures, pheromone trails and greedy information, respectively. The next solution component is probabilistically selected based on these two weighted measures. A dominantly large value of β will make the algorithm equivalent to a greedy construction algorithm, while a dominantly large value of α will ignore the greedy information and base the search only on historical solution qualities, which will significantly slow down the convergence to a good solution.

After each iteration, the pheromone trails on all the solution components are updated in two steps. Firstly, the existing pheromone trails are evaporated by a evaporation rate; then the best ants will deposit pheromone on the solution components that compose their constructed solutions, such that these solution components in the best historical solutions have higher possibility to be selected in later iterations. The ants are able to leave pheromone trails such that the solution components that are contained in high-quality solutions will receive more pheromone, which can bias the ants' construction in the subsequent iterations.

2.2.2 Metaheuristics for continuous optimization problems

Many continuous optimization problems that arise from real-world applications are difficult to solve, as they usually have many correlated variables, non-smooth objective functions that derivatives are not easily computed, and the objective function may have many local optima. Some problems are black-box problems that cannot be expressed as an explicit mathematical function. Due to these difficulties, analytical or derivative-based methods [210] are often impractical. Heuristic algorithms for hard continuous optimization problems include derivative-free algorithms [53, 135], and metaheuristic algorithms. Many derivative-free algorithms, such as the *simplex* method by Nelder and Mead [175], Powell's conjugate direction method [190], Powell's method based on quadratic model and trust region [188, 189], and pattern search [222], typically explore a certain region of the search space by an adaptive step size, and aim to reach a local optimum. Therefore, they are often referred to as local search algorithms. Metaheuristic algorithms for continuous optimization have also seen rapid developments in the last decades. These efforts include genetic al-

gorithms [96] and its local search hybrid – memetic algorithm [173], evolution strategies [28, 104], particle swarm optimization [128], differential evolution [211] and ant colony optimization [207, 143]. In the following, we dive deeper into two of them, the particle swarm optimization and evolution strategies, which will be used in later chapters of the thesis.

Particle Swarm Optimization

Similar to Ant Colony Optimization introduced above, Particle Swarm Optimization [128, 129, 187] (PSO) is a population-based method that belongs to the class of Swarm Intelligence algorithms. In a PSO algorithm, a population of simple agents, called *particles*, moves in the domain of an objective function $f : \Theta \in \mathcal{R}^n$, where n is the number of variables to optimize. Each particle i at iteration t has four vectors associated with it:

1. Current position, denoted by $\vec{x}_i^t$: This vector stores the latest candidate solution generated by the particle.
2. Velocity, denoted by $\vec{v}_i^t$: This vector represents the particle's search direction.
3. Personal best position, denoted by $\vec{b}_i^t$: This vector stores the best solution found by the particle since the beginning of the algorithm's execution, that is, $\vec{pb}_i^{\,t} = \arg\min_{s\in\{0,\cdots,t\}}\{f(\vec{x}_i^s)\}$.
4. Neighborhood best position, denoted by $\vec{nb}_i^{\,t}$: This vector stores the best solution found by the particle since the beginning of the algorithm's execution, that is, $\vec{b}_i^t = \arg\min_{s\in\{0,\cdots,t\}}\{f(\vec{x}_i^s)\}$.

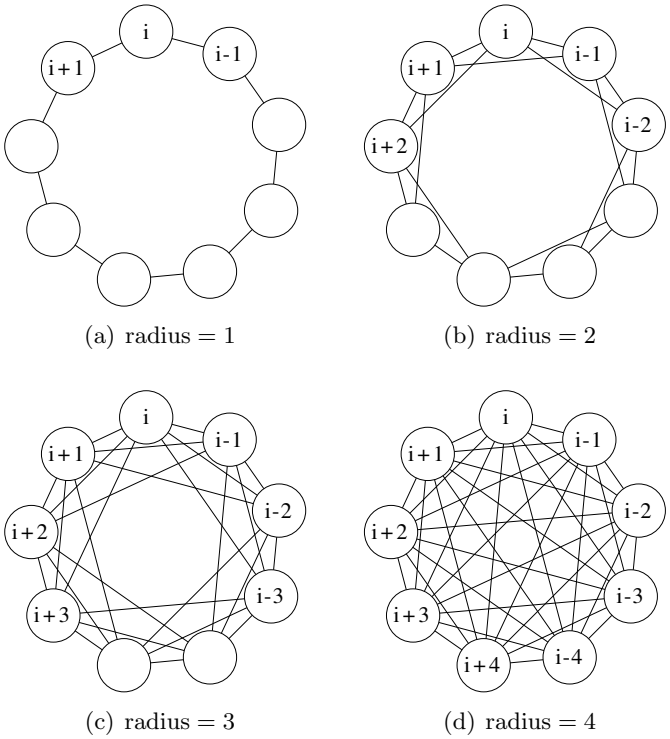
In addition, a neighborhood relation is defined among the particles through a *population topology* [127, 70], which can be thought of as a graph in which nodes represent particles, and edges represent neighbor relations. The behavior of particles in PSO algorithms is usually impacted by the best neighbor; in the following, $\mathrm{n}(i)$ gives the index of the best neighbor of particle i.

A specific variant of a PSO algorithm is the *constricted PSO* [52], where a particle i moves independently for each dimension j using the following rules:

$$v_{ij}^{t+1} \leftarrow \chi\left(v_{ij}^t + \phi_1 U_1(b_{ij}^t - x_{ij}^t) + \phi_2 U_2(b_{\mathrm{n}(i)j}^t - x_{ij}^t)\right) , \tag{2.9}$$

and

$$x_{ij}^{t+1} \leftarrow x_{ij}^t + v_{ij}^{t+1} , \tag{2.10}$$



(a) radius = 1 (b) radius = 2

(c) radius = 3 (d) radius = 4

Figure 2.4: Example population topologies with different connectivity degrees as a result of having neighborhoods of different size. In Fig. (a), radius = 1, in Fig. (b), radius = 2, in Fig. (c), radius = 3, and in Fig. (d), radius = 4.

where χ is a parameter called *constriction factor*, ϕ_1 and ϕ_2 are two parameters called *acceleration coefficients*, U_1 and U_2 are two independent, uniformly distributed random numbers in the range $[0, 1)$.

Figure 2.4 shows four topologies with different levels of connectivity for a swarm of nine particles. These topologies are specified by a parameter called *neighborhood radius*, which is the number of particles on each side of a particle if the particles are arranged as in the figure. The radius parameter takes values in the range $[1, \lfloor N/2 \rfloor]$, where N is the size of the swarm. If the radius is equal to one, the resulting topology is known as a ring topology, and if the radius is equal to $\lfloor N/2 \rfloor$, the topology becomes fully-connected (also known as star topology). This parameter controls the level of exploration and exploitation of the search.

Evolution strategies

The development of evolution strategy (ES) [192, 202] dated back to the mid 1960s by Ingo Rechenberg, Hans-Paul Schwefel and their coworkers in the Technical University of Berlin. It belongs to the class of the evolutionary algorithm, which introduces Darwin's evolution principle to a computerized algorithm for optimization. The simulated evolution typically applies mutation, recombination (also known as crossover), and selection to a population of candidate solutions in order to iteratively evolve to better solutions. The first version of ES is the $(1+1)$-ES [192, 202], which in each iteration mutates from one parent to generate one offspring, and then select one of the two to become the parent of the next iteration. The mutation is typically adding a vector of normally distributed random values to the parent, and the standard deviation of the normal distribution is called the mutation strength or step size. The mutant is selected only if its objective function value (also called fitness) is at least as good as the parent's, otherwise it is discarded.

The two canonical versions of ES are the (μ, λ)-ES and $(\mu + \lambda)$-ES. The parameter μ denotes the number of parents, and λ is the number of offspring. The (μ, λ)-ES, also known as *comma-selection*, where $\mu < \lambda$ must hold, λ offspring are generated from the μ parents in each iteration, and the best μ solutions from the λ offspring are selected as the parents of the next iteration. The $(\mu + \lambda)$-ES, also known as the *plus-selection*, differs from (μ, λ)-ES that also the μ parents joins the λ offspring to be selected, thus, a parent of the current iteration may be selected as one of the μ parents of the next iteration.

A particularly successful ES algorithm for continuous optimization is the covariance matrix adaptation evolution strategy (CMA-ES), proposed by Hansen and Ostermeier [103, 100]. CMA-ES is a (μ, λ) evolution strategy. At an iteration i, the λ offspring are sampled from a multivariate normal distribution,

$$\alpha_{i+1} \sim m_i + \sigma_i \cdot \mathcal{N}(0, C_i), i = 1, \ldots, \lambda \tag{2.11}$$

where

- $\sim$ denotes the same distribution on the left and on the right;
- $\mathcal{N}(0, C_i)$ denotes a multivariate normal distribution that centers at 0 with the covariance matrix C_i of iteration i;
- σ_i denotes the step size of iteration i;
- m_i denotes the mean value of the normal distribution.

The μ parents of iteration i are ranked by their fitness, $x_i^1, x_i^2, \ldots, x_i^\mu$: $f(x_i^1) \leq f(x_i^2) \leq \cdots \leq f(x_i^\mu)$. The distribution parameters m_i, σ_i, and C_i of each iteration i are updated based on the fitness ranking rather than their fitness values, such that they are invariant with respect to a monotonic transformation of the objective function. Each parent is given a positive weight based on their rank,

$$\sum_{j=1}^{\mu} w^j = 1, \quad w^1 \geq \cdots \geq w^\mu, \tag{2.12}$$

such that a parent of higher rank is assigned with a higher weight. The mean value m_i is obtained by recombination, more specifically, a linear combination of μ parents:

$$m_i = \sum_{j=1}^{\mu} x_i^j \cdot w^j \tag{2.13}$$

The update of both the covariance matrix and the step size takes into account the search trajectory over a number of iterations, which is termed as *evolution path*. More specifically, the update of the covariance matrix includes two constitutive parts: a rank-μ-update that estimates the covariance matrix based on the population of the current generation; and a rank-one-update that estimates the covariance matrix based on the evolution path to increase the search range on the more favorable search direction. The update of the step size σ_i also makes use of the evolution path to such that the step size is long if the search advances primarily in the similar direction; and the step size is short if the search turns to opposite direction frequently. For more details about the CMA-ES settings, especially how the default parameters are set, we refer the readers to Hansen's latest version of CMA-ES tutorial [101]. CMA-ES is considered to be a state-of-the-art evolutionary algorithm for continuous optimization [12].

2.3 The algorithm configuration problem

The metaheuristic algorithms introduced in Section 2.2 have parameters, whose value influence the algorithm performance. This thesis targets at devising an automatic tool for setting the values of such parameters offline. The study of offline algorithm configuration is usually divided into two phases. In the *training phase* of the offline configuration, an algorithm configuration that optimizes some measure of algorithm performance is to be determined in a limited amount of

time. The final algorithm configuration is then deployed in a *production phase*, where the algorithm is used to solve previously unseen instances.

A crucial aspect of the *algorithm configuration problem* is generalization, i.e., how well the trained configuration generalizes to new instances in production. In a sense, the problem occurs in other fields such as machine learning. Based on a given set of training instances, the goal is to find high-performing algorithm configurations that perform well on a potentially infinite set of unseen instances that are not available when deciding on the algorithm's parameters. Hence, one assumption that is tacitly made is that the set of training instances is representative for the instances the algorithm faces once it is employed in the production phase. The notions of best performance, generalization, etc. are made explicit in the formal definition of the algorithm configuration problem.

The problem of configuring a parameterized algorithm can be formally defined [32] as a 7 tuple $\langle \Theta, I, P_I, P_C, t, \mathcal{C}, T \rangle$, where

- Θ is the possibly infinite set of candidate configurations.
- I is the possibly infinite set of instances.
- P_I is a probability measure over the set I.
- $t : I \to \Re$ is a function associating to every instance the computation time that is allocated to it.
- $c(\theta, i, t(i))$ is a random variable representing the cost measure of a configuration $\theta \in \Theta$ on instance $i \in I$ when run for computation time $t(i)$.[2]
- $C \subset \Re$ is the range of c, that is, the possible values for the cost measure of the configuration $\theta \in \Theta$ on an instance $i \in I$.
- P_C is a probability measure over the set C: With the notation $P_C(c|\theta, i)$, we indicate the probability that c is the cost of running configuration θ on instance i.
- $\mathcal{C}(\theta) = \mathcal{C}(\theta|\Theta, I, P_I, P_C, t)$ is the criterion that needs to be optimized with respect to θ. In the most general case it measures in some sense the desirability of θ.

[2] To make the notation lighter, in the following we often will not mention the dependence of the cost measure on $t(i)$. We use the term cost to refer, without loss of generality, to the minimization of some performance measure such as the objective function value in a minimization problem or the computation time taken for a decision problem instance.

- T is the total amount of time available for experimenting with the given candidate configurations on the available instances before delivering the selected configuration.[3]

On the basis of these concepts, solving the problem of configuring a parameterized algorithm is to find the configuration $\bar{\theta}$ such that:

$$\bar{\theta} = \arg\min_{\theta \in \Theta} \mathcal{C}(\theta). \tag{2.14}$$

Throughout the whole thesis, we consider for $\mathcal{C}$ the expected value of the cost measure c:

$$\mathcal{C}(\theta) = E_{I,C}[c] = \int c \, \mathrm{d}P_C(c|\theta, i) \, \mathrm{d}P_I(i), \tag{2.15}$$

where the expectation is considered with respect to both P_I and P_C, and the integration is taken in the Lebesgue sense [29]. However, other options for defining the cost measure to be minimized such as the median cost or a percentile of the cost distribution are easily conceivable.

The measures P_I and P_C are usually not explicitly available and the analytical solution of the integrals in Eq. 2.15 is not possible. In order to overcome this limitation, the expected cost can be estimated in a Monte Carlo fashion on the basis of running the particular algorithm configuration on a training set of instances.

The cost measure c in Eq. 2.15 can be defined in various ways. For example, the cost of a configuration θ on an instance i can be measured by the objective function value of the best solution found in a given computation time $t(i)$. In such a case, the task is to tune algorithms for an optimization problem and the goal is to optimize the solution quality reached within a given computation time. In the case of decision problems, the goal is rather to choose parameter settings such that the computation time to arrive at a decision is minimized. In this case, the cost measure would be the computation time taken by an algorithm configuration to decide an instance i. Since arriving at a decision may take infeasibly long computation times, the role played by the function t is to give a maximum computation time budget for the execution of the algorithm configuration. If after a cutoff time of $t(i)$ the algorithm has not finished, the cost measure may use additional penalties [118]. Finally, let us remark that the definition of the algorithm configuration problem applies not only to the configuration of stochastic algorithms, but it extends also to deterministic, parameterized algorithm: in this case, $c(\theta, i, t(i))$ is strictly speaking not anymore a random variable but

[3] In the following, we refer to T also as *computational budget*; often it will be measured as the number of algorithm runs instead of a total amount of computation time.

a deterministic function; the stochasticity is then due to the instance distribution P_I.

One basic question concerns how many times a configuration should be evaluated on each of the available problem instances for estimating the expected cost. Assuming that the performance of a stochastic algorithm is evaluated by a total of N runs, it has been proved by Birattari [31, 33], that sampling N instances with one run on each instance results in the lowest variance of the estimator. Hence, it is always preferable to have a large set of training instances available. If, however, only few training instances are provided, one needs to go back to evaluating algorithm configurations on the instances more than once.

2.4 Types of parameters

As mentioned in the introduction, algorithms can have different types of parameters. There we have distinguished between *categorical* and *numerical* parameters. Categorical parameters typically refer to different procedures or discrete choices that can be taken by an algorithm (or, more in general, an algorithm framework such as a metaheuristic). Example in metaheuristic algorithms include the type of perturbation and the particular local search algorithm used in iterated local search (ILS) or the type of neighborhood structure to be used in iterative improvement algorithms, as well as different variants of the ACO algorithms to be used. Sometimes it is possible to order the categories of these categorical parameter according to some surrogate measure. For example, neighborhoods may be ordered according to their size or crossover operators in genetic algorithms according to the disruptedness they introduce. Hence, sometimes categorical parameters can be converted into ordinal ones. Categorical parameters that may be ordered based on secondary criteria are called *pseudo-ordinal* parameters.[4]

Besides categorical parameters, numerical parameters are common in many algorithms. `Continuous` numerical parameters take as values some subset of the real numbers. Examples of these are the pheromone evaporation rate in ACO, or the cooling rate in simulated annealing. Often, numerical parameters take integer values; such examples include the population size of an ACO algorithm, or the strength of a perturbation that is measured by the number of solution components that change. If such parameters have a relatively large domain, they may

[4]Note that, strictly speaking, binary parameters are also ordinal ones, although they are usually handled without considering an ordering.

be treated in the configuration task as continuous parameters, which are then rounded to the next integer. In the following, we call such integer parameters *quasi-continuous* parameters. In this thesis, the term *numerical parameter* refers to both continuous and quasi-continuous parameters.

Furthermore, it is often the case that some parameter is only in effect when another parameter, usually a categorical one, takes certain values. This is the case of a *conditional* parameter. An example can be given in ILS, where as one option a tabu search may be used as the local search; in this case, the tabu list length parameter is a conditional parameter that depends on whether a categorical parameter "type of local search" indicates that tabu search is used.[5] The iterated F-Race approaches introduced in Chapter 4 are able to handle all aforementioned types of parameters, including conditional parameters.

2.5 Summary

Optimization problems are ubiquitous in our everyday life. It concerns finding the best solution in a solution space with respect to an objective function. Based on the type of variables that defines the solution space, optimization can be categorized into discrete optimization such as the TSP or the QAP, and continuous optimization such as optimizing a sphere function or rastrigin function. In this thesis, we focus on the computationally hard optimization problems that cannot be easily solved by known exact algorithms. For such hard optimization problems, metaheuristic algorithms are usually applied to obtain high-quality solutions. State-of-the-art metaheuristics for discrete optimization include simulated annealing, tabu search, iterated local search, ant colony optimization, etc. State-of-the-art metaheuristics for continuous optimization include particle swarm optimization, evolutionary strategy, etc. This thesis studies the algorithm configuration problem, which targets at devising automatic tools for setting the values of algorithm parameters for a given class of hard optimization problem.

[5]It is worth noticing that sometimes it may make sense to replace a numerical parameter by a categorical parameter plus a conditional parameter, if changing the numerical parameter may lead to drastic changes in design choices of an algorithm. Consider as an example the probability of applying a crossover operator. This parameter may take a value of zero, which indicates actually that no crossover is applied. In such cases it may be useful to introduce a binary parameter, which indicates whether crossover is used or not, together with a conditional parameter on the crossover probability, which is only used if the binary parameter indicates that crossover is used.

Chapter 3

Configuration algorithms

A *configuration algorithm* (or *configurator*), also known as *tuning algorithm* (or *tuner*), is a method for solving the configuration problem described in the previous chapter. In this chapter, we will define a framework for constructing a configuration algorithm. As discussed in the previous chapter, a configuration problem can be regarded as a black-box, mixed-discrete-continuous stochastic optimization problem. There are two sources of stochasticity in the configuration problem. First, the algorithm to be configured can be stochastic, e.g., when random decisions are made during algorithm execution. Second, the goodness of each configuration depends on the sampling of the instances from the problem class of interest. To solve the configuration problem, a configurator should be composed so as to handle the two aspects: the *search* aspect (also known as *sampling* aspect [233]) that iteratively generates candidate algorithm configurations from the black-box mixed-discrete-continuous parameter space; and the *evaluation* aspect (also known as *stochasticity handling* aspect [233]) that evaluates the generated configurations under the inherent stochasticity in the configuration problem and selects the more promising one(s). The evaluation method and search method that suit the configuration purpose are detailed in Section 3.1 and 3.2, respectively. How the evaluation method and search method are combined is discussed in Section 3.3, including an iterated selection approach and a post-selection approach. A framework of how a configurator is constructed is illustrated in Figure 3.1. Section 3.4 provides a brief survey of related works.

Note that given a fixed evaluation budget, there exists a trade-off in the evaluation aspect of a configurator. On one hand, the so-called *exploitation* aspect of a configuration problem means that by increasing the number of evaluations on each generated configuration, we increase our confidence in the quality of the evaluated configuration; on the

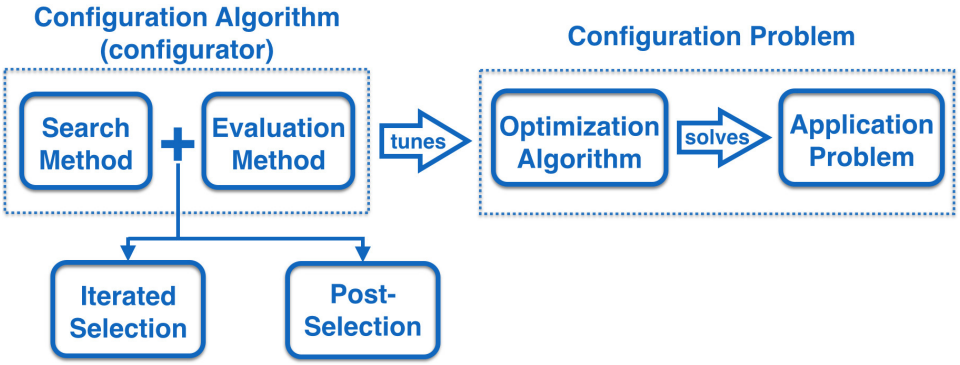


Figure 3.1: A tuning problem consists of an optimization algorithm to be tuned, and an application problem to which this underlying optimization algorithm is applied. A tuning algorithm (or tuner) for addressing the tuning problem usually consists of a search method, which iteratively generates candidate configurations, and an evaluation method, which evaluates the generated configurations. The search method and the evaluation method can be combined using either an iterated selection approach or a post-selection approach.

other hand, the so-called *exploration* aspect means that it is desirable to generate as many configurations as possible so that a high-performing configuration is more likely to be found. A good configurator should utilize the evaluation budget wisely in order to balance the exploitation and exploration aspects.[1]

3.1 Evaluation method: Evaluation budget allocator for ranking and selection

The first component of a configurator concerns the aspect of evaluation, or stochasticity handling. Given is a set of candidates Θ, whose qualities are defined by Equation (2.15) and are estimated by Monte-Carlo simulation. Usually given is an *evaluation budget* B, i.e., the maximum number of times that the candidates can be evaluated; we assume $B \gg |\Theta|$. The task is either to select the best from the candidate set; or to rank the set of candidates, such that the selected candidates are most likely to be the best ones; or, on the other hand, the evaluation budget used during the ranking and selection process is minimum while

[1]Note that the exploitation versus exploration may also refer to the balance that is used in the search method: generate solutions close to the best ones versus exploration of the search space. In this thesis, we mainly focus on the tradeoff between search method and evaluation method.

the best candidate is selected with a given confidence threshold. This process is known as *ranking and selection*. How to allocate the available evaluation budget for the task of ranking and selection is an important topic in the study of configuration algorithms, since the evaluation error of a candidate reduces as the number of evaluations increases, and, on the other hand, the evaluation budget is usually very limited due to the high computational expense. Approaches for this purpose is called *evaluation budget allocator*. Two evaluation budget allocators are introduced in this section: a brute-force *repeated evaluation* and a *racing* method, more specifically, F-Race.

3.1.1 Repeated evaluation

The conceptually simplest approach for allocating evaluation budget for ranking or selection from a set of candidates Θ is to repeat the evaluation of each candidate $\theta \in \Theta$ the same number of times. In this case, each candidate receives the same number of evaluations. Once the overall evaluation budget is consumed, the candidate with the lowest estimate, either by average or by other statistical measures such as median, is chosen as the best candidate. This approach is dubbed *repeated evaluation*, and the number of repetitions is usually denoted throughout the thesis by nr. The main drawback of using this brute-force approach is that the poor performing candidates are evaluated with the same amount of computational resources as the good ones.

3.1.2 F-Race

The racing approach

As one possibility to avoid the disadvantages of the brute-force approach we have used a racing approach. The racing approach originated from the machine learning community [156], where it was first proposed for solving the model selection problem [43]. We adapted this approach to make it suitable for the algorithm configuration task. The racing approach performs the evaluation of a finite set of candidate configurations using a systematic way to allocate the computational resources among them. The racing algorithm evaluates a given finite set of candidate configurations `step` by `step`. At each `step`, all the remaining candidate configurations are evaluated in parallel,[2] and the poor candidate configurations are discarded as soon as sufficient sta-

[2] A round of function evaluations of surviving candidate configurations on a certain instance is called an `evaluation step`, or, simply, a `step`. By `function evaluation`, we refer to one run of the candidate configuration on one instance.

tistical evidence is gathered against them. The elimination of the poor candidates allows to focus the evaluations on the most promising ones to obtain lower variance estimates for these. In this way, the racing approach overcomes the two major drawbacks of the brute-force approach. First, it does not require a fixed number of steps for each candidate configuration but it determines it adaptively based on statistical evidence. Second, poor performing candidates will not be evaluated as soon as enough evidence is gathered against them. A graphical illustration of the *racing* algorithm and the *brute-force* approach is shown in Figure 3.2.

To describe the racing approach formally, suppose a sequence of training instances i_k, with $k = 1, 2, \ldots$, is randomly generated from the target class of instances I following the probability model P_I. Denote by c_k^θ the cost of a single run of a candidate configuration θ on instance i_k. The evaluation of the candidate configurations is performed incrementally such that at the k-th `step`, the array of observations for evaluating θ,

$$\mathsf{c}^k(\theta) = \left(c_1^\theta, c_2^\theta, \ldots, c_k^\theta\right),$$

is obtained by appending c_k^θ to the end of the array $\mathsf{c}^{k-1}(\theta)$. A *racing* algorithm then generates a sequence of nested sets of candidate configurations

$$\Theta_0 \supseteq \Theta_1 \supseteq \Theta_2 \supseteq \ldots,$$

where Θ_k is the set of the surviving candidate configurations after step k. The sets of surviving candidate configurations start from a finite set $\Theta_0 \subseteq \Theta$, which is typically obtained by sampling $|\Theta_0|$ candidate configurations from Θ. How the initial set of candidate configurations can be generated is the topic of Chapter 4. The step from a set Θ_{k-1} to Θ_k is obtained by possibly discarding some configurations that appear to be suboptimal on the basis of information that becomes available at step k.

At step k, the surviving candidates in Θ_{k-1} are tested on a new instance i_k. Their observed cost c_k^θ for $\theta \in \Theta_{k-1}$ is appended to $\mathsf{c}^{k-1}(\theta)$ to form the arrays $\mathsf{c}^k(\theta)$. Step k terminates defining set Θ_k by dropping from Θ_{k-1} the candidate configurations that appear to be suboptimal based on some statistical test that compares the arrays $\mathsf{c}^k(\theta)$ for all $\theta \in \Theta_{k-1}$.

The above described procedure is iterated and stops either when all candidate configurations but one are discarded, a given maximum number of instances have been sampled, or when the predefined com-

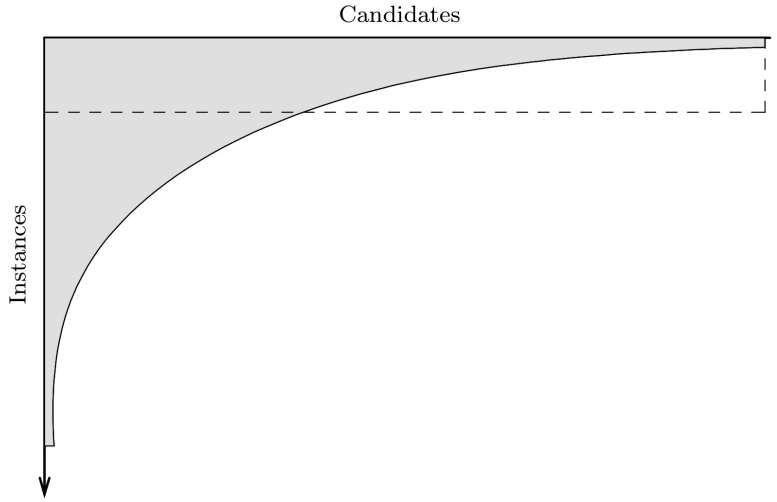


Figure 3.2: Graphical illustration of the allocation of configuration evaluations by the racing approach and the brute force repeated evaluation. In the racing, as soon as sufficient evidence is gathered that a candidate is suboptimal, such candidate is discarded from further evaluation. As the evaluation proceeds, the racing method focuses on the most promising candidates by evaluating them with more instances. On the other hand, the repeated evaluation tests all given candidates on the same number of instances. The shadowed figure represents the computation performed by the racing approach, while the dashed rectangle the one of the repeated evaluation. The two figures cover the same surface, that is, the two approaches are allowed to perform the same total number of experiments.

putational budget B has been exhausted.[3]

The peculiarity of F-Race

F-Race is a racing algorithm based on the non-parametric Friedman's two-way analysis of variance by ranks [54], for short, Friedman test. This algorithm was first proposed by Birattari et al. [36] and studied in detail in Birattari's PhD thesis [32, 33].

To describe F-Race, assume it has reached step k, and $m = |\Theta_{k-1}|$ candidate configurations are still in the race. The Friedman test assumes that the observed costs are k mutually independent m-variate

[3]The computational budget may be measured as a total available computation time T (see definition of the configuration problem on page 28). It is, however, often more convenient to define the maximum number of function evaluations, if each function evaluation is limited to a same amount of computation time.

random variables

$$\begin{aligned} b_1 &= \left(c_1^{\theta_{v_1}}, \; c_1^{\theta_{v_2}}, \; \ldots, \; c_1^{\theta_{v_m}}\right) \\ b_2 &= \left(c_2^{\theta_{v_1}}, \; c_2^{\theta_{v_2}}, \; \ldots, \; c_2^{\theta_{v_m}}\right) \\ \vdots \quad & \quad \vdots \quad \vdots \quad \ddots \quad \vdots \\ b_k &= \left(c_k^{\theta_{v_1}}, \; c_k^{\theta_{v_2}}, \; \ldots, \; c_k^{\theta_{v_m}}\right) \end{aligned}$$

called blocks, where each block b_l corresponds to the computational results obtained on instance i_l by each surviving configuration at step k.

Within each block, the costs c_l^θ are ranked in non-decreasing order; average ranks are used in case of ties. For each configuration $\theta_{v_j} \in \Theta_{k-1}$, R_{lj} is the rank of θ_{v_j} in block b_l, and $R_j = \sum_{l=1}^{k} R_{lj}$ is the sum of ranks for configuration θ_{v_j}, over all instances i_l, with $1 \leq l \leq k$. The test statistic used by the Friedman test is the following [54]:

$$T = \frac{(m-1)\sum_{j=1}^{m}\left(R_j - \frac{k(m+1)}{2}\right)^2}{\sum_{l=1}^{k}\sum_{j=1}^{m} R_{lj}^2 - \frac{km(m+1)^2}{4}}.$$

Under the null hypothesis that all candidates are equivalent, T is approximately χ^2 distributed with $m-1$ degrees of freedom [182]. If the observed value of T is larger than the $1-\alpha$ quantile of this distribution, the null hypothesis is rejected. This indicates that at least one candidate configuration gives better performance than at least one of the others.

If the null hypothesis is rejected in this *family-wise* test, it is justified to do pairwise comparisons between individual candidates. There are various ways of conducting these Friedman *post hoc tests*. For F-Race, we have chosen one particular one that is presented in the book of Conover [54]: candidates θ_j and θ_h are considered to be statistically significantly different if

$$\frac{|R_j - R_h|}{\sqrt{\frac{2k\left(1-\frac{T}{k(m-1)}\right)\left(\sum_{l=1}^{k}\sum_{j=1}^{m} R_{lj}^2 - \frac{km(m+1)^2}{4}\right)}{(k-1)(m-1)}}} > t_{1-\alpha/2},$$

where $t_{1-\alpha/2}$ is the $1-\alpha/2$ quantile of the Student's t distribution.

If F-Race does not reject at `step` k the null hypothesis of the family-wise comparison, all candidate configurations in Θ_{k-1} pass to Θ_k; if

the null hypothesis is rejected, pairwise comparisons are performed between the best candidate configuration and each other one. The best candidate configuration is selected as the one that has the lowest expected rank. All candidate configurations that result significantly worse than the best one are discarded and will not appear in Θ_k.

When only two candidates remain in the race, the Friedman test reduces to the *binomial sign test for two dependent samples* [203]. However, in the F-Race algorithm, the *Wilcoxon matched-pairs signed-ranks test* [54] is adopted, for the reason that the Wilcoxon test is more powerful and data-efficient than the binomial sign test in such a case [204].

In F-Race, the test statistic is based on the ranking of the candidates. Ranking plays an important two-fold role. The first one is due to the non-parametric nature of a test based on ranking. A second role played by ranking in F-Race is to implement in a natural way a blocking design [61, 171]. By focusing only on the ranking of the different configurations within each instance, this blocking design becomes an effective way for normalizing the costs observed on different instances.

3.2 Search method: Black-box optimizers

In the previous section, it is assumed that $|\Theta| \ll B$, i.e. the number of parameter configurations is far smaller than the available configuration budget, which means the parameter space Θ is a small finite set. This usually doesn't hold in practice. In fact, considering the algorithm parameters which take values of real numbers, such as evaporation rate in ant colony optimization, or cooling rate in simulated annealing, the set of possible parameter configurations will be infinite. Therefore, a mechanism that effectively explores the parameter space and locates the promising region or even the optimal parameter configuration, is essential for a high-performing configurator. A mechanism for this purpose is called a *search* method (or *sampling* method[4]) throughout this thesis.

There exist different types of parameters, namely numerical parameters and categorical parameters. Thus, the search method of a configurator should be able to explore all types of parameters. In the following, we distinguish mechanisms for sampling numerical parameters and categorical parameters. Section 4.2.2 introduces a full-functioning configurator I/F-Race, which includes an ad-hoc search method for

[4]Note that the term *sampling* is used differently in some literature for sampling instances, i.e., sample a set of instances used for evaluation. Here, and throughout the thesis, what *sampling* refers to is sampling parameter configurations, i.e., sample in the parameter space.

numerical, categorical and conditional parameters.

3.2.1 Searching the numerical parameter space

In this thesis, the main focus is on the numerical parameters such as real-valued parameters. It has further been shown in [233] that integer-valued parameters with a large range, such as colony size in ant colony optimization and population size in particle swarm optimization, can be handled as real-valued parameters and then be rounded to the nearest integer. Thus, search in such a numerical parameter space can be regarded as a (quasi-)continuous optimization problem:

$$\min_{p \in \Theta} f(p) \tag{3.1}$$

with $f : \Theta \subseteq \mathbb{R}^d \to \mathbb{R}$, where the domain Θ refers to the possible values that the algorithm parameters can take, which can be usually viewed as a bounded continuous space. Let d be the dimensionality of Θ, i.e., the number of algorithm parameters. One of the characteristics of handling the search aspect of a configuration problem is that the function f is a black-box: it does not have an analytical expression, and its value is given by the quality of the solution obtained by running the algorithm with a parameter configuration $\theta \in \Theta$. Clearly, for optimizing such a black-box function f, no continuity or smoothness can be assumed and no derivative information is available. This makes most of the conventional, gradient-based continuous optimizers inapplicable. Hence, derivative-free optimizers should be a good choice.

There have been many studies on derivative-free optimizers for dealing with black-box functions [172, 102]. To better explore the parameter space, the derivative-free search methods should take into account the following.

Expensive evaluation. The continuous search methods in the literature are usually developed or compared based on analytical functions. Such algorithm studies usually focus on a relatively high budget, typically with a total number of function evaluations larger than 10^4. However, the evaluation of a real-world algorithm configuration consists in running the algorithm with the configuration typically multiple times, and is thus computationally much more expensive than evaluating a function. Depending on the experimental setting, each configuration process usually generates no more than thousands of candidate configurations. Besides, configurating numerical parameters can be used as a subproblem of an iterative two-phase configuration process,

which iteratively fixes the categorical parameters in the first phase and then (quickly) optimizes the numerical parameters in the second phase as a subproblem. Such a subproblem usually samples only tens to hundreds of numerical parameter configurations.

Low dimension. The configuration of numerical parameters can be considered as a subtask of the configuration problem when all the categorical parameters are set. Thus, the parameter space in such case usually has a small number of numerical parameters. Therefore, it is essential for us to focus on search methods that perform well on low-dimensional continuous spaces with a low number of evaluations.

3.2.2 Searching the categorical parameter space

The search problem in the categorical parameter space can be stated as follows:

$$\min_{p \in \Theta} f(p) \tag{3.2}$$

with $f : \Theta \subseteq C_1 \times \cdots \times C_d \to \mathbb{R}$, where C_i refers to a set of categorical values for the i-th categorical parameter, and thus the domain $\Theta \subseteq C_1 \times \cdots \times C_d$ is a d-dimensional space of categorical variables. Optimizing in a multi-dimensional, categorical space is usually referred to as mixed-variable optimization [208, 1, 143], multidisciplinary design optimization [159], or categorical optimization [147]. The typical number of categorical parameters depends on the algorithm to be configured. For example, the ACOTSP software [212] has three categorical parameters to be configured (ACO variant, local search type, and whether to use don't look bit) [37]; the state-of-the-art branch-and-cut algorithm implemented in the mixed integer programming solver CPLEX version 12.6 [120] has 74 categorical parameters to be configured [115, 237].

Note that a conditional parameter in an algorithm (if any) is usually conditional to a categorical parameter. In such case, whether a conditional parameter is effective depends on the value of its *master* parameter. There are two possible ways to handle conditional parameters in practice.

Search and undo. As done in Iterated F-Race [37] (see Section 4.2.2), search the space for all parameters, then check each conditional parameter for its master parameter's value, and nullify the conditional parameters that are not deemed to be effective due to its master parameter.

Multi-level search. Divide the search of a parameter configuration by multiple levels depending on the conditional hierarchy, fix the

Algorithm 3.1 Iterated selection

Require: parameter space X, a black-box iterated search method S, an evaluation method E
  set iteration counter $l = 1$, set initial elite configurations $\Theta_*^0 = \emptyset$
  repeat
    proceed S to search the set of configurations $\Theta_0^l \in X$ of iteration l
    include elite configurations $\Theta_0^l = \Theta_0^l \bigcup \Theta_*^{l-1}$
    rank and select elite configurations Θ_*^l from Θ_0^l using E
    update S based on the selected elite configurations Θ_*^l
    $l = l + 1$
  until termination criterion is met
  return the best configuration $\theta^* \in \Theta_*^l$

value of the higher-level master parameters, determine the eligibility of the rest of the parameters, and then search in the space of eligible parameters.

3.3 Combining evaluation method and search method

Given an evaluation method and a black-box optimizer as search method, a configurator essentially consists of an efficient, non-trivial combination of the two. We discuss two possibilities below, namely, the iterated selection and the post-selection mechanism. The key difference between them is how to keep track of the incumbent, i.e., the best-so-far parameter configuration found during the configuration process.

3.3.1 Iterated selection

Iterated selection, as outlined in Algorithm 3.1, refers to the algorithm configuration approaches where two distinctive phases are iterated: firstly, new candidate configurations are generated by a search method; and then these generated configurations are evaluated against the incumbent configuration by an evaluation method. The evaluation results are used to bias the search of the next iteration, and to possibly update the incumbent configuration.

Most of the established configurators are based on some form of iterated selection, including SPO [17] and SPO+ [117], iterated racing techniques such as iterated F-Race [14, 37], MADS/F-Race [236], and CMA-ES/F-Race [233], or FocusedILS [118]. These methods include the incumbent from iteration to iteration. Some of them consider

Algorithm 3.2 Post-selection

Phase 1: elite qualification. Run configurator and collect the best configurations as *elite* configurations. Each elite configuration θ_e is stored in Θ_e. A budget of R_n ($n = |\Theta_e|$) depending on the number of elite configurations is reserved for Phase 2. Go to Phase 2 when the budget for the qualification phase finishes.

Phase 2: elite selection. Use an evaluation method, e.g. racing, to select the best θ^* from Θ_e.

using an intensification mechanism to preserve the incumbent (e.g. FocusedILS [118] and SPO+ [117], and the incumbent protection mechanism in MADS/F-Race [233]).

The possible drawbacks of iterated selection are that an incumbent may be lost if no specific mechanism for incumbent preservation is used, while if an incumbent preservation mechanism is used, it may be too aggressive in eliminating potentially promising new candidates, leading to stagnation as observed occasionally in FocusedILS [118]. Besides, the applicability of iterated selection is limited, since some search method might not require ranking and selection of candidate configurations in each iteration, for example, model-based algorithms such as BOBYQA (see Sec. 6.1.1)

3.3.2 Post-selection

The basic idea of the post-selection mechanism is to divide the configuration process into two phases: a first *elite qualification* phase and a second *elite selection* phase. During the *elite qualification* phase, a number of elite configurations are identified by running a configurator. These elite configurations can be collected by, for example, enforcing quick convergence of the configurator and then taking the best configuration in each independent restart. Alternatively, different configurators may be run simultaneously and the best configurations returned by various configurators may be qualified as elites. In the *elite selection* phase, an evaluation method will be applied to select the best from these elite configurations. See Algorithm 3.2 for a summary of the post-selection mechanism. A number of configurators are devised following the post-selection approach and investigated in Chapter 6. We also compare post-selection configurators to iterated selection techniques such as in the iterated racing methods and FocusedILS.

3.4 Related works

In recent years, automating the process of finding good algorithm parameter settings has attracted more and more research attention [97, 107, 22]. These research efforts can be roughly categorized into online adaptation and offline configuration. We provide a survey and discussion about the two lines of works in Section 3.4.1, and goes into more details of existing offline configurators in Section 3.4.2. A literature survey of applications of F-Race is given in Section 3.4.3.

3.4.1 Offline configuration versus online adaptation

The automatic algorithm configuration methods can generally be categorized into two classes: offline method and online method. The offline configuration method, also referred to as parameter tuning, consists in finding a good parameter configuration for the target algorithm in a training phase *before* the algorithm run [33]. It learns and selects the best parameter configuration based on a set of available training instances that resemble the future, unseen instances. These training instances can, in practice, be obtained from a simulated instance generator or historical data if the target optimization problem happens in a recurring manner. Examples for the latter include optimizing logistic plans on weekly delivery demand, arranging university course timetable on a semester basis, or scheduling school buses on a yearly basis.

There exist a number of established approaches for the task of offline configuration[5], for example, statistical racing techniques [36] and iterated racing [37], an iterated local search approach (ParamILS) [118], gender-based genetic algorithm (GGA) [5], random forest model-based approach SMAC [111], and continuous optimizers with post-selection techniques [233, 242] specifically for numerical parameters. They regard the target algorithm to be configured as a black box, and can be applied to any algorithms without human intervention. Once the training phase is finished, the target algorithm is deployed using the best parameter configuration found, to solve previously unseen instances.

Most of the applications of offline configuration fix the configuration during the algorithm run, regardless of the instances encountered. There exist also specific offline configuration approaches for heterogeneous instances. One such approach is called portfolio-based algorithm selection [194, 137]. The most successful example is SATzilla [228], which builds a portfolio of algorithms, trains an empirical hardness model [142] that predicts runtime for each algorithm in the portfolio

[5]Some of the established offline configuration approaches listed here were developed after the work presented in this thesis.

based on features of different instances, and then selects an algorithm from the portfolio with the best predicted performance for a given instance to be solved. A limitation in SATzilla is that it requires a priori domain knowledge to build an uncorrelated static portfolio. A follow-up work to overcome this limitation, Hydra [229], uses offline configuration to dynamically generate complementary configurations to update the portfolio. Another approach for handling heterogeneous instances is instance-specific algorithm configuration (ISAC) [125, 146]. The basic idea is to cluster the instances based on their features and apply offline configuration of the algorithm on each cluster of instances. Both approaches have reported promising results. For example, SATzilla is reported to be the most successful solver in the SAT competition. However, both portfolio-based selection and instance-specific configuration approaches keep their configuration fixed when solving an instance.

On the other hand, an online configuration method varies the parameter setting *during* the algorithm run. Such approaches are also referred to as parameter adaptation [4] or parameter control [73]. There are two potential advantages of varying parameter settings at run time. Firstly, a good-performing fixed parameter setting across a class of problem may not perform well on each particular instance, especially when the instances are heterogeneous. Secondly, different search stages may prefer different parameter settings, for example, a rule of thumb is to use more explorative settings at the beginning of the search and then switch to more exploitive settings at the later stage to converge the search. The online parameter adaptation problem has attracted much attention and research efforts, especially in the field of evolutionary algorithm [148]. The usage of machine learning techniques in parameter adaptation is also the unifying research theme of reactive search [22].

There are various ways of varying the configuration over time. Matthews et al. [161] has reported a uniformly random variation of parameter values over time improves the performance and robustness of an ACO algorithm with a fixed setting. Stützle et al. [218] discuss parameter adaptation techniques for ACO algorithms and show the usefulness of a simple parameter variation scheme by a pre-scheduled function. Most of the other works focus on more sophisticated approaches to vary parameter settings over different search stages adaptively based on runtime feedback from the search. Such examples include reactive search [22] and adaptive operator selection [60, 75]. However, there exists no strong experimental evidence to support the assumption that adaptive approaches vary the parameter values at different search stages better than the non-adaptive approaches. Mascia

et al. [160] have conducted an empirical study on reactive tabu search (RTS) [23], and found that setting the tabu list length randomly with the same empirical distribution of the RTS adaptation scheme leads to performance very close to that of RTS. They further used offline configuration to tune a distribution for randomly setting tabu list length at each search stage in the robust tabu search [220], which significantly outperforms RTS. The PhD thesis of Fialho [75] (pages 161–170) also showed that randomly generating operators by an offline tuned distribution performs equivalently to his best adaptive operator selection method applied to differential evolution.

Although the online and offline methods approach the automatic algorithm configuration problem differently, they can be regarded as complementary to each other, and thus can be incorporated together. For example, the online methods usually have a number of hyper-parameter to be configured, and this can be fine-tuned by an offline method [78, 241]. Besides, offline methods can provide a good starting parameter configuration, which is then further adapted by an online mechanism once the instances to be solved are given. However, an empirical study by Pellegrini *et al.* [185] has shown that adapting the offline tuned configuration usually performs worse than fixing the tuned configuration for ant colony optimization algorithms. Francesca *et al.* [82] also found that using an offline tuned static operator is more preferable than using online adaptation in operator selection. A study by Yuan *et al.* [241] further showed that varying the operators by an offline tuned probability distribution performs better than offline tuned static operator as well as adaptive operator selection strategies.

3.4.2 Offline configuration algorithms

Most of the existing offline algorithm configurators can be generally classified as iterated selection approaches. They iteratively generate candidate configurations by a search method. Then they evaluate them and compare them to the incumbent configuration by an evaluation method. Next, they bias the search and update the incumbent according to the evaluation results. In the following, we analyze established algorithm configurators based on the framework of iterated selection.

CALIBRA

CALIBRA [2] is one of the first works that adopt a design of experiment (DoE) approach [171] for algorithm configuration. It was designed for configuring up to five numerical or ordinal algorithm parameters by means of DoE and DoE-based iterated local search. It starts with a

full factorial design for each parameter with two levels (at 25% and 75% of the parameter range, respectively). The subsequent iterative local search procedure generates in each iteration nine configurations around the best-so-far configuration by means of a fractional factorial design. The step size of the newly generated configurations from the best-so-far configuration gradually decreases such that it slowly converges. When a local optimum is reached, it restarts from a new starting configuration that is "perturbed" from the locally optimal configuration. The search method underlying CALIBRA can be classified as an iterated local search procedure: it is initialized by DoE, defines the neighborhood of each iteration by DoE, and applies an ad-hoc perturbation to obtain a new starting configuration from the locally optimal one.

Compared to how the search method is described in [2], CALIBRA's evaluation method is relatively brief and undetailed. Based on [2, page 107], CALIBRA uses repeated evaluation as the evaluation method. The number of repetitions (nr) is determined by the number of total algorithm evaluations (aka. configuration budget) divided by the number of candidate configurations, which is fixed to 350 by default. This might not be a good setting, as the best setting of nr depends on the nature of the target algorithm to be configured [236] as well as the total budget level [242], and the setting of nr can have a significant influence on the performance of the algorithm configurator. Another algorithm configurator based on iterated local search, ParamILS, as detailed below, uses a more sophisticated evaluation method for algorithm configuration.

ParamILS

ParamILS [119, 118] is designed to configure categorical parameters. If there are numerical parameters to be configured, they need to be discretized before running ParamILS. The search method underlying ParamILS is an iterated local search mechanism. The search starts by comparing a number of configurations generated uniformly at random together with the default configuration. Each local search operation is to generate a new configuration from the incumbent configuration by randomly changing the value of one randomly selected parameter. A perturbation mechanism is applied probabilistically.

There are two main versions of ParamILS, namely, BasicILS and FocusedILS. Their difference lies in the evaluation methods used. The basic version of ParamILS uses repeated evaluation as its evaluation method: evaluates each generated configuration by a fixed number nr of evaluations. A new configuration with better evaluation result based on the nr evaluations will become the new incumbent. A more

advanced version of ParamILS, FocusedILS, uses a so-called intensification mechanism as evaluation method to evaluate and decide which is the new incumbent. In the local search phase, FocusedILS generates one configuration at a time, and compares it with the incumbent. The comparison of the newly generated configuration with the incumbent is done sequentially, on an increasing number of instances that the incumbent has been evaluated on. Once the new configuration is worse than the incumbent in one comparison, it is eliminated immediately, and the incumbent is evaluated on one additional instance. If the new configuration is not eliminated until its number of evaluations reaches the number of the incumbent, it becomes the new incumbent, and will be evaluated on a number of additional instances.

In [119], ParamILS is compared with CALIBRA, and it is found to be the substantially better performing configurator in their context.

MADS

The mesh adaptive direct search (MADS) algorithm [10] is a derivative-free algorithm for continuous optimization. It is a mesh-based search method, and iteratively adapts its mesh coarseness and search direction based on the incumbent point. In Audet and Orban [9], MADS is applied as an algorithm configurator to fine-tuning the numerical parameters of a trust region algorithm. This MADS-based algorithm configurator adopts MADS as the search method in the continuous parameter space. The goodness of each configuration generated by MADS is evaluated on all the instances in a benchmark set, and there is no separation of training set and testing set of instances. In such case, it is not validated whether the tuned configuration can be generalized to other untested instances. Besides, this amounts to repeated evaluation on each configuration. Repeated evaluation is simple but not really effective, see the discussion in CALIBRA. Section 4.5 [236] further studies how to apply MADS as a search method for algorithm configuration, especially how to decide on the number of evaluations, and how to hybridize MADS with racing techniques.

SPO

Sequential Parameter Optimization [17] is one of the first works that introduces a model-based search method to tackle the automatic algorithm configuration problem. It applies the Gaussian process, also known as Kriging models, as a search method.[6] The Gaussian process

[6] Earlier work of Bartz-Beielstein *et al.* [20] applies regression tree as a surrogate model, such that not only numerical parameters but also categorical parameters can be configured.

tries to build a model to approximate the quality response of the parameter space, and samples a new configuration where the expected improvement is maximized. The Kriging model is sequentially refined as more configurations are evaluated. The original implementation of the Gaussian process in SPO adopts the MATLAB Kriging toolbox DACE (Design and Analysis of Computer Experiments) [152, 198]. Note that the Gaussian process search method also forms the basis for the efficient global optimization (EGO) [123] algorithm for the black-box continuous optimization. However, both DACE and EGO are proposed for deterministic optimization problems. A main algorithmic contribution of SPO is to extend them by a *stochasticity handling* (also termed as *performance evaluation* in [17], noise reduction in [21], or intensification in [117]) mechanism to deal with the noise existing in the algorithm configuration problem.

The main idea underlying SPO's stochasticity handling mechanism is the following. In each SPO iteration, the newly generated candidate configuration and the incumbent configuration are evaluated by the same number of evaluations, and this number of repeated evaluations increases gradually with the number of iterations. As candidate configurations generated in the later SPO iterations are usually of higher quality and lie in the more interesting region of the Kriging model, increasing the number of their evaluations will decrease their estimation error and improve the model accuracy in the more interesting region. The SPO version 0.3 [17, 21] doubles the number of evaluations from iteration to iteration when the incumbent is not rejected, while a later version 0.4 of SPO [18] increments the number of evaluations by one at each non-rejecting iteration for the incumbent.

A further investigation into SPO is done by Hutter et al. [117]. A main contribution is to replace the stochasticity handling mechanism in SPO described above by the intensification mechanism that is used in FocusedILS [118]. The proposed variant of SPO with intensification mechanism, dubbed SPO+ [117], is compared with SPO version 0.3 and 0.4, and it significantly improves its performance in algorithm configuration. Interestingly, another parallel work to SPO on extending DACE (or EGO) to noisy functions, sequential Kriging optimization (SKO) [109], which does not use repeated evaluation but chooses candidate configuration that maximizes its expected performance minus one standard deviation. SKO is however confirmed to perform worse than all SPO variants in the algorithm configuration problem [117]. This line of research has also showcased that the mechanism for stochasticity handling, which has been usually understated in the literature, plays an important role for algorithm configuration.

SPO+ is further investigated in [116] for algorithm configuration problems with a configuration budget defined by a time bound instead of a maximum number of algorithm evaluations (TB-SPO). The configuration goal is to minimize the computation time of the target algorithm. Hence, reducing the time complexity of the configurator becomes crucial. To this end, two main contributions for such configuration scenarios are proposed. Firstly, the initial latin hypercube sampling in SPO+, which takes half of the configuration budget, is removed. To ensure sufficient exploration in the search of the parameter space, one uniformly random candidate configuration is generated in each SPO+ iteration and compared to the incumbent. Secondly, the computationally time consuming Gaussian Process model is replaced by a drastically faster projected process approximation of the standard Gaussian Process model.

This line of research for SPO has one practical drawback for the algorithm configuration problem and remains as a proof-of-concept: these SPO-based configurators are only applicable to tuning on one single problem instance. The reason for this consists in combining the Gaussian process search method with the intensification mechanism as an evaluation method. Both the intensification mechanism in the original SPO 0.3 or 0.4 implementation, and the intensification in SPO+ or TB-SPO, evaluate candidate configurations with different numbers of algorithm evaluations, and average their evaluation responses for building the prediction model. However, if different configurations are evaluated on different instances, how to build the model based on responses from different instances is not straightforward.

GGA

An evolutionary computation based approach, the gender-based genetic algorithm (GGA) [5], is also proposed for algorithm configuration. The population is divided into two "genders", a competitive gender C and a non-competitive gender N. The crossover is performed on two individuals of different genders: one from a small proportion of the best individuals of C, and one taken uniformly at random from N. With a small probability, the mutation is performed either by randomly selecting a value for a categorical parameter, or generating a value from the current value by a Gaussian distribution.

As described in the last paragraph [5, page 151], GGA uses a similar stochasticity handling technique as SPO. It also includes the best-so-far configuration into the next iteration, adopts the repeated evaluation as the evaluation method in each iteration, and linearly increases the number of repetitions nr from iteration to iteration. The increase of nr

ensures that the later iteration of GGA, where the performance of candidate configurations is usually closer, receives more training instances for evaluation and thus increases its evaluation accuracy.

SMAC

Sequential Model-based Algorithm Configuration (SMAC) [112] extends the line of research on SPO to be applicable to the general algorithm configuration problem. Similar to SPO+ [117], SMAC has adopted the state-of-the-art evaluation method known as the intensification method as in FocusedILS [118] (also mentioned as aggressive racing in [112]). In order to be able to configure categorical parameters, the Kriging model used in SPO+ is replaced by the random forests [40], so that both continuous and categorical parameters can be modelled. Besides, in order to configure algorithms across the different instances in a training set, a number of features can be specified for each instance, and these instance features are used together with the parameter values as input features for the random forest model. SMAC uses the random forest model to predict the expected performance of newly generated configurations on a given instance. In each iteration, SMAC applies a multi-start local search in the parameter space, and evaluates each configuration by its expected improvement over the incumbent, i.e., its probability to outperform the incumbent. The best configuration with the maximum expected improvement value will be compared with the incumbent by running the target algorithm using the intensification mechanism. SMAC has been experimentally shown to perform better than FocusedILS and GGA across different algorithm configuration benchmarks [112], and is regarded as one of the state-of-the-art algorithm configurators.

3.4.3 Applications of F-Race

Throughout the thesis, we have applied F-Race as an evaluation method inside our algorithm configurator. F-Race has received significant attention since it was first proposed in 2002 [36], which has received the 2012 SIGEVO Impact Award. This article has received 557 citation as of April 2019 according to Google Scholar. By the time this PhD work started in June 2009, it has been received 99 citations in the Google scholar database. In what follows, we give an overview of the early researches that applied F-Race before June 2009.

Fine-tuning algorithms.

The by far most common use of F-Race is to use it as a method to fine-tune an existing or a recently developed algorithm. Often, the tuning through F-Race is also done before comparing the performance of various algorithms. In fact, this latter usage is important to make reasonably sure that performance differences between algorithms are not simply due to an uneven configuration effort.

A significant fraction of the usages of F-Race is due to researchers either involved in the development of the F-Race method or by their collaborators. In fact, F-Race has been developed in the research for the Metaheuristics Network, an EU-funded research and training network on the study of metaheuristics. Various applications there have been for configuring different metaheuristics for the university-course timetabling problem [50, 196] and for various other problems [48, 51, 62, 195, 200].

Soon after these initial applications, F-Race has been adopted by a number of other researchers. Most applications focus on configuring SLS methods for combinatorial optimization problems [30, 15, 64, 89, 141, 184, 186]. However, also other applications have been considered including the tuning of algorithms for training neural networks [38, 206], or the tuning of parameters of a control system for simple robots [178, 179].

Industrial applications.

Few researches have evaluated F-Race in pilot studies for industrial applications. The first has been a feasibility study, where F-Race was used to configure a commercial solver for vehicle routing and scheduling problems, which has been developed by the software company SAP. In this research, six configuration tasks have been considered that ranged from the study of specific parameters, which determined the frequency of the application of some important operators of the program, to the configuration of the SLS method that was used in the software package. F-Race was compared to a strategy that after each fixed number of instances discarded a fixed percentage of the worst candidate configurations, showing, as expected, advantages for F-Race when the performance differences between configurations were stronger. Some results of this study have been published in [26]; more details are available in a master thesis [25].

Yuan et al. [234] have adopted F-Race to configure several algorithms for a highly constrained train scheduling problem arising at

Deutsche Bahn AG. A comparison of various configured algorithms identified an iterated greedy algorithm as the most promising one.

Algorithm development.

F-Race has occasionally been used to explicitly support the algorithm development process. A first example is described by Chiarandini et al. [49] who used F-Race to design a hybrid metaheuristic for the university-course timetabling problem. In their work, they have adopted F-Race in a semi-automatic way. They observed the algorithm candidates that were maintained in a race and based on this information they generated new algorithm candidates that were then manually added to the ongoing race. In fact, one of these newly injected candidate algorithms was finally the best performing algorithm in an international timetabling competition (see also `http://www.idsia.ch/Files/ttcomp2002`).

The PhD work of Matthijs den Besten [62] provides an empirical investigation into the application of ILS to solve a range of deterministic scheduling problems with tardiness penalties. Racing, in general, and F-Race, in particular, is a very important ingredient throughout the algorithm development and calibration. The ILS algorithms are built in a modular way and F-Race is applied to assess each combination of modular components of the algorithm.

Comparison of F-Race to other methods

There have been some comparisons of F-Race with other racing algorithms. Some preliminary results comparing F-Race and t-test based racing techniques are presented by Birattari [32, 33], showing that F-Race typically performs best.

Yuan and Gallagher [230] discuss the use of F-Race for the empirical evaluation of evolutionary algorithms. They also use an algorithm called *A-Race*, where the family-wise test is based on the *analysis of variance* (ANOVA) method. From the experiments they conduct, they conclude that their version of an F-Race obtains better results than *A-Race*.

Caelen and Bontempi in their work [44] compare five techniques from various communities on a model selection task. The techniques compared are (i) a two-stage selection technique proposed in the stochastic simulation community, (ii) a stochastic dynamic programming approach conceived to address the multi-armed bandit problem,

(iii) a racing method, (iv) a greedy approach, (v) a round-search technique. F-Race is mentioned and applied for comparison purposes. The comparison results shows that the bandit strategy yields the most promising performance when the sample size is small, but F-Race outperforms other techniques when the sample size is sufficiently large.

Extensions and Hybrids of F-Race.

The F-Race algorithm has been adopted as a module integrated into an ACO algorithm framework for tackling combinatorial optimization problems under uncertainty [34]. The resulting algorithm is called ACO/F-Race and it uses F-Race to determine the best of a set of candidate solutions generated by the ACO algorithm. In later work by Balaprakash et al. [16] on the application of estimation-based ACO algorithms to the probabilistic traveling salesman problem the Friedman test is replaced by an ANOVA.

Yuan and Gallagher [231, 232] propose an approach to tune evolutionary algorithms by hybridizing Meta-EA and F-Race. Meta-EA is an approach that uses various genetic operators to tune the parameters of EAs. It is reported that one major difficulty in Meta-EA is that it cannot handle effectively categorical parameters. These categorical parameters are usually handled in Meta-EA by pure random search. The proposed hybridization uses Meta-EA to evolve part of the numerical parameters and leave the categorical parameters for F-Race. Experiments show that Meta-EA plus F-Race required only around 12% of the computational effort taken by Meta-EA.

Up to June 2009, it is shown in the literature that F-Race (or iterated F-Race) has been applied to tuning maximum 11 parameters; however, this does not impose a limit for F-Race or iterated F-Race. It is also shown that most of the F-Race applications up to then obtain the initial set of parameter settings for F-Race by a full factorial design. Usually not only the categorical parameters are treated by full factorial design, but also the numerical parameters are first discretized and then extracted by full factorial design. The sampling design methods for F-Race discussed in this article, more specifically the iterated F-Race, will be substantially useful for the conventional F-Race users.

3.5 Summary

The algorithm configuration problem can be regarded as a black-box, mixed-variable, stochastic optimization problem. There are two sources

of stochasticity in the configuration problem: the randomness of the algorithm, and the random sampling of the training instances. A configuration algorithm that tackles the configuration problem usually contain two components: a search method that generates candidate configurations from the black-box search space, and an evaluation method that evaluates generated candidate configurations under stochasticity and selects the best one(s) therefrom. Most of the existing research on automated algorithm configuration focuses on the search aspect. The evaluation aspect is relatively less studied. Given a fixed evaluation budget, a good configuration algorithm should balance the trade-off of exploration that generates more candidate configurations and exploitation that evaluates carefully on generated configurations. We provide two frameworks for combining search method and evaluation method to form high-performing configuration algorithm. One is the iterated selection, which iteratively uses search methods to generate new configurations, then applies evaluation methods to evaluate and select the best from the generated ones, and uses the evaluation results to bias the further search. The other is the post-selection, which divides the configuration process into two phases. The first phase of elite qualification tries to identify a set of elite configurations, which are then carefully evaluated and selected in the second phase of elite selection. We further classify that all the established configuration algorithms, such as CALIBRA, iterated racing, ParamILS, SPO, GGA, SMAC belong to the iterated selection framework. In the rest of the thesis, we will go into more details about developing high-performing configuration algorithms, following both the iterated selection and post-selection frameworks.

Chapter 4

Iterated selection using F-Race: Iterated F-Race and beyond

In this chapter, we study algorithm configuration approaches that use iterated selection and F-Race.

F-Race, or more generally the racing method as introduced in Chapter 3, is an effective evaluation method for selecting the best from an initial set of candidate parameter configurations. In the previous sections, the question, how the set of candidate parameter configurations can be generated and updated, was left open. This is the question we address in this chapter. More specifically, we study an iterated selection approach to iteratively generate candidate parameter configurations, and use F-Race to rank and select the best from the generated parameter configurations in each iteration. In a next step, we bias the further search by the selected best configurations. In the following, we will dive deeper into the idea of iterated selection using F-Race, or iterated F-Race. We first review the previous strategies of generating candidate configurations for F-Race in Section 4.1; then introduce the framework of iterated F-Race and provide an example implementation in Section 4.2, and experimentally validate the iterated F-Race for configuring numerical parameters and categorical parameters in Section 4.3 and 4.4, respectively; a second example of iterated F-Race by hybridizing F-Race with an established continuous optimization algorithm mesh adaptive direct search, dubbed MADS/F-Race, is presented in Section 4.5; Section 4.6 concludes the chapter and provides outlook.

4.1 Previous strategies for generating configurations for F-Race

F-Race was first proposed in the seminal work done by Birattari et al. [36] in 2002. In this work, F-Race is used to select the best parameter configuration from a given set of initial candidate configurations. A simple and straightforward way to come up with a set of initial candidate configurations is by conducting a full factorial design, which is described in Section 4.1.1. Balaprakash et al. [14] considered instead using sampling techniques to generate numerical parameter configurations for F-Race. These sampling techniques include a random sampling design (Section 4.1.2), and an ad-hoc iterative sampling design, which will be discussed in more detail in Section 4.2. In this section, we will first have an overview of these previous strategies for generating parameter configurations for F-Race.

4.1.1 Full factorial design

A full factorial design can be done by determining for each parameter a number of levels, typically manually by expertise or by equidistant levels. Then, each possible combination of these levels represents a unique configuration and the set of initial configurations contains all these combinations.

One main drawback of a full factorial design is that it requires expertise to select the levels of each parameter. Maybe more importantly, the set of candidate configurations grows exponentially with the number of parameters. Suppose that d is the dimension of the parameter space and that each dimension has l levels; then the total number of candidate configurations would be l^d. It therefore becomes quickly impractical and computationally prohibitive to test all possible combinations even for a reasonable number of levels at each dimension.

4.1.2 Random sampling design

The drawbacks of using the full factorial design were pointed out also by Balaprakash et al. [14]. It was shown that F-Race with initial candidate configurations generated by a random sampling design significantly outperforms the full factorial design for a number of applications. In the random sampling design, the initial elements are sampled according to some probability model P_X defined over the parameter space X.[1] If *a priori* information is available, such as the effects of

[1] Note that the space of possible parameter value combinations X is different from the one-dimensional vector of candidate algorithm configurations Θ, and there exists a

certain parameters or their interactions, the probability model P_X can be defined accordingly. However, this is rarely the case, and the default way of defining the probability model P_X is to assume a uniform distribution over X. In what follows, we denote the uniform random sampling strategy for generating candidate configurations for F-Race as U/F-Race.

Two main advantages of the random sampling design are that for numerical parameters, no *a priori* definition of the levels needs to be done and that an arbitrary number of candidate configurations can be sampled while still covering the parameter space, on average, uniformly.

4.2 Iterated F-Race

As a next step, Balaprakash et al. [14] proposed an iterative sampling strategy for F-Race. The resulted tuner, named I/F-Race, uses F-Race at each iteration to rank and select the best candidates. The selected best candidates of the current iteration is used to bias the sampling of new candidate configurations in the next iteration, so as to focus the sampling of candidate configurations around the most promising ones.

However, this version of I/F-Race proposed in 2007 was only able to handle numerical parameters. It was further refined and developed into a fully functioning tuner [37], which not only improves its performance for tuning numerical parameters, but also extends it to handle categorical parameters and conditional parameters.

4.2.1 The framework of iterated F-Race

Iterated F-Race can be regarded as a type of iterated selection using F-Race as its evaluation method. Its basic idea is to proceed the tuning process in a number of iterations. In each iteration, first a set of candidate configurations is sampled according to a model. This is followed by one run of F-Race to rank and select the elite configurations from the sampled candidate configurations. Then the model is updated based on the elite configurations and the next iteration starts. An outline of the general framework of iterated F-Race is given in Algorithm 4.1.

There are many possible ways how iterated F-Race can be implemented. In fact, one possibility would be to use some black-box search algorithms for generating candidate configurations, which will be detailed in Chapter 5. However, a difficulty here may be that for F-Race to be effective, the number of candidate configurations should be reasonably large. Besides, due to the necessarily strongly limited number

one-to-one mapping from X to Θ.

Algorithm 4.1 Iterated F-Race

Require: parameter space X, a stochastic black-box objective function f.
  initialize probability model P_X for sampling from X
  set iteration counter $l = 1$, set initial elite configurations $\Theta_*^0 = \emptyset$
  repeat
    sample the set of configurations Θ_0^l of iteration l based on P_X
    include elite configurations $\Theta_0^l = \Theta_0^l \bigcup \Theta_*^{l-1}$
    rank and select elite configurations Θ_*^l from Θ_0^l by f using F-Race
    update P_X based on selected elite configurations Θ_*^l
    $l = l + 1$
  until termination criterion is met
  return the best configuration $x^* \in \Theta_*^l$

of *function evaluations*, and in order not to challenge the incumbent too often, few iterations should be run. Therefore, a different approach was followed and an ad-hoc method was proposed for biasing the sampling of numerical parameters [14]. Nevertheless, the idea presented there can be generalized to categorical parameters. In what follows, we first give a general discussion of the issues that arise in the definition of an iterated F-Race algorithm and then we present one particular implementation in Section 4.2.2. For the following discussion, we assume that the total computational budget B for the configuration process, which is measured by the number of function evaluations, is given *a priori*.

How many iterations? Iterated F-Race is an iterative process and therefore one needs to define the number of iterations. For a given computational budget, using few iterations will allow to sample at each iteration more candidate configurations and, hence, lead to more exploration at the cost of less possibilities of refining the model. In the extreme case of using only one iteration, this amounts to an execution of U/F-Race. Intuitively, the number of iterations should depend on the number of parameters: if only few parameters are present, we expect, others things being equal, the problem to be less difficult to optimize and, hence, less iterations to be required.

Which computational budget at each iteration? Another issue concerns the distribution of the computational budget B among the iterations. The simplest idea is to divide the computational budget equally among all iterations. However, other possibilities are certainly reasonable; for example, one may decrease the number of function evaluations available with an increase of the

iteration counter to increase exploration in the first iterations.

How many candidate configurations at each iteration? For F-Race, the number of candidate configurations to be sampled needs to be defined. A good idea is to make the number of candidate configurations dependent on the status of the race, in other words, the iteration counter. Typically, in the first iteration(s), the sampled candidate configurations are very different from each other, resulting in large performance differences. As a side effect, poor candidate configurations usually can be quickly eliminated. In later iterations, the sampled candidate configurations become more similar and it becomes more difficult to determine the winner, that is, more instances are needed to detect significant differences among the configurations. Hence, for a same budget of function evaluations for one application of F-Race, in early iterations more configurations can be sampled, while in later iterations less candidate configurations should be generated to identify with a low variance a winning configuration.

When to terminate F-Race at each iteration? At each iteration l, F-Race terminates if one of the following two conditions is satisfied: (i) if the computational budget for the l-th iteration, B_l, is spent; (ii) when a minimum number of candidate configurations, denoted by N_{min}, remains. Another question concerns the value of N_{min}. F-Race terminates by default if a unique survivor is identified. However, to maintain sufficient exploration of the parameter space, in iterated F-Race it may be better to keep a number of survivors at each iteration and to sample around these survivors the candidate configurations for the next iteration. Additionally, for setting N_{min}, it may be a good idea to take into account the number of parameters: the larger the parameter space, the more survivors should remain to ensure sufficient exploration.

How should the candidate configurations be generated? In an iterated sampling approach, candidate configurations of each iteration are randomly sampled in the parameter space according to some probability distribution. For continuous and quasi-continuous parameters, continuous probability distributions are appropriate; for categorical and ordinal parameters, however, discrete probability distributions will be more useful. A first question related to the probability distributions is of which type they should be. For example, in the first paper on iterated F-Race [14], normal distributions were chosen as models, but this choice need not be optimal. Another question related to the probability

distributions is how they should be updated and, especially, how strong the bias towards the surviving configurations of the current iteration should be. Again, here the trade-off between exploration and exploitation needs to be taken into account.

4.2.2 I/F-Race: An example iterated F-Race algorithm

Here we describe one example implementation of iterated F-Race, to which we refer as I/F-Race in the following. This example implementation is extended from the previous one published by Balaprakash et al. in 2007 [14]. However, it differs in a number of perspectives. Firstly, it extended the first version by defining a way to handle categorical parameters and conditional parameters. Secondly, a number of modifications in the settings has been made, including a more reasonable resample mechanism and more efficient budget usage. Thirdly, it differs in some parameter choices such that the setting is more reasonable when the number of parameters to be tuned is large. Note that the proposed parameter settings are still chosen in an ad-hoc fashion; tuning the parameter settings of I/F-Race is beyond the scope here.

Number of iterations. We denote by L the *number of iterations* of I/F-Race, and increase L with d, the number of parameters, using a setting of $L = 2 + round(\log_2 d)$. L increases as more parameters are there to be tuned, ranging from three iterations for two parameters, to ten iterations with 256 parameters to be tuned.

Computational budget at each iteration. The *computational budget* is distributed as equally as possible across the iterations. B_l, the computational budget in iteration l, where $l = 1, ..., L$, is set to $B_l = (B - B_l^{used})/(L - l + 1)$; B_{l-1}^{used} denotes the total computational budget used until iteration $l - 1$, initialized as $B_0^{used} = 0$. If the last iteration finishes with sufficient unused budget left, a new iteration will start with the remaining budget.

The number of candidate configurations. The number of candidate configurations N_l at each iteration $l \in \{1, ..., L\}$ depends on the budget B_l. We introduce a parameter μ_l, and set $N_l = \lfloor B_l/\mu_l \rfloor$. We let μ_l increase with the number of iterations, using a setting of $\mu_l = 5 + l$. This allows more evaluation steps to identify the winners when the configurations are deemed to become more similar.

Termination of F-Race at each iteration. In addition to the usual termination criteria of F-Race, we stop it if at most $N_{min} =$

$2+round(\log_2 d)$ candidate configurations remain, such that more survivors are allowed for larger parameter space.

Generation of candidate configurations. In the first iteration, all candidate configurations are sampled uniformly at random. Once F-Race terminates, the best N_s candidate configurations are selected for the update of the probability model for the next iteration. We use $N_s = \min(N_{survive}, N_{min})$, where $N_{survive}$ denotes the number of candidates that survive the race. These N_s elite configurations are then ranked, and weighted according to their ranks, where the weight of an elite configuration with rank r_z ($z = 1, \ldots, N_s$) is given by:

$$w_z = \frac{N_s - r_z + 1}{N_s \cdot (N_s + 1)/2}. \tag{4.1}$$

In other words, the weight of an elite configuration is inversely proportional to its rank. Since the training instances are picked randomly from the training set and change from iteration to iteration, the N_s elite configurations of the lth iteration will be re-evaluated in the $(l+1)$st iteration, together with the $N_{l+1} - N_s$ candidate configurations to be sampled anew. (Alternatively, it is possible to evaluate the configurations on fixed instances, so that the results of the elite configurations from the last iteration could be reused; more results on this topic are shown in Section 4.3.2.) The $N_{l+1} - N_s$ new candidate configurations are iteratively sampled around one of the elite configurations. To do so, for sampling each new candidate configuration, first one elite configuration z ($z \in \{1, \ldots, N_s\}$) is chosen with a probability proportional to its weight w_z and next a value is sampled for each parameter dimension. The sampling distribution of each parameter depends on whether it is a numerical one (the set of such parameters is denoted by X^{num}) or a categorical one (the set of such parameters is denoted by X^{cat}). We have that the parameter space $X = X^{num} \cup X^{cat}$.

First suppose that X_i, i.e., the i-th parameter dimension of X, is a numerical parameter, i.e. $X_i \in X^{num}$, with boundary $X_i \in [\underline{X}_i, \overline{X}_i]$. Denote $v_i = \overline{X}_i - \underline{X}_i$ the range of the parameter X_i, and denote x_i^z the value of the ith parameter of configuration z. The sampling distribution of X_i follows a normal distribution $N(x_i^z, \sigma_i^l)$, with x_i^z being the mean and σ_i^l being the standard deviation of X_i in the lth iteration. The standard deviation is reduced in a geometric fashion from iteration to iteration using a setting of

$$\sigma_i^{l+1} = v_i \cdot \left(\frac{1}{N_{l+1}}\right)^{\frac{l}{d}} \qquad \text{for } l = 1, \ldots, L-1. \tag{4.2}$$

In other words, the standard deviation for the normal distribution is reduced by a factor of $\left(\frac{1}{N_{l+1}}\right)^{\frac{1}{d}}$ as the iteration counter increments. Hence, the more parameters, the smaller the update factor becomes, resulting in a stronger bias of the elite configuration on the sampling. Furthermore, the larger the number of candidate configurations to be sampled, the stronger the bias of the sampling distribution.

Now, suppose that $X_i \in X^{cat}$ with n_i levels $F_i = f_1, \ldots, f_{n_i}$. Then we use a discrete probability distribution $P_l(F_i)$ with iteration $l = 1, \ldots, L$, and initialize P_1 to be uniformly distributed over F_i. Suppose further that after the l-th iteration ($l > 1$), the ith parameter of the selected elite configuration z takes level f_i^z. Then, the discrete distribution of parameter X_i is updated as:

$$P_{l+1}(f_j) = P_l(f_j) \cdot (1 - \frac{l}{L}) + I_{j=f_i^z} \cdot \frac{l}{L} \quad \text{for } l = 1, \ldots, L-1 \text{ and } j = 1, \ldots, n_i \tag{4.3}$$

where I is an indicator function; the bias of the elite configuration on the sampling distribution is getting stronger as the iteration counter increments.

The conditional parameters are sampled only when they are activated by their associated upper-level categorical parameter; and their sampling model is updated only when they appear in elite configurations.

4.3 Case studies: I/F-Race for numerical parameters

4.3.1 Comparing I/F-Race with previous sampling mechanisms for F-Race

As I/F-Race was first proposed in 2007 [14] (dubbed IF2007 in this section), it was only designed for tuning numerical parameters, i.e. real-valued or large-domain integer parameters. Its tuning performance has been compared with other way of generating candidate configurations

for F-Race, such as uniform random sampling (U/F-Race) or full factorial design (FFD/F-Race). Extensive experiments have proved that IF2007 performs better than U/F-Race and FFD/F-Race.

The I/F-Race introduced in Section 4.2.2 and [37] (dubbed IF2010 in this section) not only extends IF2007 by enabling it to tune categorical and conditional parameters, but it also differs from IF2007 in its tuning mechanisms for the numerical parameters. More specifically, IF2010 adopts:

- a more efficient use of available tuning budget: the unused budget of previous iteration will be able to be used in later iterations, and extra iterations will be launched if there is budget left after I/F-Race finishes.
- a resample mechanism:
  - if a candidate configuration is sampled outside the parameter boundary, it will be resampled instead of taking the border value; this is equivalent to sampling by a truncated normal distribution.
  - If two same candidate configurations are sampled, resample.
  - If a maximum number (by default ten) of resample is called without sampling enough different candidate configurations, soften the sampling probability so as to allow more candidates to be sampled.

Here, the IF2010 and IF2007 are compared in tuning seven numerical parameters of $\mathcal{MM}$AS by three different tuning budget and two different computation time for running $\mathcal{MM}$AS (5 and 20 seconds). As a baseline, we also include the U/F-Race. The comparison results are shown in Figure 4.1, and the magnitude difference and statistical significance are listed in Table 4.12. As can be observed, in 4 out of 6 cases, IF2010 significantly outperforms IF2007; only in the two least budgets of the 5-second cases, the difference is too small to reach statistical significance. Summing up from these six case studies, IF2010 outperforms IF2007 in most of the case studies, and is never significantly worse.

4.3.2 Fixed instance order and common random seed

The early version of iterated F-Race [37] adopted a shuffled instance order, i.e., the order of the tuning instances changes from iteration to iteration. Hence, from the second iteration on, the elite parameter configurations evaluated from the previous iterations will be evaluated on new tuning instances from scratch. The reason for shuffling the

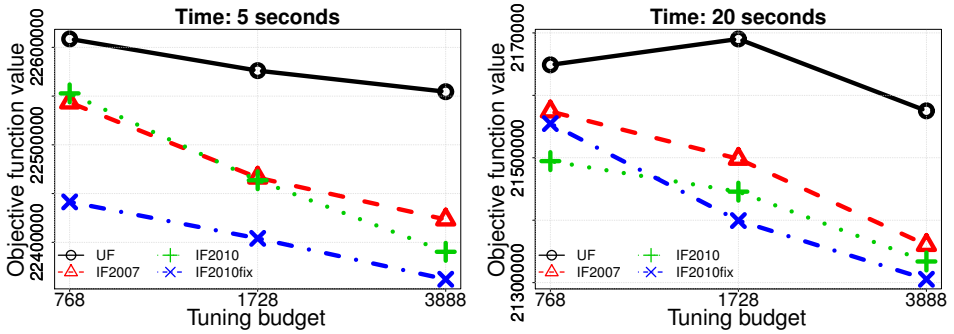


Figure 4.1: The comparison of the mean performance of four tuners: (i) the uniform random sampling F-Race (dubbed UF); (ii) 2007 version of I/F-Race (dubbed IF2007 [14]); (iii) 2010 version of I/F-Race (dubbed IF2010 [37]), (iv) and the 2010 version of I/F-Race with fixed instance order and common random seed (IF2010fix [37], see Section 4.3.2) for tuning $\mathcal{MM}$AS for the TSP with seven numerical parameters under computation time 5 seconds (left) and 20 seconds (right).

tuning instances at each iteration is to avoid the risk of over-tuning by forcing the elite parameter configurations to be evaluated on more instances, and bring them to a "fair" start together with the other newly generated parameter configurations. However, the disadvantage of shuffling the training instances is a waste of computation resources: the previous evaluations of the elite parameter configurations are not utilized for their evaluations in the latter iterations. Besides, the previous evaluations of the elite parameter configurations have proved their goodness. Racing them iteratively again and again on new instances ignoring their existing evidence of goodness increases the probability of wrongly rejecting them.

Using a fixed instance order allows each iteration of F-Race to evaluate candidate configurations on the same instances on the same order. Then we can store historical evaluation results in an archive to avoid re-evaluating a same parameter configuration on a same instance. Thereby, the elite configurations that survives from a previous racing iteration can enter the next racing iteration with its archived evaluations without being re-evaluated. Besides, if the same parameter configuration is sampled again, its results on the evaluated instances can be read from the archive instead of rerunning the optimization algorithm. Therefore, adopting a fixed instance order leads to considerable saving of unnecessary re-evaluation comparing with shuffled instance order. The saving becomes bigger as more configuration budget is available.

To further reduce the experimental variance, we adopted the technique of common random seed. Using common random seed is a stan-

Table 4.1: Comparison of four tuners: (i) the uniform random sampling F-Race (dubbed U/F-Race); (ii) 2007 version of I/F-Race (dubbed IF2007 [14]); (iii) 2010 version of I/F-Race (dubbed IF2010 [37]), (iv) and the 2010 version of I/F-Race with fixed instance order and common random seed (dubbed IF2010fix [37], see Section 4.3.2) for tuning $\mathcal{MM}$AS for the TSP with seven numerical parameters for a computation time bound of 5 and 20 seconds, respectively. The column entries with the label `per.dev` shows the mean percentage deviation of each algorithm from the reference cost. $+x$ ($-x$) means that the solution cost of the algorithm is x% more (less) than the reference cost. The column with the label `max.bud` gives the maximum number of evaluations given to each tuner. The pairwise Wilcoxon signed rank test with Holm's adjustment for multiple comparisons is applied. The statistically best tuner in each computation time with each configuration budget is marked with boldface; and the tuner that statistically outperforms IF2007 is marked with *.

	5 seconds	20 seconds	
tuner	per.dev	per.dev	max.bud
U/F-Race	+0.32	+0.37	768
IF2007	−0.03	+0.03	768
IF2010	−0.07	**−0.34***	768
IF2010fix	**−0.42***	−0.06*	768
U/F-Race	+0.44	+0.85	1 728
IF2007	−0.05	−0.05	1 728
IF2010	−0.06	−0.29*	1 728
IF2010fix	**−0.33***	**−0.51***	1 728
U/F-Race	+0.54	+0.85	3 888
IF2007	−0.14	−0.16	3 888
IF2010	−0.19*	−0.28*	3 888
IF2010fix	**−0.31***	**−0.41***	3 888

* Significantly better than IF2007 according to pairwise Wilcoxon signed rank test with Holm's adjustment for multiple comparisons.

dard technique for reducing variance in simulation [227, 132, 139]. In 1992, McGeoch [162] introduced the use of common random number as a variance reduction technique for algorithm analysis. In our experiments, each instance is assigned a random number, which serves as the common random seed each time this instance is used for evaluation of any parameter configuration.

In the following, we assess the use of fixed instance order and com-

mon random seed in the latest I/F-Race [37], dubbed IF2010fix. The results are shown in Figure 4.1 and Table 4.12. We observed that IF2010fix outperforms the I/F-Race with shuffled instances (IF2010) in five out of six case studies. IF2010fix is only outperformed in the 20-second case with the minimum budget (768), where the budget is too small to sample repeated parameter configurations; thus, the saving in re-evaluation becomes less significant. All differences between IF2010 and IF2010fix are statistically significant by the pairwise Wilcoxon signed rank test with Holm's adjustment for multiple comparisons. Besides, IF2010fix significantly outperforms IF2007 in all the six case studies.

Based on the experimental results presented in this section, we have adopted the IF2010fix as our I/F-Race implementation throughout this thesis.

4.4 Case studies: I/F-Race for categorical and conditional parameters

In this section, we experimentally evaluate the presented variant of I/F-Race and we compare it in three case studies to U/F-Race and *F-Race(FFD)*.

All three case studies concern the configuration of ant colony optimization (ACO) algorithms applied to the traveling salesman problem (TSP). They are ordered according to the number of parameters to be tuned. In particular, they involve configuring $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS), a particularly successful ACO algorithm [213], using four categorical parameters and configuring $\mathcal{MM}$AS using seven categorical parameters. Both case studies use the $\mathcal{MM}$AS implementation available in the ACOTSP[2] software package, which implements several ACO algorithms for the TSP. The third case study uses the ACOTSP package as a black-box software and involves setting 12 mixed parameters. Among others, one of these parameters is the choice of which ACO algorithm should be used.

In all experiments we used Euclidean TSP instances with 750 nodes, where the nodes are uniformly distributed in a square of side length 10 000. We generated 1 000 instances for training and 300 for evaluating the winning configurations using the DIMACS instance generator [121]. The experiments were carried out on cluster computing nodes, each equipped with two quad-core XEON E5410 CPUs running at 2.33

[2]The ACOTSP package is available at `http://www.aco-metaheuristic.org/aco-code/`.)

GHz with 2×6 MB second level cache and 8 GB RAM. The cluster was running under Cluster Rocks Linux version 4.2.1/CentOS 4. The programme was compiled with gcc-3.4.6-3, and only one CPU core was used for each run due to the sequential implementation of the ACOTSP software.

For each case study, we have run a total of six experiments, which result by all six combinations of two different computation time limits allocated for each `function evaluation` to the ACOTSP software (five and twenty CPU seconds) and three values for the computational budget. The different levels of the computational budget have been chosen to examine the dependence of the possible advantage of I/F-Race as a function of the corresponding computational budget.

In each of the six experiments, 10 `trials` were run. Each `trial` is the execution of the `configuration process` (in our case, either *F-Race(FFD)*, U/F-Race, or I/F-Race) together with a subsequent `testing procedure`. In the testing procedure, the final parameter setting returned by configuration process is evaluated on 300 test instances.

4.4.1 Case study 1, $\mathcal{MM}$AS under four parameters

In this case study, we tune four parameters of $\mathcal{MM}$AS: the relative influence of pheromone trails α; the relative influence of heuristic information β; the pheromone evaporation rate ρ; and the number of ants, m.

In this first and the second case study, we discretize these numerical parameters and treat them as categorical ones. Each parameter is discretized by regular grids, resulting in a relatively large number of levels. Their ranges and number of levels as listed in Table 4.2.[3] The motivation for discretizing numerical parameters is to test whether I/F-Race is able to improve over U/F-Race and *F-Race(FFD)* for categorical parameters; previously, it was already shown that I/F-Race gives advantages for numerical parameters [14].

The three levels of the computational budget chosen are $6 \cdot 3^4 = 486$, $6 \cdot 4^4 = 1\,536$ and $6 \cdot 5^4 = 3\,750$. In this way the candidate generation of *F-Race(FFD)* can be done by selecting the same number of levels for each parameter, in our case three, four, and five. Without a priori knowledge, the level of each parameter is selected randomly in *F-Race(FFD)*.

The experimental results are given in Table 4.3. The table shows

[3]For the other parameters, we use default values and we opted for an ACO version that does not use local search.

Table 4.2: Given are the parameters, the original range considered before discretization and the number of levels considered after discretization for the first case study. The number of candidate parameter settings is 12 100.

parameter	range	# levels
α	$[0.01, 5.00]$	11
β	$[0.01, 10.00]$	11
ρ	$[0.00, 1.00]$	10
m	$[5, 100]$	10

the average percentage deviation of each algorithm from the reference cost, which for each instance is defined by the average cost across all candidate algorithms on that instance. The results of the algorithms tuned by *F-Race(FFD)*, U/F-Race, and I/F-Race, are compared using the non-parametric pairwise Wilcoxon test with Holm adjustment, using blocking on the instances; the significance level chosen is 0.05. Results in boldface indicate that the corresponding configurations are statistically better than the ones of the two competitors.

In all experiments, I/F-Race and U/F-Race significantly outperform *F-Race(FFD)*. Overall, I/F-Race has a slight advantage over U/F-Race: in three of six experiments I/F-Race returns configurations that are significantly better than those found by U/F-Race, while the opposite is true on only one experiment. The trend appears to be that with larger total budget, the advantage of I/F-Race over U/F-Race increases. The reason for the relatively good performance of U/F-Race could be due to the fact that the parameter space is rather small (12100 candidate configurations) and that the number of levels (10 or 11) for each parameter is large.

4.4.2 Case study 2, $\mathcal{MM}$AS under seven parameters

In this case study, we have chosen seven parameters. These are the same as in the first case study plus three additional parameters: γ, a parameter that controls the gap between the minimum and maximum pheromone trail value in $\mathcal{MM}$AS, $\gamma = \tau_{max}/(\tau_{min} \cdot instance\_size)$; nn, the number of nearest neighbors used in the solution construction phase; and q_0, the probability of selecting the best neighbor deterministically in the pseudo-random proportional action choice rule; for a detailed definition see [71].

The parameters are discretized using the ranges and number of levels given in Table 4.4. Note that in comparison to the previous ex-

Table 4.3: Computational results for configuring $\mathcal{MM}$AS for the TSP with 4 discretized parameters for a computation time bound of 5 and 20 seconds, respectively. The column entries with the label `per.dev` shows the mean percentage deviation of each algorithm from the reference cost. $+x$ ($-x$) means that the solution cost of the algorithm is x% more (less) than the reference cost. The column with the label `max.bud` gives the maximum number of evaluations given to each algorithm.

	5 seconds	20 seconds	
algo	per.dev	per.dev	max.bud
F-Race(FFD)	+0.85	+0.79	486
U/F-Race	**−0.58**	−0.44	486
I/F-Race	−0.26	−0.34	486
F-Race(FFD)	+0.51	+1.27	1 536
U/F-Race	−0.08	−0.66	1 536
I/F-Race	**−0.42**	−0.61	1 536
F-Race(FFD)	+0.40	+0.71	3 750
U/F-Race	−0.12	−0.27	3 750
I/F-Race	**−0.28**	**−0.45**	3 750

periment, the parameter space is more than one order of magnitude larger ($259\,200 \gg 12\,100$). Besides, there is smaller number of levels for each parameter, usually between four to nine. We use the same experimental setup as in the previous section, except that for the computational budget, we choose $6 \cdot 2^7 = 768$ such that each parameter in *F-Race(FFD)* has two levels, $6 \cdot 2^5 \cdot 3^2 = 1\,728$, such that in *F-Race(FFD)* five parameters will have two levels and the other two three levels, and $6 \cdot 2^3 \cdot 3^4 = 3\,888$, such that in *F-Race(FFD)*, three parameters will have two levels, and the other four parameters have three levels.

The experimental results are listed in Table 4.5 and the results are analyzed in a way analogous to case study 1. The results clearly show that I/F-Race significantly outperforms *F-Race(FFD)* and U/F-Race in each experiment. As expected, also U/F-Race outperforms *F-Race(FFD)* significantly.

4.4.3 Case study 3, ACOTSP under twelve parameters

In a final experiment, 12 parameters of the ACOTSP software are examined. This configuration task is the most complex and it requires the setting of categorical as well as numerical parameters.

Among these parameters, firstly two categorical parameters have to

Table 4.4: Given are the parameters, the original range considered before discretization and the number of levels considered after discretization for the first case study. The number of candidate parameter settings is 259 200.

parameter	range	#levels
α	$[0.01, 5.00]$	5
β	$[0.01, 10.00]$	6
ρ	$[0.00, 1.00]$	8
γ	$[0.01, 5.00]$	6
m	$[5, 100]$	5
nn	$[5, 50]$	4
q_0	$[0.0, 1.0]$	9

be determined. The first is the choice of the ACO algorithm, among the five variants $\mathcal{MM}$AS, ant colony system (ACS), rank-based ant system (RAS), elitist ant system (EAS), ant system (AS). The second is the local search type l, including four levels: no local search, 2-opt, 2.5-opt, and 3-opt. All the ACO construction algorithms share the three continuous parameter α, β, and ρ, and two quasi-continuous parameters m and nn, which have been introduced before. Moreover, five conditional parameters are considered: (i) the continuous parameter q_0 (introduced in Section 4.4.2) is only in effect when ACS is deployed; (ii) the quasi-continuous parameter $rasrank$, is only in effect when RAS is chosen; (iii) the quasi-continuous parameter $eants$ is only in effect when EAS is applied; (iv) the quasi-continuous parameter $nnls$ is only in effect when local search is used, namely either 2-opt, 2.5-opt or 3-opt; (v) the categorical parameter dlb is only in effect when local search is used. Here, only U/F-Race and I/F-Race are tested because *F-Race(FFD)* has so far already been outperformed by the other two variants, and, due to the large number of parameters, running *F-Race(FFD)* becomes infeasible. As computational budgets we adopted 1 500, 3 000 and 6 000 algorithm runs. The experimental results are given in Table 4.6. The two algorithms U/F-Race and I/F-Race are compared using the non-parametric pairwise Wilcoxon test using a 0.05 significance level. The statistical comparisons show that I/F-Race is again dominating. It is significantly better performing in five out of six experiments; only in one case no statistically significant difference can be identified. However, the quality differences in this set of experiments are quite small, usually below 0.1% in the five CPU seconds case, while in the 20 CPU seconds case the difference is below 0.01%. This shows that the solution quality is not very sensitive to the parameter settings,

Table 4.5: Computational results for configuring $\mathcal{MM}$AS for TSP with seven categorical parameters in 5 and 20 CPU seconds. For an explanation of the table entries see the caption of Table 4.3.

	5 seconds	20 seconds	
algo	per.dev	per.dev	max.bud
F-Race(FFD)	+9.33	+4.61	768
U/F-Race	−4.49	−1.35	768
I/F-Race	**−4.84**	**−3.25**	768
F-Race(FFD)	+1.58	+2.11	1 728
U/F-Race	−0.49	−0.78	1 728
I/F-Race	**−1.10**	**−1.33**	1 728
F-Race(FFD)	+0.90	+2.38	3 888
U/F-Race	−0.27	−0.33	3 888
I/F-Race	**−0.63**	**−2.05**	3 888

which is usually the case when a strong local search such as 3-opt is used.

4.5 MADS/F-Race: Hybridizing F-Race with MADS

In the previous sections, an iterated sampling method was found to be effective for generating candidate parameter configurations for F-Race. In this section, we follow this direction by hybridizing F-Race with an established iterative sampling method called Mesh Adaptive Direct Search (MADS) [10]. MADS is a mesh-based method that adapts its search radius and directions systematically. The choice of MADS for this study mainly is due to its simplicity in cooperation with F-Race. The search of MADS is governed by the incumbent, which can be decided with the assistance of F-Race. How this is realized is shown in Section 4.5.2. The hybridization of MADS and F-Race, dubbed MADS/F-Race, is also a proof-of-concept in two aspects: firstly, instead of designing an ad-hoc iterative sampling procedure for F-Race, one can also apply established continuous optimization algorithms for generating configurations for F-Race; secondly, using F-Race as an evaluation method under the framework of iterated F-Race (Algorithm 4.1), or more generally, iterated selection, is an effective alternative for handling the stochasticity in the algorithm configuration problem.

Table 4.6: Computational results for configuring $\mathcal{MM}$AS for TSP with 12 parameters in 5 and 20 CPU seconds. For an explanation of the table entries see the caption of Table 4.3.

	5 seconds	20 seconds	
algo	per.dev	per.dev	max.bud
U/F-Race	+0.06	+0.005	1 500
I/F-Race	**−0.06**	**−0.005**	1 500
U/F-Race	+0.04	+0.009	3 000
I/F-Race	**−0.04**	**−0.009**	3 000
U/F-Race	+0.07	−0.001	6 000
I/F-Race	**−0.07**	+0.001	6 000

4.5.1 Mesh Adaptive Direct Search.

MADS [10] is a mesh-based direct search algorithm. It is reported to be robust for non-smooth optimization problems [8], and its asymptotic convergence property on continuous function with bound constraints is proved in [9]. It is an extension to the generalized pattern search [222, 7] by allowing a more extensive exploration of the variable space. The details of the algorithm is given as follows.

MADS is an iterative algorithm that generates at each iteration a population of trial points. The trial points must lie on the *current mesh*, whose coarseness is controlled by a *mesh size parameter* $\Delta_k \in \mathbb{R}_+$. At iteration k, the set of mesh points is given by

$$M_k = \bigcup_{p \in S_k} \{p + \Delta_k z : z \in \mathbb{Z}^d\} \tag{4.4}$$

where S_k is the point set that was evaluated in the previous iterations. At each iteration of MADS, two steps are applied, the `search` step and the `poll` step. In the `search` step, trial points are randomly sampled on the current mesh. In the `poll` step, trial points are generated around the incumbent one. The trial points of `poll` form the set called *frame*:

$$F_k = \{p_k \pm \Delta_k b : b \in B_k\} \tag{4.5}$$

where $B_k = \{b^1, b^2, \ldots, b^d\}$ is a basis in $\mathbb{Z}^d$. At iteration k, the mesh size parameter is updated as follows:

$$\Delta_{k+1} = \begin{cases} \Delta_k/4 & \text{if no improved mesh point is found;} \\ 4\Delta_k & \text{if an improved mesh point is found, and } \Delta_k \leq \frac{1}{4}; \\ \Delta_k & \text{otherwise,} \end{cases} \tag{4.6}$$

Algorithm 4.2 The mesh adaptive direct search

Step 0: Initialization Given p_0, a feasible initial point, $\Delta_0 > 0$, set $k = 0$ and go to Step 1.

Step 1: `search` step Let $L_k \subset M_k$ be the finite set of mesh points, and let $q^* = \arg\min_{q \in L_k} f(q)$ be the iteration best point from `search`. If $f(q^*) < f(p_k)$, declare k successful and set $p_{k+1} = q^*$ and go to Step 3; otherwise go to Step 2.

Step 2: `poll` step Generate the frame F_k as in Eq. 4.5, and let $q^* = \arg\min_{q \in F_k} f(q)$ be the iteration best point from `poll`. If $f(q^*) < f(p_k)$, declare k successful and set $p_{k+1} = q^*$; otherwise declare k unsuccessful. Go to Step 3.

Step 3: Parameter update If iteration k is unsuccessful, set $p_{k+1} = p_k$, otherwise set p_{k+1} as the improved mesh point. Update mesh size parameter Δ_k according to Eq. 4.6, $k \leftarrow k + 1$ and go to Step 1.

where the range of each variable is normalized as 1. An outline of the MADS algorithm is given in Algorithm 4.2.

4.5.2 MADS/F-Race

The stochasticity associated to the evaluation of candidate algorithm configurations usually requires to reevaluate a configuration several times to reduce the variance of the performance estimate. The most straightforward way of doing so is to apply each candidate algorithm configuration a fixed, same number of times; this variant we call MADS(fixed). The number of times an algorithm candidate configuration is run in MADS(fixed) we denote by nr, where nr is a parameter. Two disadvantages of MADS(fixed) are that (i) a priori an appropriate value of nr is not known, and (ii) the same amount of computation time is allocated to good and bad performing candidates, wasting useful computation.

These disadvantages are reduced by hybridizing F-Race and MADS, resulting in the MADS/F-Race algorithm. This hybrid can be obtained in a rather straightforward way. At each MADS iteration, independent of whether it is the `search` or `poll` step, the population of candidate configurations sampled by MADS is evaluated by F-Race, together with the incumbent point p_k. In this hybrid, F-Race identifies the best point among the sampled ones and $\{p_k\}$. If F-Race determined p_k as the best, the iteration is declared unsuccessful; otherwise the iteration is successful, and the MADS parameters are updated accordingly.

For the setting of the MADS algorithm, we follow [9]. The specific parameter values are listed in Table 4.7. For MADS/F-Race, we essen-

Table 4.7: Parameters used in MADS. d denotes the dimension of the parameter space.

parameter	value
Initial search type	Latin hypercube search
#Points in initial search	d^2
Iterative search type	random search
#Points in iterative search	$2d$
#Points in `poll` step	$2d$
Speculative search	yes
Initial mesh size parameter	1.0

tially only need to set the maximum number of algorithm evaluations we allow in each application of F-Race. Here, we explore a fixed setting of 10 times the number N_l of trial points generated from MADS at the l-th iteration.

4.5.3 Case study: MADS/F-Race vs. MADS(fixed)

We have compared the performance of MADS/F-Race and MADS(fixed) on six benchmark *configuration problems*. Each *configuration problem* consists of a parameterized algorithm to be configured, and an optimization problem to which this algorithm is applied. The six configuration problems are listed below:

- MMASTSP: the $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) algorithm [213] applied to the traveling salesman problem (TSP). There are six parameters of $\mathcal{MM}$AS to be configured (listed in Table 4.8). No local search is applied.
- MMASTSP_ls: the $\mathcal{MM}$AS applied to TSP with 2-opt local search on solutions constructed by each ant. The six parameters to be configured are the same as MMASTSP.
- ACSTSP: the ant colony system (ACS) algorithm [66] for the TSP. The six parameters to be configured are listed in Table 4.8. No local search is applied.
- ACSTSP_ls: ACS applied to TSP with 2-opt local search. The six parameters to be configured are the same as in ACSTSP.
- ILSQAP: an iterated local search (ILS) algorithm applied to the quadratic assignment problem (QAP) [215]. This ILS algorithm applies a robust tabu search as the local search and a simulated

Table 4.8: Range of each parameter considered for configuring MMASTSP (including MMASTSP_ls) and ACSTSP (including ACSTSP_ls).

MMASTSP		ACSTSP	
parameter	range	parameter	range
α	$[0.0, 5.0]$	α	$[0.0, 5.0]$
β	$[0.0, 10.0]$	β	$[0.0, 10.0]$
ρ	$[0.0, 1.00]$	ρ	$[0.0, 1.00]$
γ	$[0.01, 5.00]$	q_0	$[0.0, 1.0]$
m	$[1, 1200]$	m	$[1, 1200]$
nn	$[5, 100]$	nn	$[5, 100]$

Table 4.9: Range of each parameter considered for configuring ILSQAP. The individual parameters refer to parameters concerning the tabu search algorithm, the simulated-annealing type of acceptance criterion of the ILS, and the perturbation strength.

parameter	range	parameter	range	parameter	range	parameter	range
tc	$[0, 5]$	ia	$[3, 99]$	iu	$[0, 2]$	iy	$[0, 1]$
ti	$[0, 20]$	it	$[0, 10]$	ix	$[1, 100]$	iz	$[0, 1]$

annealing type acceptance criterion. The eight parameters to be configured are listed in Table 4.9;

- SAQAP: the simulated annealing (SA) for QAP with three parameters to be configured (Table 4.10).

In our experiments, each of the configuration problems contains 12 *domains*. These are defined by considering for each configuration problem four computation time limits (1, 2, 5, and 10 CPU seconds) for stopping the algorithm to be configured, and three values of *configuration budget* (1000, 2000, and 4000). The term *configuration budget* is determined as the maximum number of times that the algorithm to be configured can be applied during the configuration process. Take MADS(fixed) for example, given a configuration budget of 1000, and $nr = 20$, then each parameter configuration will be evaluated 20 times, and in total 50 different parameter configurations will be evaluated.

Table 4.10: Range of each parameter considered for configuring SAQAP; the parameters define the annealing schedule.

parameter	range	parameter	range	parameter	range
sb	$[0.0, 10.0]$	sc	$[0.0, 1.0]$	sd	$[1, 100000]$

Comparison of MADS/F-Race to random MADS(fixed)

As a first benchmark test for MADS/F-Race, we compare its performance to MADS(fixed), for which six values for nr are tested: $nr \in \{1, 5, 10, 20, 30, 40\}$. Given no a priori information about the best choice for nr, we select one of the six values randomly for each problem domain, and compare it with MADS/F-Race across all domains, with blocking on each instance. The comparison is done using the Wilcoxon signed rank test with continuity correction; the α-level chosen is 0.05. The experimental results show that MADS/F-Race statistically significantly performs better than MADS(fixed) with randomly chosen nr on each individual domain. The average percentage improvement of the cost obtained by MADS/F-Race over the cost obtained by the random MADS(fixed) is around 0.25%.

The leave-one-out cross-validation

The dominance of MADS/F-Race over a random selection of the value nr for MADS(fixed) is convincing if no a priori knowledge about an appropriate value for nr is available. As our knowledge of MADS(fixed) grows, we may learn a good setting for MADS(fixed), with which it can be applied to an unknown problem domain. In the sequel, we study how well the knowledge of MADS(fixed) over the learned domains can be generalized to an unknown domain. To this end, a statistical technique named leave-one-out cross-validation is applied.

Cross-validation is a technique that is commonly used in machine learning and statistics, to assess how the results of statistical analyses are generalized to an independent data set. To do so, on each cross-validation iteration, the available data set is partitioned into two sets, the training set based on which the quality of the candidate settings are observed and analyzed, and the test set on which the previously assessed results are tested. This process is iterated, using different partitions in each iteration, to reduce the variability of the validation.

There are various ways how the cross-validation can be performed. In our case, the common leave-one-out cross-validation is applied. In

Algorithm 4.3 The leave-one-out cross-validation

Given a set of domains D, a set of candidate algorithms A^c, and the function $c(a, E)$ referring to the expected cost of algorithm $a \in A^c$ on a set of domains $E \in D$.
Goal is to collect the validation set V^c
Set $V^c \leftarrow \emptyset$
for $d \in D$ **do**
  Select $\bar{a} = \arg\min_{a \in A^c} c(a, D \setminus \{d\})$
  Set $V^c \leftarrow V^c \cup \{c(\bar{a}, d)\}$
end for
Return V^c for assessment

each iteration, one domain is picked for testing, and the rest of the domains serve as the training set. The best candidate chosen in the training set is applied on the test domain, and its results are collected into a validation set. The process repeats until each domain has collected its validation data. A more formal description of how we apply the leave-one-out cross-validation is given in Algorithm 4.3.

The leave-one-out cross-validation in our experiments takes as candidates the MADS(fixed) with six values of nr. The goal is to collect the validation set as described above, and compare it with the results obtained by MADS/F-Race. Note that the data collected in the validation set are first trained, while MADS/F-Race is not. The unfairness of the comparison will show the advantage of MADS/F-Race in robustness over the naive MADS(fixed).

Comparison of MADS/F-Race to tuned MADS(fixed)

Here we compare MADS/F-Race with the tuned version of MADS(fixed). Firstly we applied the leave-one-out cross-validation for MADS(fixed) over all the six configuration problems. This is done as follows. Across $12 \cdot 6 = 72$ domains, the validation set of each domain is collected by applying MADS(fixed) with the value of nr, which has the best performance for the other 71 domains. We observed that the best value of nr determined by each combination of the 71 domains is always five. As previously, we apply the Wilcoxon test. It shows that MADS/F-Race significantly outperforms the tuned MADS(fixed). The observed difference in the mean performance between the tuned MADS(fixed) and MADS/F-Race is 0.17%.

Moreover, we observed that the best setting of nr in MADS(fixed) may differ for each individual configuration problem. To further fine-tune nr for MADS(fixed), we did a blocking on the set of domains by

Table 4.11: Comparison results of MADS/F-Race and MADS(fixed) with nr tuned in six individual configuration problems separately by leave-one-out cross-validation. The column entries with label "best.nr" show the tuned nr value for each configuration problem. It is worth noticing that, in each cross-validation process, this nr value selected by each training set can vary. But in our case, it happened to remain constant. The column entries with the label `per.dev` show the mean percentage deviation of cost obtained by MADS/F-Race comparing with tuned MADS(fixed). $+x$ ($-x$) means that the solution cost of MADS/F-Race is x% more (less) than tuned MADS(fixed), i.e. MADS/F-Race performs x% worse (better) than tuned MADS(fixed).

problem	`per.dev`	best.nr	problem	`per.dev`	best.nr
MMASTSP	-0.12	5	MMASTSP_ls	$+0.055$	5
ACSTSP	-0.64	40	ACSTSP_ls	$+0.027$	40
SAQAP	-0.030	30	ILSQAP	$+0.13$	40

configuration problems, and applied the leave-one-out cross-validation on 12 domains of each configuration problem separately. Since nr is better tuned, the results of MADS(fixed) in this case are better than obtained by MADS(fixed) tuned across all configuration problems. The comparison results of MADS/F-Race and MADS(fixed) tuned in individual configuration problems are listed in Table 4.11, together with the best value of nr for each configuration problem. It shows that MADS/F-Race obtains significantly better results than tuned MADS(fixed) on individual configuration problems MMASTSP, ACSTSP and SAQAP, while in the rest of the configuration problems MMASTSP_ls ACSTSP_ls and ILSQAP, MADS/F-Race has significantly worse results than tuned MADS(fixed). Also note that when tuning nr for individual configuration problem, the tuned nr values vary a lot from 5 to 40.

The fact that the advantage of MADS/F-Race is more clear in some domains than in others motivates us to examine factors of the configuration problems that may affect the performance of the configuration algorithms MADS(fixed) and MADS/F-Race. Given that F-Race is particularly designed to deal with the stochastic aspect of the problem, one conjecture is that for problem domains where the stochasticity is relatively high, better performance is obtained by using F-Race. As a scale-free indicator of the variability of the configurations in a problem domain, we have investigated the *variation coefficient*. The variation coefficient of each configuration problem is obtained as follows. Firstly, 200 randomly generated parameter settings are applied on 25 generated

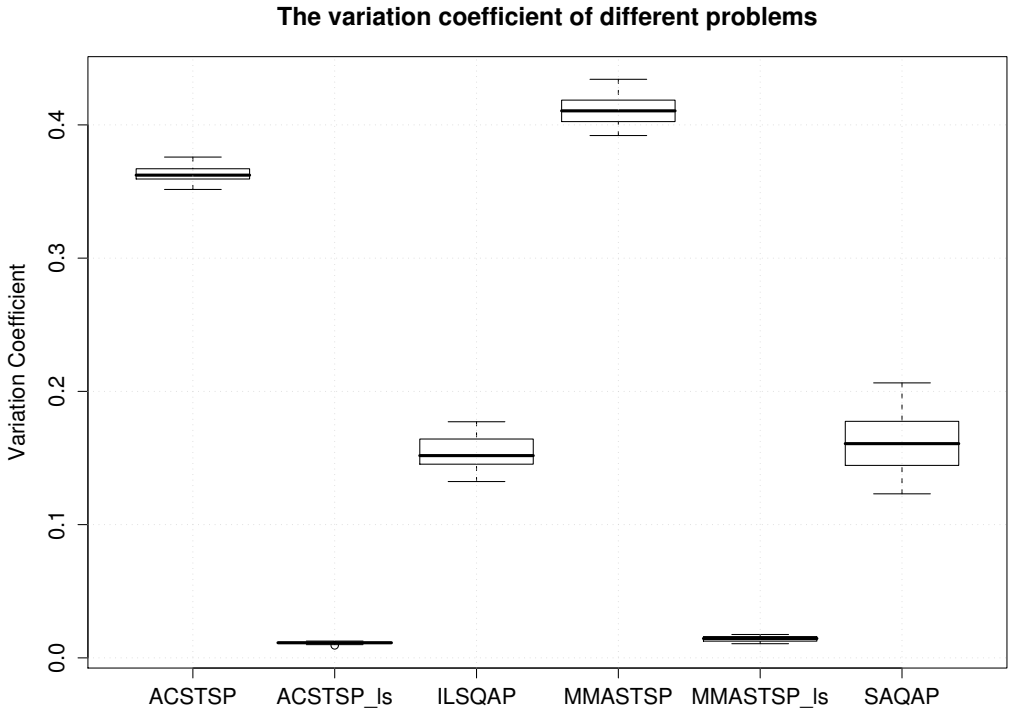


Figure 4.2: The variation coefficient of each problem used.

instances. Then, for each of the instances we compute the ratio of the standard deviation over the mean cost of the 200 parameter settings. A high value of the variation coefficient indicates steep slopes in the search landscape, while a low value of the variation coefficient indicates a flat landscape, which is often the case for algorithms when a strong local search is adopted. The box-plot of the measured variation coefficients on the six configuration problems is shown in Figure 4.2.

Interestingly, the two configuration problems MMASTSP and ACSTSP, on which MADS/F-Race performs particularly well, are the two with the highest variation coefficient. Although the variation coefficient of SAQAP is not significantly higher than that of ILSQAP, it is still much larger than that of ACSTSP_ls and MMASTSP_ls. This gives further evidence for our conjecture that the impact of F-Race is the larger the higher the variability of algorithm performance.

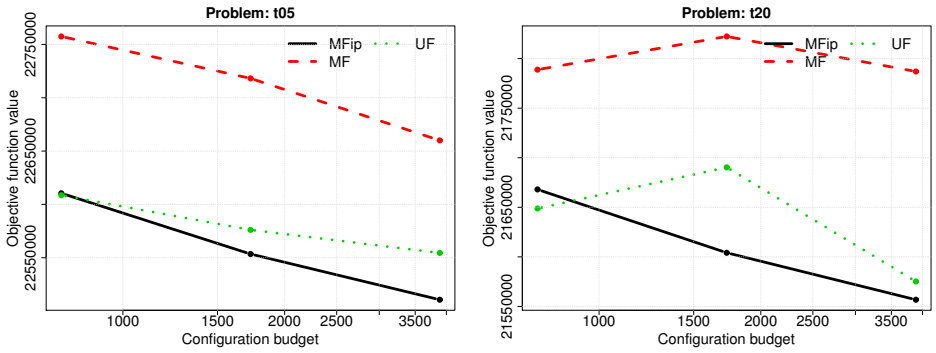


Figure 4.3: The comparison of the mean performance of three tuners: basic version of MADS/F-Race (dubbed MF [236]), its extension with a incumbent protection mechanism (MFip), and the uniform random sampling F-Race (dubbed UF) for tuning MMASTSP with seven numerical parameters under computation time 5 seconds (left) and 20 seconds (right).

4.5.4 Incumbent protection mechanism

MADS/F-Race (dubbed MF in this section) is an example of iterated F-Race: it iteratively generates candidate configurations, while at each iteration, F-Race is applied to select the incumbent among the incumbent and the newly sampled configurations of the iteration. However, the fact that the incumbent is raced at each iteration increases the probability of type I error, i.e. rejecting the incumbent when it should not. In fact, it has often been observed that the incumbent evaluated on many instances were discarded from F-Race based on only few "unlucky" instances. This leads to a poor performance of MF, as can be seen in Figure 4.3, which shows that MF performs even worse than U/F-Race (dubbed UF), the uniform random sampling design for F-Race.

To prevent the early elimination of the incumbent and make full use of the previous incumbent evaluations, the basic version of MADS/F-Race is extended with an incumbent protection mechanism. This is inspired by the intensification mechanism used in SPO$^+$ [117]. In each iteration, if an incumbent p^* that has been evaluated on N^* instances would be eliminated in F-Race by another point p with $n < N^*$ instances, we keep p^* in the race and double the number of evaluated instances $n := \min(2n, N^*)$. This is done until either the new point p is eliminated from the race, or n reaches N^*. Then we free the protection and continue the race normally. The incumbent protection mechanism is done to avoid the incumbent being eliminated by some inferior but "lucky" point with few evaluations. Experimental results illustrated in Figure 4.3 and Table 4.12 show that MADS/F-Race with

Table 4.12: Comparisons of three tuners: basic version of MADS/F-Race (dubbed MF [236]), its extension with a incumbent protection mechanism (MFip), and the uniform random sampling F-Race (dubbed UF) for tuning $\mathcal{MM}$AS for the TSP with four numerical parameters for a computation time bound of 5 and 20 seconds, respectively. The column entries with the label `per.dev` shows the mean percentage deviation of each algorithm from the reference cost. $+x$ ($-x$) means that the solution cost of the algorithm is x% more (less) than the reference cost. The column with the label `max.bud` gives the maximum number of evaluations given to each tuner.

	5 seconds	20 seconds	
tuner	per.dev	per.dev	max.bud
UF	−0.22	**−0.24**	768
MF	+0.44	+0.40	768
MFip	−0.21	−0.16	768
UF	−0.17	−0.07	1 728
MF	+0.45	+0.54	1 728
MFip	**−0.28**	**−0.47**	1 728
UF	−0.09	+0.30	3 888
MF	−0.29	+0.68	3 888
MFip	**−0.38**	**−0.38**	3 888

the incumbent protection mechanism results in a substantially and statistically significant improvement over the basic MADS/F-Race in all the six case studies. MADS/F-Race with incumbent protection also statistically significantly outperforms U/F-Race in four out of six case studies when the tuning budget is sufficiently large. However, it performs slightly worse than U/F-Race when the budget is small.

4.6 Summary

While F-Race is an effective evaluation method for selecting the best from a set of candidates under stochasticity, it needs a sampling mechanism that explores the parameter space and generates candidate configurations. In this chapter, we studied different approaches to generate parameter configurations for F-Race, and found a number of iterated sampling mechanisms performing better than previous ways of generating candidate parameter configurations such as full factorial design and random sampling design. The framework of iterated F-Race (see Algorithm 4.1) summarizes the general idea of hybridizing an iterated

sampling mechanism and F-Race as follows: in each iteration, include the best-so-far parameter configuration(s) with the newly sampled configurations, and use F-Race to rank and select the best configuration(s) from them; next, bias the sampling of the later iterations based on these selected best configurations.

Different iterated approaches are considered for generating candidate configurations for F-Race: I/F-Race, an ad hoc iterated random sampling procedure that is specifically designed for F-Race, and MADS/F-Race, which hybridizes an established continuous optimization algorithm MADS with F-Race. I/F-Race is a complete tuner that tunes not only numerical parameters but also categorical and conditional parameters, while MADS/F-Race is designed only for tuning numerical parameters. MADS/F-Race serves also as another example how established sampling algorithms can be hybridized with F-Race by generating candidate configurations iteratively.

One of the issues raised from the hybridization of F-Race and iterated sampling method is the increasing probability of type I error in the sequential hypothesis testing, as observed in Section 4.5.4. There, it is shown that a simple incumbent protection mechanism can improve the performance of MADS/F-Race or, more generally speaking, iterated F-Race. A second issue concerning the framework of iterated F-Race (Algorithm 4.1) is that, if an established sampling algorithm does not require ranking and selection at each iteration, how F-Race can be hybridized will be a problem. To address these issues, a post-selection mechanism introduced in the next chapters will be useful.

Chapter 5

Continuous optimizers for tuning numerical parameters

From this chapter on, we focused on tuning numerical parameters, including real-valued and integer parameters with a large domain [235, 233]. The resulting tuning problem can be regarded as a stochastic continuous optimization problem.

There exist a number of established tuning algorithms as reviewed in Chapter 3. Some of them, such as CALIBRA [2], SPO [19] and SPO$^+$ [117], have been designed for tuning numerical parameters only. Others, such as F-Race [36], iterated F-Race [37], ParamILS [118], REVAC [174], and gender-based genetic algorithms [5], are able to also tune categorical parameters. The aforementioned methods have produced good results and, for many optimization algorithms, they have found parameter values that improve over the parameter values originally proposed by the designers of the algorithms.

For tuning numerical parameters, a promising alternative is to treat the resulting offline algorithm tuning problem as a stochastic continuous optimization problem and to apply continuous optimization techniques to it [235, 233]. In this chapter, we follow this direction. In particular, we use state-of-the-art continuous optimization algorithms and enhance them by mechanisms for handling stochasticity in the offline algorithm tuning problem. We consider tuning both real parameters, such as the acceleration coefficients in PSO algorithms, and integer parameters, such as the colony size in ACO algorithms. When tuning integer parameters, the values returned by the continuous optimizers are rounded to the nearest integer, which is a reasonable approach if the domain of an integer parameter is large.

The remainder of this chapter is structured as follows. Section 5.1 describes the continuous optimization algorithms we use as tuning algo-

rithms, and the possible ways that they can be enhanced for handling the stochasticity. The considered tuning problems are introduced in Section 5.2. The experimental setup and results are discussed in Section 5.3. Section 5.4 analyses the parameter space of the tuning problems considered, and Section 5.5 summarizes the chapter.

5.1 Tuning algorithms

A tuning algorithm (tuner) is composed of a search algorithm and a stochasticity handling method. In what follows, we describe the possible choices we have considered for these two components in this chapter.

5.1.1 Search methods

The parameter space that we focus on contains continuous and large-domain integer parameters, where continuity and derivative information are usually unknown. Therefore the search methods for such problems should be able to handle the black-box type of objective function. To this end, we consider a number of state-of-the-art black-box continuous optimization algorithms as search methods described as follows. These search algorithms are taken from the mathematical optimization as well as the evolutionary computation communities.

Bound Optimization by Quadratic Approximation (BOBYQA)

BOBYQA [189] is a model-based trust-region algorithm for derivative-free optimization. It extends the NEWUOA algorithm [188] by enabling it to handle bound constraints. BOBYQA at each iteration forms a quadratic model that interpolates the current trust region determined by m sampled points. At each iteration, BOBYQA first determines the point that is minimal with respect to the interpolation by the quadratic model; then, the actual value of the generated point is determined by direct evaluation; finally, the model is refined by considering the actual value of the generated point. If in these steps a new best point is found, the trust region centers at this point and enlarges its radius; otherwise, if the new point fails to be the best, the trust region becomes smaller. It is recommended to set the number m of points for the quadratic model building as $m = 2d + 1$ [189], where d denotes the dimensionality of the search space. NEWUOA (BOBYQA without bound constraints) is considered to be a state-of-the-art algorithm for derivative-free continuous optimization [12].

Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

CMA-ES [98] is a (μ, λ) evolutionary strategy algorithm: at each iteration, λ offspring points are generated around the μ elite parent points. In CMA-ES, the offspring are sampled from a multivariate Gaussian distribution, which centers at a linear combination of the elite parent points and automatically adapts the covariance matrix taking into account the search trajectory. The aim of this covariance matrix adaptation is to use the trajectory information to better predict the variable influence and interactions. CMA-ES is considered to be a state-of-the-art evolutionary algorithm [12].

The default CMA-ES is further improved here by hybridizing a uniform random search in the first iteration, instead of a biased Gaussian distribution sampling. The best uniformly sampled point will serve as a starting point for CMA-ES. This modification results in statistically significant improvements in our experiments, especially when the total number of points sampled in a run of CMA-ES is small, which is usually the case in algorithm tuning problems.

Mesh Adaptive Direct Search (MADS)

MADS [10] is a mesh-based direct search algorithm that systematically adapts the mesh coarseness, the search radius, and the search direction. At each iteration, a number of points lying on a mesh are sampled. If a new best point is found, the mesh centers at this point, the mesh becomes coarser, and the search radius becomes larger; otherwise, the mesh becomes finer and the search radius becomes smaller. MADS extends the generalized pattern search methods [222] by a more extensive exploration of the variable space. For example, it allows sampling of mesh points that are at different distances from the incumbent point, and sampling from possibly infinitely many search directions.

Uniform Random and Iterated Random Sampling (URS & IRS)

URS and IRS are used as a baseline for the assessment of other tuning algorithms. They are the search methods that are used in two tuners, which are known as U/F-Race and I/F-Race, respectively [37]. URS samples a given space uniformly at random, while IRS has a mechanism to focus the search on promising regions. IRS is a model-based optimization algorithm [243] with a $(\mu, \mu + \lambda)$ selection strategy. At each iteration, λ offspring points are generated from μ elite parent points. Each offspring is sampled from a Gaussian distribution centered at one of the elite parents, and the standard deviation decreases over time in

order to force convergence of the search around the best solutions.

5.1.2 Evaluation methods

Handling stochasticity is an important aspect in the tuning problem. There are two sources of stochasticity: the randomized nature of the optimization algorithm being tuned, and the sampling of the instances of the application problem. The task of an evaluation method in a tuning problem is to evaluate the given candidate configurations and select the best. We considered the following two evaluation methods to deal with the inherent stochasticity of the tuning problem.

Repeated evaluation

The most straightforward approach is to evaluate the objective function more than once and return the average value as an estimate of the true value. Here, nr refers to the number of repetitions of the objective function evaluation. The advantages of this approach are that it is simple, and that the confidence in the estimate can be expressed as a function of nr. We tested repeated evaluation with all search methods using different values of nr. The main disadvantage of this technique is that it is blind to the actual quality of the parameter configurations being re-evaluated.

F-Race.

F-Race is a technique aimed at making a more efficient use of computational power than done by repeated evaluation. Given a set of parameter configurations, the goal of F-Race is to select the best configuration. It does so by evaluating the configurations instance by instance and by eliminating inferior configurations from consideration as soon as statistical evidence is gathered against them. The early elimination of statistically inferior configurations focuses computational resources on the more promising ones. F-Race terminates when either only one configuration remains, or a predefined computational budget is reached. The elimination mechanism of F-Race is independent of the composition of the initial set of parameter configurations. It is thus possible to integrate F-Race with any method that selects the best configurations from a given set.

5.1.3 Combining search methods and evaluation methods

F-Race is usually combined with a search method by either an iterated selection approach (see Section 3.3.1), such as I/F-Race and MADS/F-

Race studied in Chapter 4. However, F-Race cannot be combined with BOBYQA in the iterated selection approach (see Section 3.3.1), because (i) this algorithm generates one single trial point per iteration and does not need to select the best out of a set and (ii) the numerical evaluations of each point are used in the building of the quadratic model. Mainly due to this latter issue, BOBYQA requires that each configuration is evaluated using the same instances. As the main objective of this chapter is to assess the search performance of the modern continuous optimization algorithms in the tuning problems, we consider in this chapter only tuners with repeated evaluations as well as simple post-selection process; details on this post-selection process are given in Section 5.3.3.

5.2 Benchmark tuning problems

In order to compare the performance of the search algorithms described in Section 5.1.1, a number of benchmark *tuning problems* are devised as case studies. One set of case studies is obtained by considering the application of an ACO algorithm to the traveling salesman problem (TSP); another set of case studies is obtained by considering a PSO algorithm applied to continuous optimization.

5.2.1 $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System – Traveling Salesman Problem

As a first set of case studies, we have chosen $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) [213], one of the most successful ACO algorithms, as the optimization algorithm to be tuned. The application problem being addressed here is the TSP. In the TSP, one is given a graph $G = (V, A)$, where V is the set of nodes and A is the set of arcs that fully connects the graph. Each arc (i, j) has associated a cost d_{ij} of traversing it. The goal of the TSP is to find a shortest Hamiltonian cycle in the given graph G.

$\mathcal{MM}$AS is an iterative algorithm, where a colony of m ants is deployed to construct solutions during each iteration. Initially, each ant is placed on a randomly chosen node. At each step of the solution construction, an ant k at node i probabilistically selects the next node j to go to. The probabilistic choice is biased by two factors: the pheromone trail value $\tau_{ij}(t)$, where t is the iteration counter, and the locally available heuristic information η_{ij}, which in the TSP case is set to $\eta_{ij} = 1/d_{ij}$. An ant k located at node i selects to go to node j with probability

$$p_{ij}^k(t) = \frac{[\tau_{ij}(t)]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i^k} [\tau_{il}(t)]^\alpha \cdot [\eta_{il}]^\beta} \qquad \text{if } j \in \mathcal{N}_i^k \tag{5.1}$$

where $\mathcal{N}_i^k$ denotes the set of feasible nodes that ant k can move to from node i, and α and β are parameters that control the relative influence of pheromone trails and heuristic information, respectively. At the end of each iteration, on each arc (i, j) the pheromone trails are updated by

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \Delta\tau_{ij}^{best}(t), \tag{5.2}$$

where ρ, $0 < \rho \leq 1$, denotes the fraction of pheromone that is evaporated. In $\mathcal{MM}$AS, only one ant is allowed to deposit pheromone at each iteration, being either the global-best, the restart-best or the iteration-best ant. The amount of pheromone deposited is set to

$$\Delta\tau_{ij}^{best}(t) = \begin{cases} 1/f(s^{best}(t)) & \text{if } (i,j) \text{ is used by the best ant in iteration } t, \\ 0 & \text{otherwise,} \end{cases} \tag{5.3}$$

where $f(s^{best}(t))$ denotes the solution cost of solution $s^{best}(t)$. One of the most prominent features that distinguishes $\mathcal{MM}$AS from other ACO algorithms is the use of a maximum and minimum bound on the pheromone trail values, $\tau_{min} \leqslant \tau_{ij} \leqslant \tau_{max}$, to avoid search stagnation. The maximum pheromone trail is set to

$$\tau_{max} = \frac{1}{\rho \cdot f(s^{gb}(t))}, \tag{5.4}$$

where $f(s^{gb}(t))$ denotes the solution cost of the global best solution, which will be updated at every iteration. The initial pheromone trail is set to the upper bound $\tau_0 = \tau_{max}$, where $f(s^{gb}(0))$ in Equation 5.4 is determined by a simple estimation. The lower bound is set to

$$\tau_{min} = \frac{\tau_{max}}{\gamma \cdot n}, \tag{5.5}$$

where n is the dimension of the problem, which in the TSP is given by the number of nodes, and γ is a parameter. Another parameter when applying $\mathcal{MM}$AS to the TSP is nn, which determines the size of each node's candidate list, from which the next node is selected.

We have also studied the *pseudo-random proportional action choice rule* [67] in $\mathcal{MM}$AS, as this is shown to be promising in [217]. Here, with probability q_0, $0 \leq q_0 < 1$, the ant deterministically chooses as the

Table 5.1: Given are the range and the default value (def.) of each parameter considered for tuning $\mathcal{MM}$AS.

$\mathcal{MM}$AS							
param.	α	β	ρ	m	γ	nn	q_0
range	$[0.0, 5.0]$	$[0.0, 10.0]$	$[0.0, 1.00]$	$[1, 1200]$	$[0.01, 5.00]$	$[5, 100]$	$[0.0, 1.0]$
def.	1.0	2.0	0.5	25	2.0	20	0

Table 5.2: Given are the parameters used for tuning in each case study of $\mathcal{MM}$AS. We have studied the tuning problem with 2 to 6 parameters to be tuned, there are 3 case studies for each number of parameters from 2 to 6. Hence, there are a total of 15 case studies.

n.param.	case 1	case 2	case 3
2	α β;	ρ m;	γ nn;
3	α β m;	β ρ nn;	ρ γ nn;
4	α β ρ m;	α β γ nn;	ρ m γ nn;
5	α β ρ m nn;	α β ρ m γ;	α β m γ nn;
6	α β ρ m γ nn;	α β ρ m γ q_0;	α β ρ m nn q_0;

next node j the node for which $[\tau_{ij}(t)]^\alpha \cdot [\eta_{ij}]^\beta$ is maximal; with probability $1 - q_0$, the ant probabilistically chooses the next node according to Equation 5.1.

The range and the default values of the seven studied parameters are listed in Table 5.1. The default values that we adopted are suggested by the ACOTSP software [212].[1] As candidate tuning problems, we have extracted three case studies for each of $d \in \{2, 3, 4, 5, 6\}$ parameters to be tuned, resulting in $3 \times 5 = 15$ case studies. The parameters that were used for tuning in each of the case studies are listed in Table 5.2. The unused parameters in each of the case studies were fixed to their default values (see Table 5.1). The goal of each case study was to find a parameter configuration such that $\mathcal{MM}$AS performed well on the TSP within a given time limit, which was set in our experiments to five seconds.

[1]Default parameter setting for $\mathcal{MM}$AS have been proposed by [71] with $\rho = 0.02$ and $m = n$. For TSP instances in the range of the instance size we use, [214] uses the parameter setting $\rho = 0.04$ and $m = n/2$. We experimentally compared these two proposed settings with the default setting of the ACOTSP software, and adopted the latter as default for this study because, in our context, it resulted in significantly better performance than the other two settings.

Note that local search is not applied in this study. It is widely recognized that the hybrid with local search is essential in obtaining high performing ACO algorithms [71]. However, a previous study in Section 4.5 has also shown that, when incorporating local search in ACO algorithms, the landscape of the parameter space becomes rather flat. Therefore, the algorithm performance becomes relatively insensitive to the parameters, which makes it not an ideal testbed for studying tuning algorithms.

The TSP instances were generated by the DIMACS instance generator [121]. All instances are Euclidean TSP instances with 750 nodes, where the nodes are uniformly distributed in a square of side length 10 000. 1 000 such instances are generated for the tuning phase, and 300 for the testing phase.

5.2.2 Particle swarm optimization – Rastrigin functions

The second set of case studies uses a PSO algorithm [128, 129, 187]. In a PSO algorithm, a population of simple agents, called *particles*, moves in the domain of an objective function $f : \Theta \in \mathcal{R}^n$, where n is the number of variables to optimize. Each particle i at iteration t has three vectors associated with it:

1. Current position, denoted by $\vec{x}_i^t$: This vector stores the latest candidate solution generated by the particle.
2. Velocity, denoted by $\vec{v}_i^t$: This vector represents the particle's search direction.
3. Personal best position, denoted by $\vec{b}_i^t$: This vector stores the best solution found by the particle since the beginning of the algorithm's execution, that is, $\vec{b}_i^t = \arg\min_{s\in\{0,\cdots,t\}}\{f(\vec{x}_i^s)\}$.

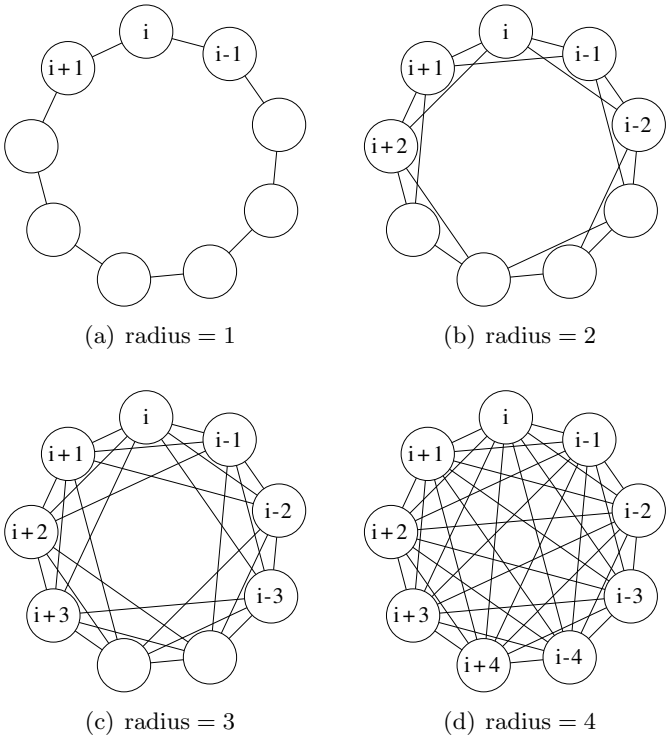
In addition, a neighborhood relation is defined among the particles through a *population topology* [127, 70], which can be thought of as a graph in which nodes represent particles, and edges represent neighbor relations. The behavior of particles in PSO algorithms is usually impacted by the best neighbor; in the following, $\mathrm{n}(i)$ gives the index of the best neighbor of particle i.

A specific variant of a PSO algorithm is the *constricted PSO* [52], where a particle i moves independently for each dimension j using the following rules:

$$v_{ij}^{t+1} \leftarrow \chi\left(v_{ij}^t + \phi_1 U_1(b_{ij}^t - x_{ij}^t) + \phi_2 U_2(b_{\mathrm{n}(i)j}^t - x_{ij}^t)\right) , \tag{5.6}$$

and

$$x_{ij}^{t+1} \leftarrow x_{ij}^t + v_{ij}^{t+1} , \tag{5.7}$$



(a) radius = 1 (b) radius = 2

(c) radius = 3 (d) radius = 4

Figure 5.1: Example population topologies with different connectivity degrees as a result of having neighborhoods of different size. In Fig. (a), radius = 1, in Fig. (b), radius = 2, in Fig. (c), radius = 3, and in Fig. (d), radius = 4.

where χ is a parameter called *constriction factor*, ϕ_1 and ϕ_2 are two parameters called *acceleration coefficients*, U_1 and U_2 are two independent, uniformly distributed random numbers in the range $[0, 1)$.

Figure 5.1 shows four topologies with different levels of connectivity for a swarm of nine particles. These topologies are specified by a parameter called *neighborhood radius*, which is the number of particles on each side of a particle if the particles are arranged as in the figure. The radius parameter takes values in the range $[1, \lfloor N/2 \rfloor]$, where N is the size of the swarm. If the radius is equal to one, the resulting topology is known as a ring topology, and if the radius is equal to $\lfloor N/2 \rfloor$, the topology becomes fully-connected (also known as star topology).

Five parameters are considered in this study, namely, χ, ϕ_1, ϕ_2, N, and the neighborhood radius. The actual range of the neighborhood radius depends on the value taken by N. Thus, we introduce a parameter p that maps linearly the continuous range $[0, 1]$ into the range $[1, \lfloor N/2 \rfloor]$. More details about the parameters used in the experiments are listed in Table 5.3. The default values of the PSO parameters were

Table 5.3: Given are the range and the default value (def.) of each parameter considered for PSO.

PSO					
param.	χ	ϕ_1	ϕ_2	N	p
range	$[0.0, 1.0]$	$[0.0, 4.0]$	$[0.0, 4.0]$	$[4, 1000]$	$[0.0, 1.0]$
def.	0.729	2.05	2.05	30	1

Table 5.4: Given are the parameters used for tuning in each case study of PSO. We have studied the tuning problem with 2 to 5 parameters to be tuned. There are 3 case studies for 2, 3, and 4 parameters, and 1 case study for 5 parameters. Hence, there are a total of 10 case studies.

n.param.	case 1	case 2	case 3
2	χ ϕ_1;	ϕ_1 ϕ_2;	N p;
3	χ ϕ_1 ϕ_2;	ϕ_1 ϕ_2 N;	χ N p;
4	χ ϕ_1 ϕ_2 N;	χ ϕ_1 ϕ_2 p;	ϕ_1 ϕ_2 N p;
5	χ ϕ_1 ϕ_2 N p;		

set as suggested by [187]. Each PSO run is stopped after 10^5 function evaluations. Similar to what we did in Section 5.2.1, we generated three case studies for each number of parameters $d \in \{2, 3, 4\}$, and one for $d = 5$. Thus, in total $3 \times 3 + 1 = 10$ case studies were constructed. The parameters that were used for tuning in each of the case studies are listed in Table 5.4. The unused parameters in each of the case studies were fixed to their default values (see Table 5.3).

For all the experiments, the constricted PSO algorithm was applied to a family of functions derived from the Rastrigin function: $nA + \sum_{i=1}^{n} (x_i^2 - A\cos(\omega x_i))$. The difficulty of this function can be adjusted by changing the values of the parameters n, A, and ω. In our experiments, we set $\omega = 2\pi$, and we vary the amplitude A to obtain functions with different fitness distance correlations (FDC) [124]. We sampled the amplitude A from a normal distribution with mean equal to 10.60171 and standard deviation equal to 2.75. These values approximately map to a normally distributed FDC with a mean of 0.7 and a standard deviation of 0.1. The FDC was estimated using 10^4 uniformly distributed random samples over the search range. Other settings are the search range and the dimensionality, n, of the problem, which are set to $[-5.12, 5.12]^n$ and $n = 100$, respectively. A total of 1 000 instances were generated for tuning and another 1 000 instances for testing.

5.3 Experiments

This section reports the results of our comparison of the tuners after some basic details of the experimental setup have been given.

5.3.1 Experimental setup

For the tuners that are obtained by combining the search algorithms with repeated evaluation, an appropriate number nr of repeated evaluations needs to be chosen. In our experiments, we consider four levels with $nr \in \{5, 10, 20, 40\}$. This choice is made following the study in [236], where the search algorithm MADS is used as the tuner. The results showed that the best nr value differed a lot from problem to problem, and ranged from 5 to 40. For F-Race, the standard version is used [33].

In our experiments, all parameters are tuned with a precision of two significant digits. This was done by rounding each sampled value. This choice was made because we observed during experimentation that reserving two significant digits in the tuning phase leads to the highest performance of the tuned algorithms. To avoid re-evaluating a same algorithm configuration on a same instance, we store historical evaluation results in an archive. If the same algorithm configuration is sampled another time, the results on the evaluated instances are read from the archive instead of rerunning the optimization algorithm.

For each of the 25 case studies mentioned in Section 5.2, we run the tuners with four different levels of the *tuning budget*. By tuning budget we mean the maximum number of times that the underlying optimization algorithm can be run during the tuning phase. Let d be the number of parameters to be tuned. The minimum level of the tuning budget is chosen to be $B_1 = 40 \cdot (2d + 2)$, which results in a budget $B_1 = 240$ when $d = 2$. The setting of B_1 is chosen in this way since BOBYQA needs at least $2n+1$ points to make the first quadratic interpolation, and this setting guarantees that BOBYQA with $nr = 40$ can make at least one quadratic interpolation guess. The other three levels of tuning budget are $B_i = 2^{i-1} \cdot B_1, i = 2, 3, 4$, thus doubling the budget from level to level.

Each tuner runs 10 *trials* on each of the four budget levels of each case study. Each trial is an execution of a *tuning phase* followed by a *testing phase*. In the testing phase, the final parameter configuration selected in the tuning phase is assessed on a set of test instances.

For the purpose of reducing experimental variance, we adopted the technique of common random instance order and common random seed. In each trial, a fixed random order of the tuning instances is used.

Additionally, each instance in each trial is assigned a random number, which serves as the common random seed each time this instance is deployed for evaluation.

The tuners are compared statistically based on the performance the optimization algorithms reach using the tuned configurations on the test instances. The test results of each tuner are compared with each other using the pairwise Wilcoxon signed rank test with blocking on each instance. In case of multiple comparisons, the Holm's adjustment is used for multiple test correction.

The experiments were carried out on cluster computing nodes, each equipped with two quad-core XEON E5410 CPUs running at 2.33 GHz with 2×6 MB second level cache and 8 GB RAM. Only one CPU core was used for each run due to the sequential implementation of both the ACOTSP software and the PSO code. The cluster runs under Cluster Rocks Linux version 5.3/CentOS 5.3. The ACOTSP and PSO programs were compiled with gcc-4.1.2-46.

In our experiments, three search algorithms, CMA-ES, MADS and BOBYQA, are extended by a restart mechanism. The restart is triggered whenever stagnation is detected. Stagnation can be detected when the search coarseness of the mesh or the trust region radius drops to a very low level; for example, less than the degree of the significant digit. In what follows, we empirically examine the tuner settings related to the restart: including the setting of the search algorithm itself, such as restart initial point, restart initial radius etc., and the setting of whether to perform a selection, dubbed post-selection, of the best configuration found during each independent restart.

5.3.2 The settings of search algorithms

For each of the search methods, the default parameter setting is adopted. However, there usually exists no default way regarding how a continuous optimizer should be restarted. These restart issues are discussed in this section, including:

- how to select the restart initial point, which is discussed below taking MADS for case study;
- how to set restart radius for BOBYQA, which is briefly discussed next;
- and another issue concerns how to initialize CMA-ES: we extend CMA-ES with a uniform sampling in the first iteration, the reason for doing so is discussed empirically in what follows.

Each of the points above is discussed in one of the subsections below. These chosen settings mainly enhance the exploration capability of the sampling methods.

Restart initial points

One issue about the restart mechanism is how to choose the initial point of each restart. In our experiments, each restart begins from a uniformly randomly sampled point in the parameter space. Other initial points could be taken into account as well. For example, we have tried restarting the algorithms from the best configuration found so far.

We compared empirically the two different initialization schemes, uniform random initial point and best-so-far initial point on the sampling algorithm MADS with post-selection and $nr = 5$. These two initialization schemes are tested on 15 case studies of tuning $\mathcal{MM}$AS for TSP, with only the highest level of tuning budget B_7. The reason to compare the restart initial settings on only the highest budget level is obvious: there are most restarts in this case so that the impact of different restart settings is highest.

The experimental results showed that the setting of uniform random initial point is better performing than the best-so-far initial point in 13 out of 15 case studies, where the difference of 10 case studies are statistically significant by Wilcoxon signed rank test. On the contrary, the best-so-far initial point is better performing in 2 case studies, where in only 1 case study the difference is statistically significant. These results show that, for a better exploration of the parameter space, using independent restart without considering previous experience is a better setting for selecting restart initial point.

As this study suggests, the restart initial point will be chosen uniformly at random for each of the sampling method considered.

Restart initial radius of BOBYQA

One of the restart setting of BOBYQA is to set its initial radius r_0. The initial radius r_0 is defined as the distance of the initial search point from the incumbent point projected at each search dimension, where the range of each dimension is scaled into a $[0, 1]$ interval. The setting r_0 should balance the trade-off: in a multi-modal landscape, a large r_0 allows to build models that cover larger region and allows to "jump" over a local optimum, so that the probability of finding a good local optimum is higher; while the small r_0 allows BOBYQA to exploit the local region more accurately. Montes de Oca et al. [170]

did an extensive empirical study to suggest that $r_0 = 0.2$, i.e. the initial search covers 40% of each dimension, is a good setting under his experiment context. We adopted his proposed setting $r_0 = 0.2$ in our experiments. Furthermore, we have found in our experiments that at the first run of BOBYQA, setting $r_0 = 0.5$, i.e., covering the whole parameter space results in notable improvements, especially in tuning problems with few trial points generated.

Initializing CMA-ES by uniform random sampling

The default setting of CMA-ES samples the population of the first iteration by a Gaussian distribution around a given initial point. This is only reasonable when sufficient a priori evidence exists for locating the promising region. However, this is usually not the case in algorithm tuning problems, and in our experiments, it is assumed that no a priori information about the parameter space is available. In this case, the initial point for biasing the sampling of the first iteration is chosen at random. Note that although each restart may use information about the parameter space from previous restarts, given the negative results in Section 5.3.2, this possibility is not further exploited in the following studies.

One alternative way to determine the initial point for CMA-ES other than choosing randomly is to hybridize CMA-ES with a uniform random sampling in the first iteration. The best uniformly sampled point will serve as the initial point for CMA-ES. Here, the number of uniformly randomly sampled points in the first iteration is set to $4 + \lfloor 3 \cdot \log d \rfloor$, which is the default setting of population size in CMA-ES. We tested this modification and compared it to the original CMA-ES with post-selection and $nr = 5$ in five case studies of $\mathcal{MM}$AS [2]. We especially extend the experiments from four budget levels to seven, adding three smaller budget levels as follows: e.g., suppose the smallest of the original four budget levels $40 \cdot (2d+2)$, then the three new budget levels are $m \cdot (2d + 2)$ with $m = 20, 10, 5$.

The empirical comparisons of the two settings of CMA-ES, the original (dubbed C) and using uniform random sampling in the first iteration (dubbed CU), are illustrated in Figure 5.2. It can be observed that using an initial uniform random sampling significantly improves the performance of CMA-ES based tuners in low budget levels (typically 1 to 3); while in high budget levels (typically 5 to 7), the performance difference is minor and less significant. The statistical significance can

[2]The five selected case studies of $\mathcal{MM}$AS are: (ρ, m), (γ, nn), (ρ, γ, nn), (α, β, ρ, m), $(\alpha, \beta, \rho, m, \gamma, nn)$

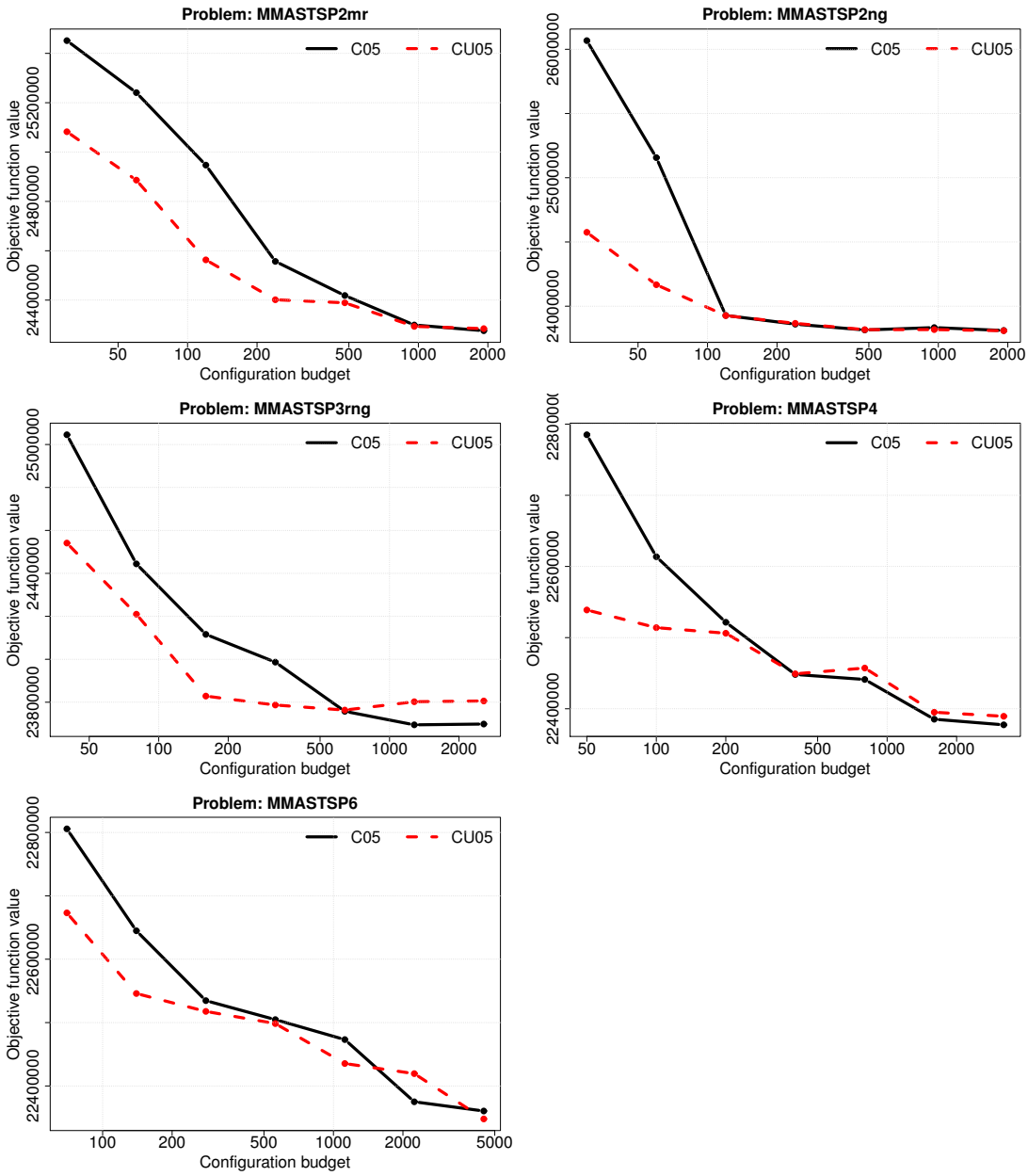


Figure 5.2: The average performance over budget level of two tuners on the five case studies of $\mathcal{MM}$AS. The two tuners are original CMA-ES (C) and CMA-ES with a uniform random sampling in the first iteration (CU). Each tuner handles stochasticity using post-selection and $nr = 5$. It shows that CU improves significantly over C in low budget levels.

Table 5.5: The comparison of two settings of CMA-ES tuners, original (C) and hybridized with uniform random sampling in the first iteration (CU). Each CMA-ES tuner handles stochasticity using post-selection and $nr = 5$. Each entry shows the number of statistically significant betters of one setting over the other, or the number of no statistical differences, in one of the seven tuning budget levels. For each budget level, there are in total five comparisons, each of which comes from one case study of $\mathcal{MM}$AS. The results show that that CU improves significantly over C in low budget levels, while in high budget levels, the performance difference is less significant.

Budget level	1	2	3	4	5	6	7
CU signif. better	5	5	4	2	2	1	1
C signif. better	0	0	0	0	1	2	2
no signif.	0	0	1	3	2	2	2

be found in Table 5.5.

5.3.3 The settings of stochasticity handling in restart

In our experiments, three sampling algorithms, CMA-ES, MADS and BOBYQA, are extended by a restart mechanism. We examine here two restart mechanisms for enhancing the stochasticity aspect of each of the search methods. These include a basic restart mechanism using repeated evaluation, and a more sophisticated restart mechanism with post-selection by F-Race.

Considering the number of restarts, it is important to mention that the speed of convergence of the three algorithms is very different. BOBYQA converges much faster than MADS, and MADS in turn much faster than CMA-ES. In fact, faster convergence also leads to a higher number of restarts and, thus, the different algorithms may benefit differently from the possibility of restarting.

Basic restart mechanism: Repeated evaluation

The simplest way to handle stochasticity is the repeatedly evaluate each generated configuration on a same fixed number of times, denoted as nr. Next, we studied what is the appropriate value of nr for the tuners that use a restart mechanism. We choose BOBYQA and MADS for this study, since they converge the fastest and, hence, have the largest number of restarts. The ranking comparisons of four nr settings for the basic restart mechanism, namely $nr \in \{5, 10, 20, 40\}$, are shown

in the left four box-plots in each of the two plots of Figure 5.3. In Figure 5.3, the distribution of the ranks of eight tuners according to their average performance in each case study at each tuning budget level is plotted. It is shown that $nr = 10$ appears to be the best setting for both BOBYQA and MADS, followed by $nr = 5, 20, 40$.

Restart mechanism with post-selection by F-Race

In the basic restart mechanism, the best setting of nr is relatively small, 10 followed by 5. In such cases, the global best configuration is identified by only a small number of evaluations. The small number of evaluations might make the selection unreliable, leading to the danger of over-tuning [33]. To avoid this, we propose a restart mechanism with post-selection by F-Race to select the best from the set of restart best configurations more carefully.

The post-selection works as follows. Each restart best configuration is stored, and in the post-selection the best across all restart best configurations is selected by means of F-Race. The tuning budget reserved for the post-selection F-Race is ω_{post} times the number of restart best configurations. The factor ω_{post} in the repeated evaluation experiments is determined by $\omega_{post} = \max\{5, (20 - nr)\}$. In the post-selection F-Race, we start the Friedman test for discarding candidates from only the $\max\{10, nr\}$-th instance, instead of five as in the default F-Race setting; this helps to make the selection more conservative.

We tested post-selection F-Race on tuners BOBYQA and MADS, and compared it with the basic restart mechanism in eight case studies of $\mathcal{MM}$AS. The results are shown in Figure 5.3 (see the right four boxplots in each of the two plots of Figure 5.3). The performance of post-selection with $nr = 5$ performs statistically significantly best among all settings. For larger nr values such as $nr = 10, 20, 40$, the difference between post-selection and repeated evaluation is not significant. Thus, the restart mechanism with post-selection and $nr = 5$ becomes our method of choice for the comparisons hereafter.

5.3.4 Comparisons of search performance of all tuning algorithms

Here we compare the search performance of all tuners composed by the five different search algorithms mentioned in Section 5.1.1. In the following, we compare the sampling algorithms directly with the setting of restart with post-selection and $nr = 5$, since it works well in the previous section. However, for URS and IRS no post-selection is

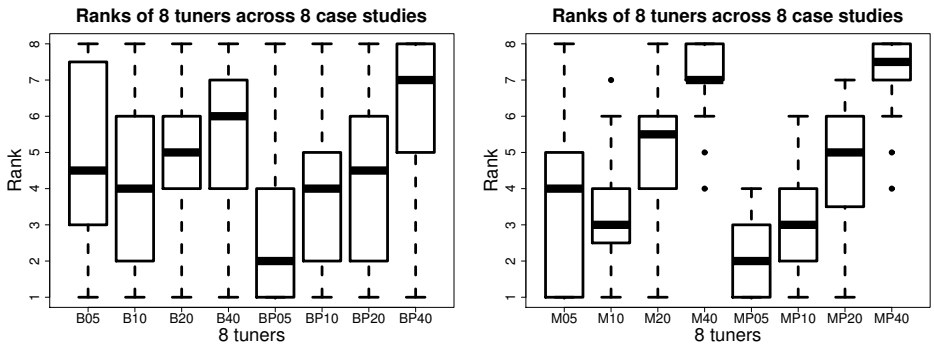


Figure 5.3: The ranking comparisons of different settings for the tuners based on BOBYQA (left) and MADS (right). Each plot shows twelve different tuners of two restart versions: without post-selection (the left four plots dubbed "B" or "M" for BOBYQA or MADS respectively) or with post-selection by F-Race (the right four plots dubbed "BP" or "MP" for BOBYQA or MADS respectively). Each version includes four levels of nr values, that is, the number of evaluations at each generated point; the four levels are $5, 10, 20, 40$. Restart with post-selection F-Race and $nr = 5$ turned out to be the best setting.

applied due to the missing restart option; still for consistency, a fixed number of evaluations $nr = 5$ is used for the comparison.

Each plot in Figure 5.4 shows the average cost of each of the five tuners on one of the 15 case studies derived from the $\mathcal{MM}$AS application to the TSP; the results for the 10 case studies derived from the PSO application to continuous function optimization are shown in Figure 5.5. Each case study includes four levels of tuning budget.

The main conclusions from this analysis are the following. For the case studies with two to four parameters, BOBYQA is in general the best performing algorithm. However, BOBYQA's performance degrades with an increase of the number of parameters to be tuned. The performance of CMA-ES is relatively robust across the number of parameters to be tuned, and across the budget levels tested. This conclusion can be further and better observed in Figure 5.6, which shows the ranking of the tuners across all numbers of parameters; and Figure 5.7, where each box plot shows the ranking of the five tuners in tuning problems of a fixed number of parameters. The large range of the boxplots for BOBYQA in Figure 5.6 reveals that the performance of BOBYQA is quite variant; Such large range of the boxplots for BOBYQA can be even observed within problems with the same number of parameters. Clear examples are $\mathcal{MM}$AS with two parameters and PSO with four parameters. The most extreme cases are two PSO case studies

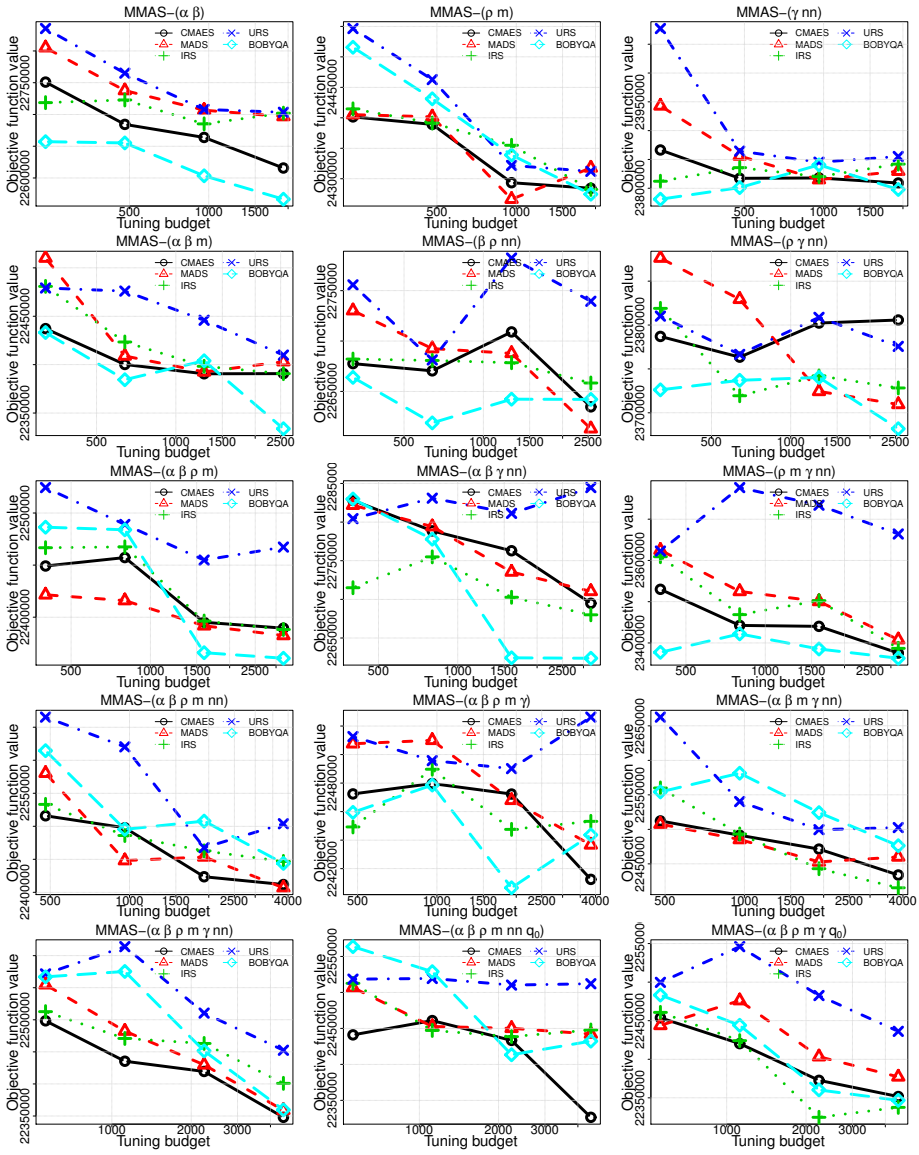


Figure 5.4: Given is the average performance over budget level of five tuners on the 15 case studies of $\mathcal{MM}$AS. Each of the five tuners uses one of the search methods BOBYQA, CMAES, MADS, IRS, or URS. Each tuner handles the stochasticity using $nr = 5$; and BOBYQA, CMAES, and MADS also use restart and post-selection. Each plot shows the average cost measured on the test instances with respect to four budget levels in each case study. Each row presents results of the case studies of the same number of parameters, from 2 (top) to 6 (bottom).

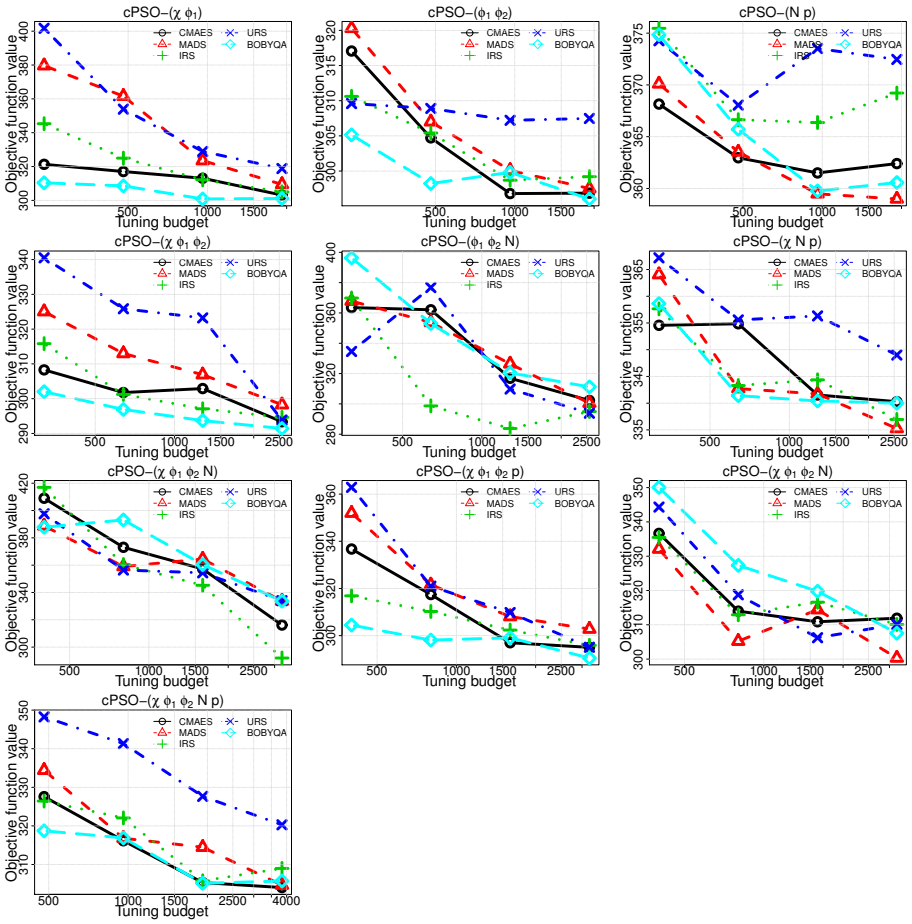


Figure 5.5: Given is the average performance over budget level of five tuners on the 10 case studies of PSO. Each of the five tuners uses one of the search methods BOBYQA, CMAES, MADS, IRS, or URS. Each tuner handles the stochasticity using $nr = 5$; and BOBYQA, CMAES, and MADS also use restart and post-selection. Each plot shows the average cost measured on the test instances with respect to four budget levels in each case study. Each row presents results of the case studies of the same number of parameters, from 2 (top) to 5 (bottom).

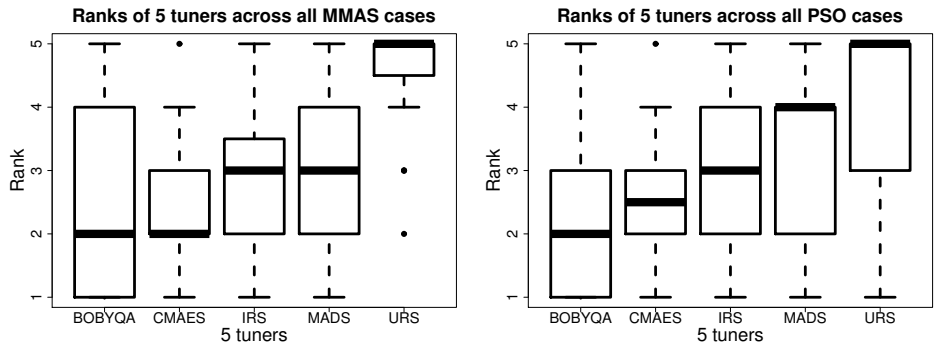


Figure 5.6: The ranking comparison of five tuners on tuning problem class $\mathcal{MM}$AS (left) and PSO (right) across all the vase studies. Each of the five tuners uses one of the search methods, namely BOBYQA, CMAES, MADS, IRS, and URS. Each tuner handles the stochasticity using $nr = 5$; and BOBYQA, CMAES, and MADS also use restart and post-selection.

with four parameters: BOBYQA performs the best in the case with χ, ϕ_1, ϕ_2, p to be tuned, but the worst in the case with ϕ_1, ϕ_2, N, p to be tuned. Finally, all the tuners tested clearly outperformed URS in most case studies (the same also holds if URS is combined with F-Race).

Figure 5.8 shows the average ranking of the five tuners grouped by the number of parameters to be tuned averaged across all budget levels for the $\mathcal{MM}$AS (left) and PSO (right) case studies. These two figures confirm the observation that BOBYQA is the best for tuning small numbers of parameters, while CMA-ES shows rather robust performance.

5.3.5 Further comparisons

Comparison between the tuned and default configurations

We compared the performance of the tuned parameter configurations to that of the default parameter configurations for $\mathcal{MM}$AS and PSO. The average cost on the test instances obtained by the default parameter configurations (see Tables 5.1 and 5.3, respectively) for $\mathcal{MM}$AS is 2.54×10^7, and for PSO is 589. Taking into account the objective ranges on the y-axis in the plots of Figures 5.4 and 5.5, we can conclude that in all 25 case studies all tuners at even the lowest tuning budget level obtained parameter configurations that resulted in significantly better average costs than the default parameter configuration.

As an example, we list the average percentage improvement of the parameter configurations obtained by restart CMAES with post-

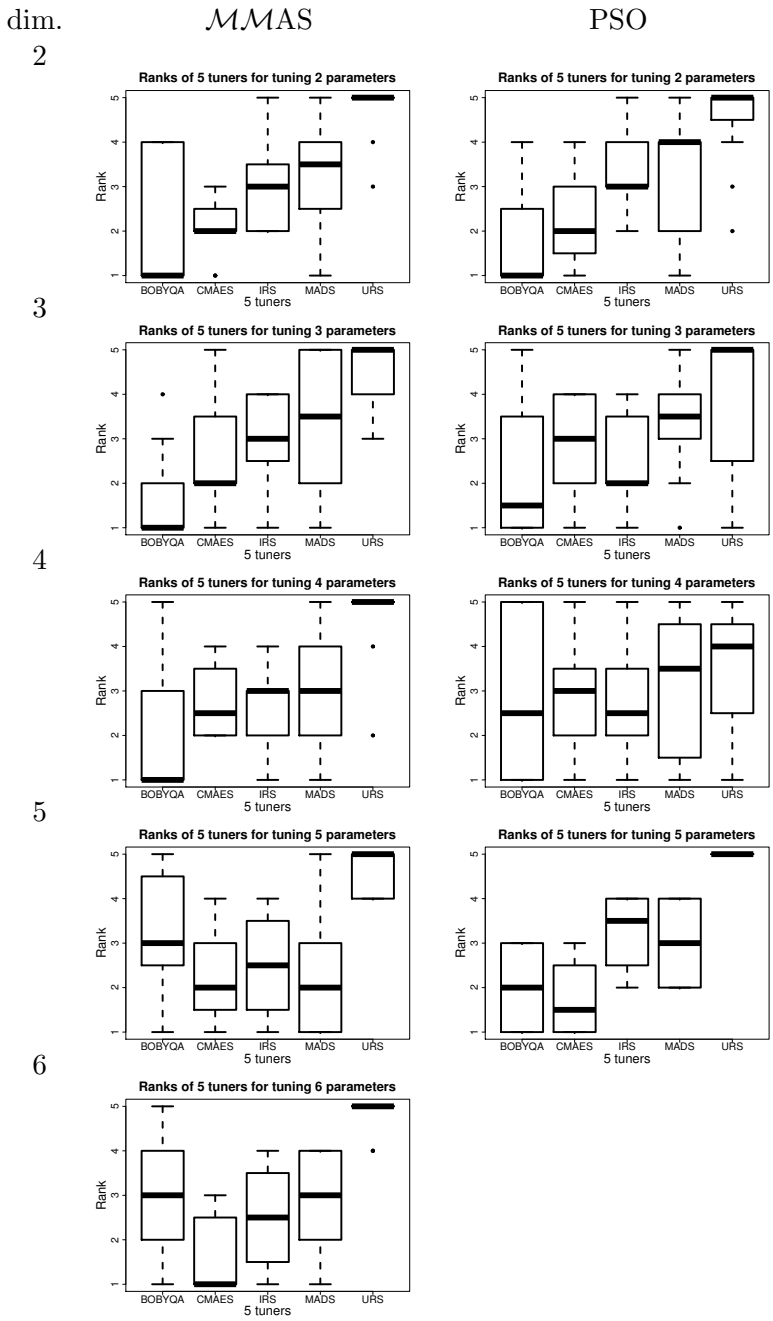


Figure 5.7: The ranking comparison of five tuners on tuning problem class $\mathcal{MM}$AS (left column) and PSO (right column) on case studies of different dimensions. Each of the five tuners uses one of the search methods, namely BOBYQA, CMAES, MADS, IRS, and URS. Each tuner handles the stochasticity using $nr = 5$; and BOBYQA, CMAES, and MADS also use restart and post-selection. Each row shows the results on the problems of a certain number of parameters from 2 to 6.

Table 5.6: Average percentage improvement of the parameter configurations tuned by CMAES with post-selection and $nr = 5$ compared to the default parameter configuration of $\mathcal{MM}$AS applied to the TSP.

budget		$\mathcal{MM}$AS		
$d = 2$	$(\alpha\ \beta)$	$(\rho\ m)$	$(\gamma\ nn)$	mean
240	10.43	3.931	6.04	6.80
480	10.69	3.980	6.23	6.97
960	10.77	4.356	6.23	7.12
1920	10.96	4.392	6.26	7.21
$d = 3$	$(\alpha\ \beta\ m)$	$(\beta\ \rho\ nn)$	$(\rho\ \gamma\ nn)$	mean
320	11.66	10.72	6.35	9.58
640	11.81	10.75	6.44	9.67
1280	11.85	10.59	6.29	9.58
2560	11.85	10.89	6.28	9.67
$d = 4$	$(\alpha\ \beta\ \rho\ m)$	$(\alpha\ \beta\ \gamma\ nn)$	$(\rho\ m\ \gamma\ nn)$	mean
400	11.62	10.13	7.36	9.70
800	11.59	10.28	7.71	9.86
1600	11.83	10.38	7.72	9.98
3200	11.85	10.65	7.97	10.16
$d = 5$	$(\alpha\ \beta\ \rho\ m\ nn)$	$(\alpha\ \beta\ \rho\ m\ \gamma)$	$(\alpha\ \beta\ m\ \gamma\ nn)$	mean
480	11.35	11.53	11.37	11.42
960	11.43	11.50	11.45	11.46
1920	11.72	11.53	11.53	11.59
3840	11.76	11.76	11.68	11.73
$d = 6$	$(\alpha\ \beta\ \rho\ m\ \gamma\ nn)$	$(\alpha\ \beta\ \rho\ m\ \gamma\ q_0)$	$(\alpha\ \beta\ \rho\ m\ nn\ q_0)$	mean
560	11.42	11.60	11.65	11.56
1120	11.67	11.73	11.57	11.66
2240	11.73	11.92	11.68	11.78
4480	12.02	12.00	12.10	12.04

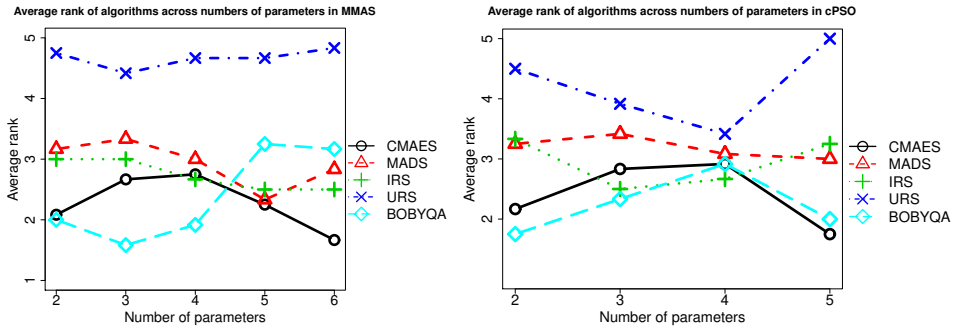


Figure 5.8: The average ranks over numbers of parameters of five tuners on case studies of $\mathcal{MM}$AS (left) and PSO (right). Each of the five tuners uses one of the search methods BOBYQA, CMAES, MADS, IRS, or URS. Each tuner handles the stochasticity using restart and post-selection with $nr = 5$.

selection and $nr = 5$ over the default parameter configurations of $\mathcal{MM}$AS and PSO in Tables 5.6 (page 107) and 5.7, respectively. In the case studies of $\mathcal{MM}$AS, the tuned configurations improve over the default configuration by around 10% on average; the average improvement for PSO is often more than 40%. Note that the default parameter configurations of the ACOTSP software and the PSO algorithm are based on years of extensive studies by experienced researchers in the field of swarm intelligence. Our results here confirm that by using algorithm tuning procedures one can automatically obtain much better performing algorithms than setting parameters based on rules of thumb and human expertise for a given problem.

The number of parameters to be tuned

In practice, when using algorithm tuning tools, we are often facing the question how many parameters should be tuned. Our computational results in Tables 5.6 and 5.7 indicate that there is a tradeoff between the number of the parameters to be tuned and the tuning budget. If we consider the largest available tuning budget in the case studies of $\mathcal{MM}$AS, it is clear that the more parameters are tuned, the better is the final the performance. In fact, in Table 5.6 the tuned performance generally improves as the number of parameters increases. A similar (but less strong) trend can be observed from Table 5.7 in the case studies of PSO. Differently from the $\mathcal{MM}$AS cases, in the PSO cases the best tuned performance is not obtained by the largest number of free parameters (five), but in the case of (χ ϕ_1 ϕ_2). If we consider limitations on the tuning budget in the case studies of $\mathcal{MM}$AS of, say, no more than 1 000 runs, the best results in Table 5.6 would be obtained

Table 5.7: Average percentage improvement of the parameter configurations tuned by CMAES with post-selection and $nr = 5$ compared to the default parameter configuration of PSO applied to a family of Rastrigin functions.

budget	PSO			
	(χ ϕ_1)	(ϕ_1 ϕ_2)	(N p)	mean
240	45.45	46.17	37.50	43.04
480	46.17	48.27	38.38	44.27
960	46.85	49.61	38.62	45.03
1920	48.54	49.60	38.47	45.54
	(χ ϕ_1 ϕ_2)	(ϕ_1 ϕ_2 N)	(χ N p)	mean
320	47.66	38.28	39.80	41.91
640	48.77	38.53	39.75	42.35
1280	48.57	46.20	42.02	45.60
2560	50.19	48.68	42.23	47.03
	(χ ϕ_1 ϕ_2 N)	(χ ϕ_1 ϕ_2 p)	(ϕ_1 ϕ_2 N p)	mean
400	30.56	42.83	42.84	38.74
800	36.66	46.11	46.69	43.15
1600	39.36	49.58	47.22	45.39
3200	46.32	49.91	47.04	47.76
	(χ ϕ_1 ϕ_2 N p)			mean
480	44.38			44.38
960	46.32			46.32
1920	48.16			48.16
3840	48.38			48.38

by tuning only three parameters, namely, α, β, and m. Therefore, as maybe expected, if the tuning budget is limited, less parameters should be tuned.

However, if less parameters are tuned, the selection of which parameters to tune becomes critical. For example, if the important parameters (ϕ_1 and ϕ_2) of PSO are not among the tuned ones, such as in case studies (N p) and (χ N p), or when tuning $\mathcal{MM}$AS without the most influential parameters α and β, such as in case studies (ρ m) and (γ nn), relatively small improvements over the default settings are obtained. In other words, knowledge on the impact of the parameters on algorithm performance is particularly useful in a situation of strongly limited tuning budget. Nevertheless, if a significantly large tuning budget is available, we would recommend to leave more parameters free for tuning. This incurs less risk regarding a wrong selection of the parameters to be tuned. An additional reason for leaving more

parameters to be tuned is that the tuning algorithm can potentially better take parameter interactions into account.

5.4 Parameter landscape analysis

We have performed a parameter landscape analysis for the swarm intelligence algorithms applied in this article. This study helps to understand better the algorithm tuning problem, and enriches our knowledge about the studied swarm intelligence algorithms.

5.4.1 The parameter landscape of two parameters

We first examine four case studies with two parameters to be tuned: (α β), (ρ m) of $\mathcal{MM}$AS, and (χ ϕ_1), (N p) of PSO. For each case, we define a 100×100 grid on the parameter space. Each grid vertex, which corresponds to a parameter configuration, is evaluated on the same 25 randomly selected instances, and the average evaluation value is computed. For each instance, a common random seed is used in order to reduce variance.

The parameter landscapes of the four case studies are visualized in the contour plots given in Figure 5.9. They contain a single global *optimal region*, where by *optimal region* we refer to the first two levels of the contours, where all the good points lie. In the PSO-(N p) case, there appear to be multiple local optima within the optimal region, but this may be caused by noise due to the small number of evaluations for each parameter configuration. This unimodality of parameter landscapes for two parameters has also been observed in [209], and shown analytically for a one-parameter space [164].

How large the optimal region of the parameter space is, depends on the particular problem. For example, if we compare the contour plots of PSO-(χ ϕ_1) with PSO-(N p), the optimal region of the former lies in a very narrow valley, where ϕ_1 takes values from 2 to 4 and χ takes values from 0.5 to 0.8; the latter has a large optimal region, where p takes values between 0.1 and 0.8 and N takes values between 100 and 300. $\mathcal{MM}$AS-(α β) has a large optimal region at the top triangle area between the three points (0.75, 10), (1, 5) and (3, 10); for $\mathcal{MM}$AS-(ρ m), the optimal region is relatively harder to locate at the corner where ρ takes values between 0.2 and 0.45 and m takes small values no more than 150.

The contour plots also give an indication of the relative influence of the parameters on algorithm performance. A typical example is $\mathcal{MM}$AS-(α β): most of the contour lines are parallel to the α-axis,

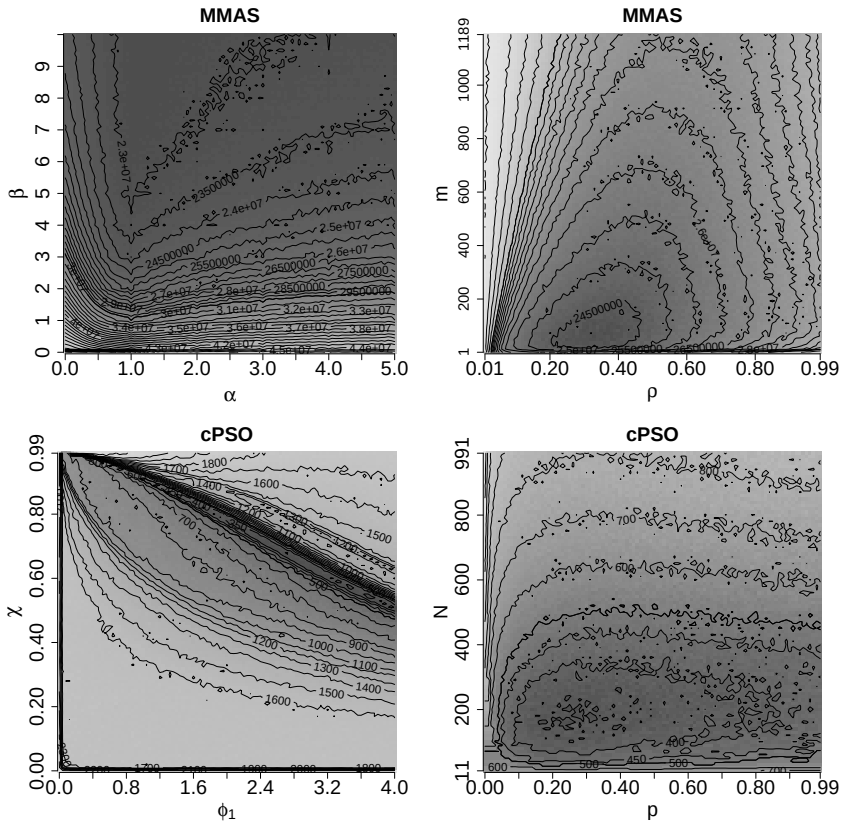


Figure 5.9: Contour plots for the parameter landscapes of four case studies with two parameters to be tuned. Top-left: $\mathcal{MM}$AS-(α β); top-right: $\mathcal{MM}$AS-(ρ m); bottom-left: PSO-(χ ϕ_1); bottom-right: PSO-(N p). The darker the region is, the better those points are. Note that in the plot of PSO-(χ ϕ_1), the region close to the χ-axis and close to the ϕ_1-axis appears to be dark because of the contour lines. It is the same case for the region close the *rho*-axis in the plot of $\mathcal{MM}$AS-(ρ m).

which shows that the parameter β is more influential than α: as long as α takes a value greater than or equal to 1, the algorithm performance is much more sensitive to the variation of β. The same conclusion can be drawn in case study PSO-(N p): as long as p takes a value between 0.1 and 0.8, the algorithm performance is more sensitive to the variation of parameter N, therefore N is more influential than p in this case.

Finally, we also observed an interesting parameter correlation in case study PSO-(χ ϕ_1), where the optimal region appears to be linearly correlated: the higher the acceleration factor ϕ_1, the smaller the constriction factor χ should be. In fact, here we fixed the parameter ϕ_2 to 2.05; a linear correlation between χ and $\phi_1 + \phi_2$ can be observed from Figure 5.10 and it is further discussed in Section 5.4.2.

5.4.2 The parameter landscape of all case studies

In the following, we extended our experiments of the parameter landscape analysis to all 25 case studies from $\mathcal{MM}$AS and PSO. To avoid the exponential growth of the number of grid points with the number of parameters, we uniformly at random sampled 4999 points in the parameter space. The 5000th point is the best parameter configuration out of the ten trials at the highest budget level of CMAES with post-selection and $nr = 5$. Each of the sampled points is evaluated on 25 instances (as in Section 5.4.1), and the average evaluation value over the 25 instances is computed.

In Tables 5.8 and 5.9 we show the best parameter configurations found by CMAES for problem class $\mathcal{MM}$AS and PSO, respectively[3], as well as the *variation coefficient* (VC) and the *fitness distance correlation* (FDC). The VC is a scale-invariant measure for the variability computed as the ratio of the standard deviation over the mean. High values of it often indicate that the algorithm performance is very sensitive to the setting of parameter values, and a careful tuning is important; low values often indicate lower sensitivity. The FDC [124] is the correlation coefficient between the fitness of sampled points and their distance to the best points. It is usually used as a measure for search difficulty of a problem. In minimization problems, a high positive FDC coefficient means that the lower the cost of a solution, the closer it is, on average, to the global optima. In the following, we discuss some conclusions from these results.

[3]Note that the best parameter configuration found by CMAES in most of the cases differs from the best one out of 5000 sampled points identified by the 25 evaluation instances. Nevertheless, we list the former instead of the latter as the "best known" parameter configuration in Tables 5.8 and 5.9 because the former is selected from the tuning process based on many more than 25 tuning instances.

Table 5.8: The best parameter configuration found by CMAES with post-selection and $nr = 5$, variation coefficient, and fitness distance correlation for each case study of $\mathcal{MM}$AS.

	$\mathcal{MM}$AS		
$d = 2$	(α β)	(ρ m)	(γ nn)
Best	(1 8)	(0.35 77)	(0.57 5)
VC/FDC	0.22 / 0.70	0.082 / 0.49	0.051 / 0.79
$d = 3$	(α β m)	(β ρ nn)	(ρ γ nn)
Best	(1 7 180)	(9 0.6 17)	(0.42 4.8 7)
VC/FDC	0.20 / 0.74	0.37 / 0.60	0.13 / 0.62
$d = 4$	(α β ρ m)	(α β γ nn)	(ρ m γ nn)
Best	(1 9 0.49 210)	(1 8 3 22)	(0.6 150 2.6 5)
VC/FDC	0.20 / 0.57	0.41 / 0.52	0.21 / 0.79
$d = 5$	(α β ρ m nn)	(α β ρ m γ)	(α β m γ nn)
Best	(1 9 0.77 180 17)	(1 8 0.52 260 0.35)	(1 7.7 270 2.2 17)
VC/FDC	0.44 / 0.50	0.21 / 0.53	0.44 / 0.54
$d = 6$	(α β ρ m γ nn)	(α β ρ m γ q_0)	(α β ρ m nn q_0)
Best	(1.1 9 0.47 310 3.7 12)	(1 8.5 0.56 330 3.1 0.41)	(1 7 0.72 270 21 0.54)
VC/FDC	0.44 / 0.35	0.14 / 0.50	0.30 / 0.35

Observing the best parameter configurations in $\mathcal{MM}$AS, it is noteworthy that the best values for α and β usually take integer values (see Table 5.8). More specifically, α usually takes value 1 (with one exception where it is 1.1) and β takes values 7, 8, or 9 (with two exceptions where it takes values 7.7 and 8.5). In fact, there is a reason why α and β are usually integers: exponentiation by integers is handled in modern compilers as multiplications while exponentiation by non-integers uses a computationally much heavier Taylor series expansion. Since the term $\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}$ of Equation 5.1 is computed very frequently, using integer values leads to a significant speedup of $\mathcal{MM}$AS, which, experimentally, can be up to 40%.

Noteworthy effects for other parameters are as follows. Firstly, the candidate list size parameter nn has a strong interaction with β: the correlation coefficient is 0.83. If β takes its default value 2, then nn takes a very small value; especially in case studies (γ nn) and (ρ m γ nn), it takes the value of its tuning lower bound, which is 5. This corresponds to a strong restriction of the size of the candidate list to compensate for the lack of greediness caused by small β. In other cases, where β takes a large value, the best nn value increases to around its default value 20. The best values for parameter m appear to be much larger than the default setting 25. The best value of parameter γ differs very strongly, and, in fact, we have found it has negligible

Table 5.9: The best parameter configuration found by CMAES with post-selection and $nr = 5$, variation coefficient, and fitness distance correlation for each case study of PSO.

	PSO		
$d = 2$	$(\chi\ \phi_1)$	$(\phi_1\ \phi_2)$	$(N\ p)$
Best	(0.53 3.8)	(3.3 1.6)	(180 0.2)
VC/FDC	0.29 / 0.52	0.40 / 0.47	0.29 / 0.68
$d = 3$	$(\chi\ \phi_1\ \phi_2)$	$(\phi_1\ \phi_2\ N)$	$(\chi\ N\ p)$
Best	(0.68 0.63 4)	(0.43 3.8 10)	(0.78 14 0.84)
VC/FDC	0.32 / 0.37	0.39 / 0.11	0.45 / 0.18
$d = 4$	$(\chi\ \phi_1\ \phi_2\ N)$	$(\chi\ \phi_1\ \phi_2\ p)$	$(\phi_1\ \phi_2\ N\ p)$
Best	(0.66 1.9 3.7 13)	(0.67 3.7 1.7 0.76)	(0.54 3.8 6 0.59)
VC/FDC	0.38 / 0.18	0.36 / 0.17	0.32 / 0.46
$d = 5$	$(\chi\ \phi_1\ \phi_2\ N\ p)$		
Best	(0.68 3.8 1.2 110 0.81)		
VC/FDC	0.41 / 0.072		

influence on algorithm performance.

Observing the best parameter configurations in PSO from Table 5.9, the most remarkable insight is that there is a strong negative linear correlation between the best values of parameter χ and $\phi_1 + \phi_2$ as shown in Figure 5.10 (correlation coefficient is -0.86). This means that the higher the sum of the acceleration factors, the lower the constriction factor should be. Note that the linear correlation of χ and ϕ_1 when fixing ϕ_2 to 2.05 can also be observed in the contour plot of tuning $(\chi\ \phi_1)$ in Figure 5.9.

Observing the VC values, in $\mathcal{MM}$AS, the parameter landscape is particularly variant when parameters β and nn are simultaneously to be tuned, which is indicated by the high VC value of these cases in Table 5.8. This means that the algorithm performance is very sensitive to the combination of β and nn. In general, we observed the tendency that the value of VC increases as the number of parameters increases, that is, the parameter landscape becomes more variant as more free parameters enter the tuning process. However, there are exceptions. In cases $(\alpha\ \beta\ \rho\ m\ \gamma\ q_0)$ and $(\alpha\ \beta\ \rho\ m\ nn\ q_0)$ the introduction of parameter q_0 results in smaller VC than when fixing q_0 to 0. Our interpretation is that the introduction of ACS's *pseudo-random proportion action choice rule* [67] into $\mathcal{MM}$AS by setting $q_0 > 0$ compensates the lack of exploitation if other parameters take poor values, making

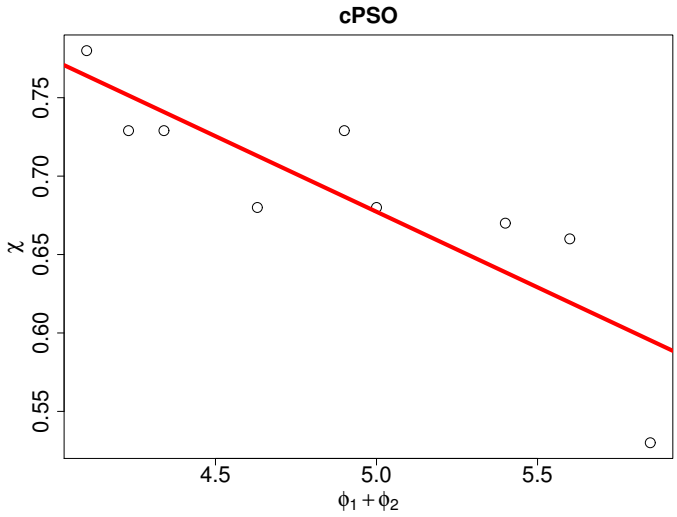


Figure 5.10: Correlation plots between $\phi_1+\phi_2$ and χ of PSO. The correlation coefficient is -0.86.

$\mathcal{MM}$AS's performance more robust. It is also worth noting that, in general, the VC in the case studies of PSO is higher than in $\mathcal{MM}$AS, since 9 out of 15 cases of $\mathcal{MM}$AS have lower VC than the lowest value 0.29 in PSO. This indicates a more variant parameter landscape for PSO.

The FDC values, in general, as depicted in Figure 5.11, decrease as the number of parameters increases. The correlation coefficient is -0.70 for $\mathcal{MM}$AS, and -0.71 for PSO. This indicates that the tuning problems become more difficult with more parameters to be tuned. Moreover, the FDC values are lower in the PSO cases than in the $\mathcal{MM}$AS cases (5 out of 10 cases in PSO have lower FDC values than the lowest value 0.35 in $\mathcal{MM}$AS); this suggests that PSO is more difficult to tune than $\mathcal{MM}$AS.

5.5 Summary

In this chapter, we have studied a number of continuous optimization algorithms for tuning numerical parameters. We have compared three state-of-the-art algorithms for black-box continuous optimization, CMAES, BOBYQA and MADS, together with uniform random sampling (URS) and the underlying sampling mechanism in iterated F-Race (IRS). These continuous optimization algorithms are improved by a restart mechanism, and CMAES is extended with a uniform random sampling in the first iteration. All the above mentioned continuous

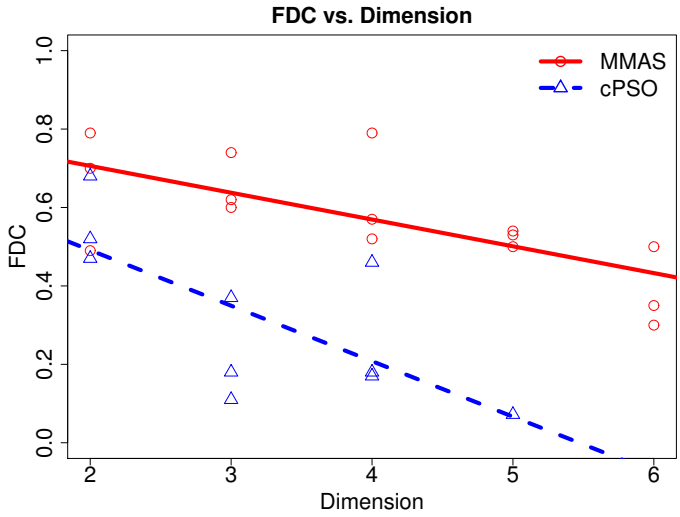


Figure 5.11: Correlation plots between the FDC and the number of parameters in case studies of $\mathcal{MM}$AS and PSO. The correlation coefficient is -0.70 for $\mathcal{MM}$AS, and -0.71 for PSO.

optimization algorithms adopt a post-selection mechanism. The basic idea of applying the post-selection mechanism is to first identify a set of elite configurations through a small number of evaluations on each configuration, and then use F-Race in the final phase of the tuning process to carefully select the best configuration among the set of elite configurations. Our experiments have proved that the post-selection mechanism is very effective in handling the stochastic nature of the tuning problem. The experiments show that both BOBYQA and CMAES exhibits good performance for searching in the algorithm parameter space. BOBYQA performs the best in the case studies with two or three parameters to be tuned, but its performance degrades with more parameters. CMAES appears to be a rather robust algorithm across all numbers of parameters that we considered.

Given the simplicity and the surprisingly good performance of the post-selection mechanism found in this chapter, a more in-depth analysis of it will be conducted in the next chapter.

Chapter 6

Post-selection mechanism for handling stochasticity in automatic configuration

In this chapter, we continue to focus on the offline configuration of numerical parameters. Offline configuration involves two sub-tasks, namely search and evaluation. The first sub-task, the generation of candidate algorithm configurations, is typically done by search algorithms including direct search methods [118, 37], model-based search methods [17, 117, 112], or modern continuous optimizers [233]. The second sub-task requires evaluating candidate algorithm configurations and at some point selecting the one with the best evaluation. This second sub-task is stochastic due to two main sources of randomness [33]. First, the algorithm to be configured may be a stochastic algorithm — this is always the case if randomized decisions are taken during the algorithm execution. Second, the stochasticity due to the fact that algorithm performance of candidate algorithm configurations may be different on different training instances — these instances may be seen as drawn from some random distribution of problem instances. Most work on configuration problem focuses on the first sub-task, while the second sub-task is relatively little discussed and studied.

In this chapter, we focus on the study of the second sub-task — the evaluation and selection of candidate configuration, and provide an in-depth analysis of the post-selection mechanism introduced in the previous chapters, which appears to be a promising mechanism for such purpose. The basic idea of the post-selection mechanism is to divide the configuration process into two phases: in the initial elite qualification phase, a number of elite algorithm configurations are identified; then, in the subsequent elite selection phase, the best of these elite

algorithm configurations is carefully selected using, for example, a racing method. Initial results in the last chapter indicated that with a careful elite selection in the final phase, the post-selection mechanism allows to use a more coarse evaluation of the candidate configurations in the elite qualification phase. As a result, more candidate configurations may be generated and, thus, potentially better configurations may be found. In the empirical study in this chapter, we extend the analysis of the post-selection method in Chapter 5 to (i) study the impact of the maximum number of algorithm runs (called configuration budget) on the configurator performance; (ii) examine the impact of using a very small number of training instances in the elite qualification phase; (iii) consider more search methods for generating the elite candidate configurations; (iv) empirically investigate some new settings of post-selection and derive new high-performing configurators for setting numerical parameters; and (v) compare the post-selection mechanism with also the best iterated selection mechanisms such as in iterated racing and FocusedILS.

The remainder of this chapter is structured as follows. Section 6.1 reviews the methods for automatic algorithm configuration. Section 6.2 studies the basic settings for post-selection, and more advanced settings for post-selection are studied in-depth in Section 6.3. The best post-selection configurators are compared with state-of-the-art configurators on further configuration benchmarks in Section 6.4. Section 6.5 provides a summary and outlook.

6.1 Configuration algorithms

A *configuration algorithm* (or *configurator*) typically combines a *search* method that generates candidate algorithm configurations and a mechanism for handling stochasticity through evaluating the configurations and selecting the most promising. Here, we introduce briefly the search methods and the evaluation methods we consider for comparisons.

6.1.1 Black-box search methods

A *search* mechanism in a configurator iteratively generates candidate algorithm configurations. In this chapter, the algorithm to be configured use mainly numerical parameters, in which case we refer to the configuration problem also as tuning problem. For these numerical parameters, including continuous or quasi-continuous parameters, we consider the following black-box continuous optimizers as search methods:

Bound Optimization by Quadratic Approximation (BOBYQA).

BOBYQA [189] is a model-based trust-region continuous optimizer that iteratively builds and refines a quadratic model based on which the trial points are sampled.

Covariance Matrix Adaptation Evolution Strategy (CMA-ES).

CMA-ES [98] is a (μ, λ)-evolutionary strategy. It iteratively samples candidate solutions from a multivariate Gaussian distribution, with a sample mean as a linear combination of μ elite parents and a covariance matrix automatically adapted based on the search trajectory.

Mesh Adaptive Direct Search (MADS).

MADS [10] is an extension of generalized pattern search algorithms. It is a mesh-based search method that systematically adapts the mesh coarseness, search radius, and search direction.

ParamILS for discrete parameters

For discrete parameter configurations, an iterated local search method underlying ParamILS is considered.

6.1.2 Evaluation method

Configurators typically have a limited *evaluation budget* B, which can be a maximum number of times the algorithm to be configured can be run on training instances. The evaluation method needs to determine how good candidate configurations are and select the one that performs best. How to allocate the available budget for evaluation is an important topic in the study of configuration algorithms, since on one side the evaluation budget is limited due to the high computational cost of each evaluation, but on the other side the evaluation error of a candidate reduces with the number of evaluations. A good compromise is to allocate more budget to promising candidates, so that they can be evaluated more carefully. The evaluation methods considered in this chapter include:

Repeated evaluation

It evaluates each candidate configuration by the same, fixed number of algorithm runs.

Racing

A *racing* method [156, 36, 33] evaluates candidate configurations instance by instance and eliminates inferior ones as soon as statistical evidence is gathered against them. Thus, better candidates also receive more evaluations. Racing methods differ in the statistical tests that are used to detect inferior candidates, for example, F-Race adopts the Friedman and its post-hoc tests, and t-Race uses Student's t-test.

Intensification

Intensification mechanisms are used in methods such as FocusedILS [118], SPO+ [117], ROAR or SMAC [112]. It is used to compare a newly generated configuration to the incumbent, and eliminate a new configuration as soon as it is worse than the incumbent in the sequence of instances the incumbent was already evaluated on. If a new candidate is not eliminated, its number of evaluations increases, and compares with the incumbent again, until it reaches the same number of evaluations as the incumbent, then the new candidate becomes the new incumbent.

6.1.3 Combination of search and evaluation

Given a search method and an evaluation method, a configurator essentially consists of an efficient, non-trivial combination of the two. We discuss two possibilities below, namely, the iterated selection and post-selection.

Iterated selection

Iterated selection we call the approach where two distinctive phases are iterated: first new candidate configurations are generated, and then evaluated by an evaluation method, possibly updating the incumbent. Most of the established configurators are based on some form of iterated selection, including SPO [17] and SPO+ [117], iterated racing techniques such as iterated F-Race [14, 37], MADS/F-Race [236], and CMA-ES/F-Race [233], or FocusedILS [118]. These methods include the incumbent from iteration to iteration. Some of them consider using an intensification mechanism to preserve the incumbent (e.g. FocusedILS and SPO+). The possible drawbacks of iterated selection are that an incumbent may be lost if no specific mechanism for incumbent preservation is used, while if an incumbent preservation mechanism is used, it may be too aggressive in eliminating potentially promis-

ing new candidates, leading to stagnation as observed occasionally in FocusedILS [118].

Post-selection

The basic idea of the post-selection mechanism, as introduced in Section 3.3.2 and 5.3.3, is to divide the configuration process into two phases: a first elite *qualification* phase and a second elite *selection* phase. During the qualification phase, a number of elite configurations are identified by running a configurator. These elite configurations can be collected by, for example, enforcing quick convergence of the configurator and then taking the best configuration in each independent restart. Alternatively, different configurators may be run simultaneously and the best configurations returned by various configurators may be qualified as elites. In the elite selection phase, an evaluation method is applied to select the best from these elite configurations. A number of configurators are devised following the post-selection approach and investigated in the following sections. We also compare post-selection configurators to iterated racing techniques and FocusedILS.

6.2 Basic settings of post-selection

Post-selection has been shown in Chapter 5 to be an effective way for combining black-box search methods and evaluation method such as racing. However, the post-selection settings reported in Chapter 5 is still naive. A further analysis of its basic settings, such as the number of instances for evaluation in each qualification, whether to use different instances for different qualification iterations, different ways of qualifying elites earlier, the hybridization of post-selection and iterated selection, are discussed in this section. This section is started by introducing the experimental setup on the configuration benchmark $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System, followed by the configuration performance of configurators that are based on repeated evaluation, iterated selection, and post-selection, and a further discussion on the basic settings of post-selection. The studied post-selection settings will be compared to the best iterated selection configurators such as iterated racing, and the intensification mechanism used in ParamILS.

Table 6.1: The range of $\mathcal{MM}$AS parameters.

param.	α	β	ρ	m
range	$[0.0, 5.0]$	$[0.0, 10.0]$	$[0.0, 1.00]$	$[1, 1200]$
param.	γ	nn	q_0	
range	$[0.01, 5.00]$	$[5, 100]$	$[0.0, 1.0]$	

6.2.1 Experimental setup

Configuration problem and instances

In this section, we focus on one configuration problem class used in the previous chapter, where the algorithm to be configured is $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) applied to the traveling salesman problem (TSP) [213]. We used the same problem setup and benchmark instances as introduced in Section 5.2.1. The numerical parameters in $\mathcal{MM}$AS that are considered include: α and β, the relative importance of pheromone trail and heuristic information; ρ, the proportion of the pheromone evaporated after each iteration; m, the number of ants; γ, which controls the gap between the minimum and maximum pheromone trail limits in $\mathcal{MM}$AS; nn, the size of the nearest neighbor candidate list in the solution construction; and q_0, the probability with which an ant selects deterministically the best possible choice at each construction step. The range of the values considered for these parameters is listed in Table 6.1. In the configuration process, each search algorithm generates the parameter space with a precision of two significant digits.

From these seven numerical parameters we extracted a number of *case studies*, where a subset of parameters is to be set while the others assume their default values. More in detail, we extracted three case studies for $d \in \{2, 3, 4, 5, 6\}$ parameters to be set, resulting in $3 \times 5 = 15$ case studies. These case studies are listed in Table 6.2.

The instances are uniformly randomly distributed Euclidean TSP instances. Two sets of instances are considered in this article: the homogeneous (`hom`) set consisting of uni-size instances of 750 nodes, 1 000 instances for the training phase, and 300 for the testing phase; and the heterogeneous (`het`) set consisting of instances ranging from 100 nodes to 1 200 nodes, 900 instances for training and 300 for testing. The computation time for $\mathcal{MM}$AS is 5 seconds. The $\mathcal{MM}$AS implementation is based on the ACOTSP software [212] with minor extensions to allow the usage of the parameter γ.

Table 6.2: The 15 case studies of configuring 2 to 6 parameters (each with 3 case studies) of $\mathcal{MM}$AS.

n.param.	case 1	case 2	case 3
2	α β;	ρ m;	γ nn;
3	α β m;	β ρ nn;	ρ γ nn;
4	α β ρ m;	α β γ nn;	ρ m γ nn;
5	α β ρ m nn;	α β ρ m γ;	α β m γ nn;
6	α β ρ m γ nn;	α β ρ m γ q_0;	α β ρ m nn q_0;

Configuration budget

In each case study, seven *budget levels* are considered. The minimum level of the configuration budget is chosen to be $B_1 = 5 \cdot (2d+2)$, which results in a budget $B_1 = 30$ when $d = 2$ and in a budget $B_1 = 70$ for $d = 6$. The other six levels of the configuration budget are $B_i = 2^{i-1} \cdot B_1, i = 2, 3, 4, 5, 6, 7$, which doubles the budget for each next level. Each budget level of each case study is considered as one *test domain*, resulting, thus, in $7 \times 15 = 105$ test domains. For each test domain, 10 trials were run. To reduce experimental variance, in each trial, the same random order of training instances is used for running each configurator, and each instance is evaluated with a common random seed. The instance order and random seed change from trial to trial. In each trial, an archive is used in order to prevent the same parameter configuration being evaluated twice on the same instance; in such case, the evaluation is read from the archive without consuming configuration budget.

Result presentation

When comparing testing results, we suppose every testing instance, large or small, is of the same importance. Therefore, in each test domain, we perform a standardized z-score normalization of the performance of configurations on each testing instance, such that for any given instance, the distribution of performance over tested configurations has mean zero and variance one. Whenever ranking results are presented, each rank is based on the mean value of the normalized performance in one test domain. Whenever results of statistical tests are reported in the following, we use Wilcoxon's signed-rank test with $\alpha = 0.05$, and with Holm's method in case of multiple comparison.

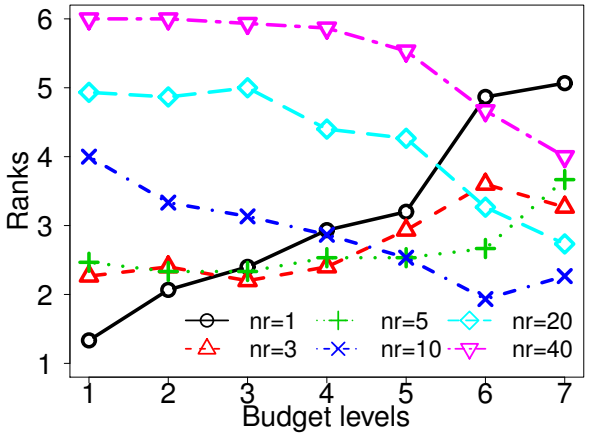


Figure 6.1: Average ranks of six nr settings ($nr \in \{1, 3, 5, 10, 20, 40\}$) for uniform random search with repeated evaluation over seven budget levels across 15 case studies of $\mathcal{MM}$AS.

Experimental environment

The experiments were carried out on cluster computing nodes, each equipped with two quad-core XEON E5410 CPUs running at 2.33 GHz with 2×6 MB second level cache and 8 GB RAM. Only one CPU core was used for each run due to the sequential implementation of the ACOTSP software. The cluster runs under Cluster Rocks Linux version 5.3/CentOS 5.3. The ACOTSP program were compiled with gcc-4.1.2-46.

6.2.2 Repeated evaluation, iterated selection, and post-selection

In this section, we examine the impact post-selection has on the performance and the behavior of various search methods with which it is combined. Before the presentations of these details, we concisely show that, in general, there exist interactions between the amount of configuration budget and settings of the evaluation method used. This sheds also some light on side-advantages of the post-selection mechanism, namely to make the configurator more robust to specific parameter settings for the evaluation method.

Repeated evaluation

The simplest evaluation method is probably repeated evaluation, where each candidate configuration is evaluated nr times. We consider here

values of $nr \in \{1, 3, 5, 10, 20, 40\}$ and evaluate their performance on the 105 test domains (15 case studies times 7 budget levels). To illustrate the trade-offs incurred between the setting of nr and different configuration budgets, we use a uniform random search (we observed similar behavior with other search methods).

Figure 6.1 shows the average ranks of the six settings of nr. The relative performance of different nr settings depends strongly on the configuration budget: while the low nr settings appears to perform best in low budget levels, their performance downgrades as the configuration budget increases. The clearest example is the setting $nr = 1$, which is the best for B_1 and B_2 but becomes the worst for the two highest budgets B_6 and B_7. On the contrary, large nr settings are the worst for low budgets but they improve as the configuration budget increases. this is especially true for $nr = 10$, which is the best setting for budget levels B_6 and B_7. Similar trade-offs were also observed in [113].

Iterated selection vs. repeated evaluation

The search methods URS and IRS are the underlying search mechanisms of U/F-Race and I/F-Race, respectively [37], both of which are established iterated selection configurators. These two iterated selection configurators were never compared to the respective search methods with a fixed number of evaluations. This experimental gap is filled here. Furthermore, note that URS and IRS do not have a restart mechanism, hence a post-selection is not applicable for them in the current context.

The results of the comparison between iterated selection and repeated evaluations are shown in Figure 6.2, where the 15 case studies of $\mathcal{MM}$AS are used for this set of experiments. The comparison of U/F-Race and URS with fixed number of repeated evaluations $nr \in \{1, 3, 5, 10, 20, 40\}$ is shown in the left plot, while the comparison of I/F-Race and IRS with fixed number of repeated evaluations is shown in the right. It is clearly shown that the iterated selection tuners, both U/F-Race and I/F-Race, significantly outperform their respective counterpart settings of a fixed number of evaluations.

Effectiveness of Post-selection

For the experiments with post-selection, we adopted whenever possible the settings used in [233]: The budget reserved for elite selection phase is set to

$$R_n = \begin{cases} 2 \cdot n^2 & \text{if } n < 10 \\ 20 \cdot n & \text{if } n \geqslant 10 \end{cases} \tag{6.1}$$

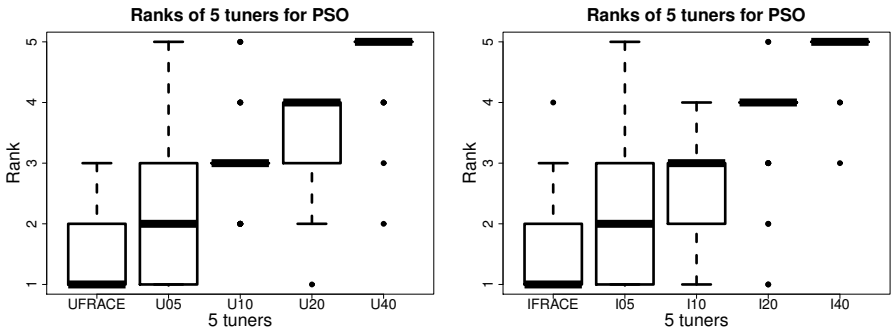


Figure 6.2: The ranking comparisons of two settings of tuners, iterated selection with F-Race and the fixed number of repeated evaluations, on two search algorithms, namely, URS (left) and IRS (right). Both plots show comparisons across 15 case studies of $\mathcal{MM}$AS. In each plot, the leftmost box shows the tuner using the iterated selection with F-Race, namely, U/F-Race or I/F-Race; and the other six boxes show the tuners using a fixed number of repeated evaluations $nr \in \{1, 3, 5, 10, 20, 40\}$, respectively. Iterated selection tuners are significantly better than its counterparts.

and the first Friedman test starts at $f_n = \min\{n + 2, 10\}$-th instance, where n is the number of candidates for the elite selection. The first line in Equation 6.1 extends [233] to ensure reasonable settings for the low budget levels. In post-selection, the default is that only restart-best configurations qualify as candidates for the elite selection, where a restart-best solution is the best solution in one independent restart of the algorithm — restarts are triggered by convergence of the algorithm. Finally, each configuration generated in the elite qualification phase is evaluated by nr same instances.

We present results with the search methods BOBYQA and MADS with either repeated evaluation or post-selection. Each search method is restarted when it stagnates. Stagnation can be detected, for example, if the search radius drops to less than the degree of significant digits (two in this work). In earlier work [233], post-selection was studied with nr ranging from 5 to 40, and it was shown that post-selection is effective when nr is small. Here, we explore smaller settings of nr equal to one and three on a subset of the test domains on the four high budget levels from B_4 to B_7 and taken from eight of the case studies and the homogeneous instance set. The box-plots for the ranking of each of the explored settings are given in Figure 6.3.

Considering the versions without post-selection (left six boxes in each plot), the best setting of nr appears to be 5 or 10, in accordance to what was observed in the previous section for uniform random sam-

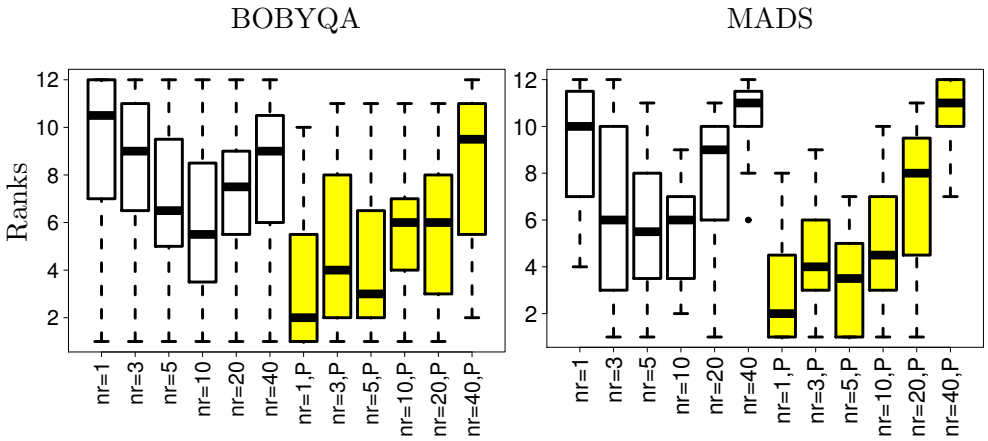


Figure 6.3: The rank distribution of 12 settings: repeated evaluation without post-selection (the left six white box-plots in each plot) or with post-selection (the right six yellow box-plots dubbed with "P"), using $nr \in \{1, 3, 5, 10, 20, 40\}$. BOBYQA (left) and MADS (right) are tested on eight case studies, each with budget levels from B_4 to B_7.

pling. However, for the versions with post-selection (right six yellow boxes dubbed with "P" for post-selection), the best setting is $nr = 1$, resulting in the best ranking improving also over the *a posterior* best settings of nr for BOBYQA and MADS without post-selection. Hence, the overall best performance with post-selection is obtained when during the run of the search method each candidate configuration is evaluated on one same instance. Saving evaluations allows to evaluate more configurations and to obtain more restart-best configurations, which then are evaluated more carefully in the elite selection phase.

6.2.3 Basic settings of post-selection

Next, we extended the study in Section 6.2.2 by considering also the low budget levels B_1, B_2, and B_3. Here, we empirically examine several basic settings of post-selection, including nr setting (Section 6.2.3), alternating instances (Section 6.2.3, dubbed A in Figure 6.4), early qualification (Section 6.2.3, dubbed E in Figure 6.4), and iterated selection hybrid (Section 6.2.3, dubbed IS in Figure 6.4).

nr setting

As high nr settings perform poorly with post-selection (see Figure 6.3), here we consider only values of $nr \in \{1, 3, 5\}$. In Figure 6.4(a-d), it can generally be observed that the best post-selection configurator in each

plot uses $nr = 1$. Considering the curves identified by "$nr = 1, P$", "$nr = 3, P$", and "$nr = 5, P$" in Figure 6.4(a-d), $nr = 1$ ranks either best or very well, and $nr = 3, 5$ rank slightly better only in the smallest budget levels of BOBYQA-based configurators (Figure 6.4(a,b)) and in high budget levels of CMA-ES-based configurators (Figure 6.4(d)). The reason for the latter is further discussed and addressed in Section 6.2.3.

Alternating instances

Note that in our first post-selection setting, each configurator restart uses the same instances (or the same instance if $nr = 1$) and only in the final elite selection phase different instances would be used. This may lead to poor results especially on instance sets where instances are heterogeneous (as in our `het` set — for this instance set good $\mathcal{MM}$AS settings are known to depend on instance size).

Here we consider a different variant, where we use *alternating instances* instead of fixed instances; hence, the nr instances used in each restart to qualify each elite configuration are different. We compared this approach empirically with the basic post-selection using fixed instances, taking BOBYQA as case study across 105 test domains of configuring $\mathcal{MM}$AS in both homogeneous instance set (`hom`, see Figure 6.4(a)) and the heterogeneous instance set (`het`, see Figure 6.4(b)). As $nr = 1$ is the best nr setting for post-selection, we compared directly $nr = 1$ fixed instance ("$nr = 1, P$" in Figure 6.4(a, b)) with alternating instances ("$nr = 1, P, A$'). In `hom`, using alternating instance performs as well as using a same, fixed instance and no statistically significant difference was detected (p-value 0.3). However, in `het`, using alternating instances leads to significant improvement. We included also $nr = 3$ alternating instances ("$nr = 3, P, A$") for `het`; it again performs significantly better than using $nr = 3$ same instances ("$nr = 3, P$"), but significantly worse than $nr = 1$ with alternating instance. To sum up, using alternating instances results in better performance, especially when the target instance set is heterogeneous.

Early qualification

CMA-ES is the only one of the three search methods, where the setting $nr = 1$ does not perform as well as $nr > 1$; its performance especially degrades as budget level increases (see Figure 6.4(d)). One reason is probably that CMA-ES converges slower than MADS and BOBYQA, so it restarts less frequently, while in our basic setting only restart-best configurations qualify for the elite selection. However, one may obtain more configurations for post-selection by qualifying configura-

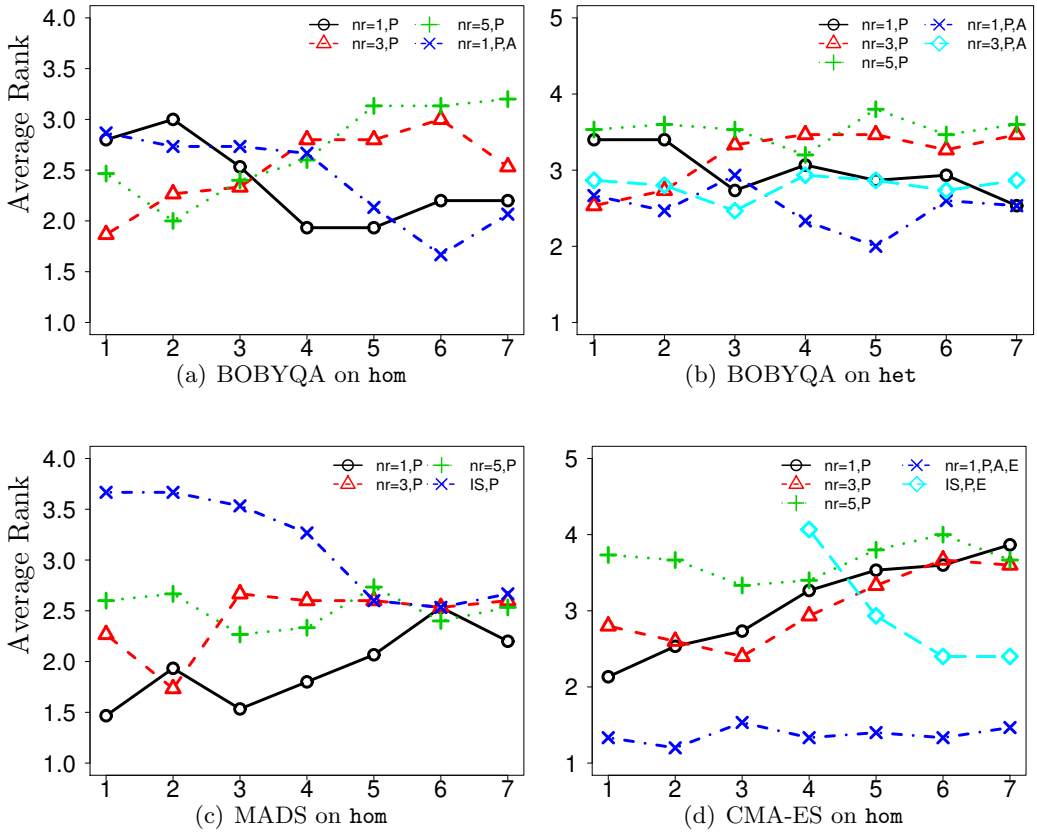


Figure 6.4: Average ranks of different settings for post-selection. These settings include *nr* values (Section 6.2.3), "P" for post-selection (Algo. 3.2), "A" for alternating instances (Section 6.2.3), "E" for early qualification (Section 6.2.3), or "IS" for iterated selection hybrid (Section 6.2.3). This study is done using three search methods as testbed, BOBYQA, MADS, and CMA-ES, and each setting is tested on seven budget levels (shown in the X-axis) for 15 case studies of $\mathcal{MM}$AS with either homogeneous (`hom`) or heterogeneous (`het`) instance sets.

tions earlier, as done, e.g., by picking all iteration-best configurations instead of only the restart-best. Besides, as suggested in Section 6.2.3 for BOBYQA, each iteration may use *alternating* instances for evaluation. This new setting "$nr = 1, P, A, E$" (E for *early qualification*) of CMA-ES configurator is shown in Figure 6.4(d) to be the significantly best-performing configurator on all budget levels.

Iterated selection with post-selection

Instead of using a fixed number of nr instances, one may apply iterated selection during the elite qualification phase. Such examples include MADS/F-Race [236, 233] introduced in Section 4.5 and CMA-ES/F-Race [233], where F-Race is not only used in the elite selection phase to select the best of the elite configurations, but also used within each iteration of the search method in the elite qualification phase to select the best among the incumbent and newly-generated configurations.[1] Besides, our CMA-ES/F-Race applies also the idea of *early qualification* (Section 6.2.3), i.e., both iteration-best and restart-best configurations are qualified as elites. However, this interesting hybrid, either MADS/F-Race ("IS, P" in Figure 6.4(c)) or CMA-ES/F-Race ("IS, P, E" in Figure 6.4(d)), does not perform well compared with the other post-selection variants derived in this work. MADS/F-Race is significantly outperformed by post-selection with $nr \leq 5$, despite the better performance of MADS/F-Race over MADS with fixed nr evaluations without post-selection [236]. CMA-ES/F-Race is also significantly outperformed by post-selection CMA-ES with one alternating instance and early qualification.

6.2.4 Post-selection vs. I/F-Race

We compare the best post-selection configurators with I/F-Race, a state-of-the-art iterated selection configurator [37] introduced in Section 4.2.2. Additionally, we compare also to U/F-Race, which generates configurations uniformly at random and then selects the best by F-Race. As the best post-selection configurators we select the best setting for each of the three search methods found in Sec. 6.2.3, including BOBYQA with one alternating instances ("$nr = 1, P, A$" in Figure 6.4(a,b)), CMA-ES with one alternating instance and early qualification ("$nr = 1, P, A, E$" in Figure 6.4(d)) and MADS with $nr = 1$ ("$nr = 1, P$" in Figure 6.4(c), only shown in `hom`).

[1] Note that in BOBYQA, each configuration has to be evaluated on the same number of instances due to the way its quadratic model is built; therefore, F-Race cannot be combined with BOBYQA in the iterated selection manner.

Figure 6.5 shows the comparison of these configurators in dependence of the budget level (top row) and the number of parameters to be configured (bottom row) on the homogeneous (left column) and the heterogeneous instance set (right column). The clear winner is the CMA-ES configurator: it significantly outperforms all other configurators in almost every budget level and every number of parameters being configured. BOBYQA generally performs well in case studies with 2, 3, or 4 parameters being configured, but its performance declines in case studies with 5 or 6 parameters, as shown in Figure 6.5(c) and Figure 6.5(d). I/F-Race is only applicable in the four high budget levels due to its default parameter settings, and it is outperformed by CMA-ES. MADS is not considered in the experiments of `het` due to its unsatisfactory performance in `hom`. All the above-mentioned configurators outperform U/F-Race.

6.2.5 Post-selection in ParamILS

For a final set of experiments we introduce post-selection into ParamILS with the goal of comparing it to the *intensification* mechanism used in FocusedILS. We adopted the version 2.3.5 of ParamILS [110], kept the search mechanism (ILS), and adapted its *intensification* mechanism into a post-selection mechanism in a straightforward manner. Since the best setting of post-selection found in Sec. 6.2.2 is using $nr = 1$, alternating instance, BasicILS(1), i.e., basic version ParamILS that evaluates each generated configuration on one instance, is adopted for the elite qualification phase of post-selection. The main question then is how to define configurations that qualify as elite. In this study, only the best configuration found in each restart is qualified. We adopted three restart schemes.

Natural restart. We restart ParamILS either when it is naturally restarted as triggered by the parameter $p_{restart}$ (set to 0.01 by default) or when the search falls into a local optimum and perturbation starts. Post-selection ParamILS with natural restart is denoted as `PPn`.

Fixed early restart. We enforce ParamILS to restart earlier so as to qualify more elites. The simplest way to enforce early restart is to restrict the maximum number of evaluation B_r for each run to a small value. Considering that each ParamILS run evaluates 10 uniformly random initial configurations before starting ILS, $B_r = 30$ appears to be a setting that allows reasonable exploitation while keeping reasonably frequent restarts. This version is denoted as `PP30`.

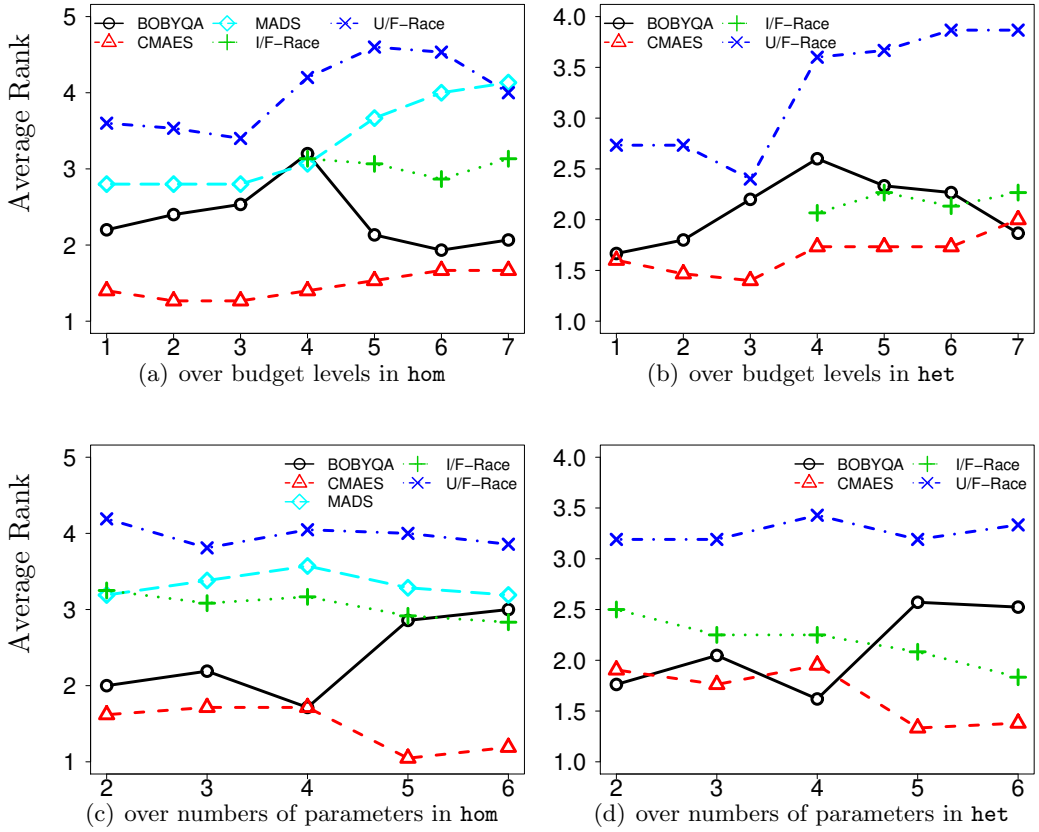


Figure 6.5: Comparison of best post-selection configurators with I/F-Race and U/F-Race for configuring 15 case studies of $\mathcal{MM}$AS with either homogeneous (`hom`) or heterogeneous (`het`) instance sets.

Incremental early restart. Besides fixing B_r, we also consider incrementing B_r by 10 from restart to restart, i.e. let $B_r = 10$ in the first restart, increment B_r to 20 in the second restart, $B_r = 30$ in the third, etc. Post-selection ParamILS with incremental early restart is denoted as `PPi`.

We compared FocusedILS with the three versions of post-selection ParamILS, `PPn`, `PP30`, and `PPi` on the six case studies of $\mathcal{MM}$AS with five or six parameters to be configured. Both homogeneous and heterogeneous instance sets are considered. Since ParamILS handles only discrete parameters, each parameter of our case studies is discretized into 10 equi-distant values. ParamILS does not support standardized z-score normalization, and so we adopted the *mean* algorithm performance as the objective measure. Accordingly, the post-selection applies

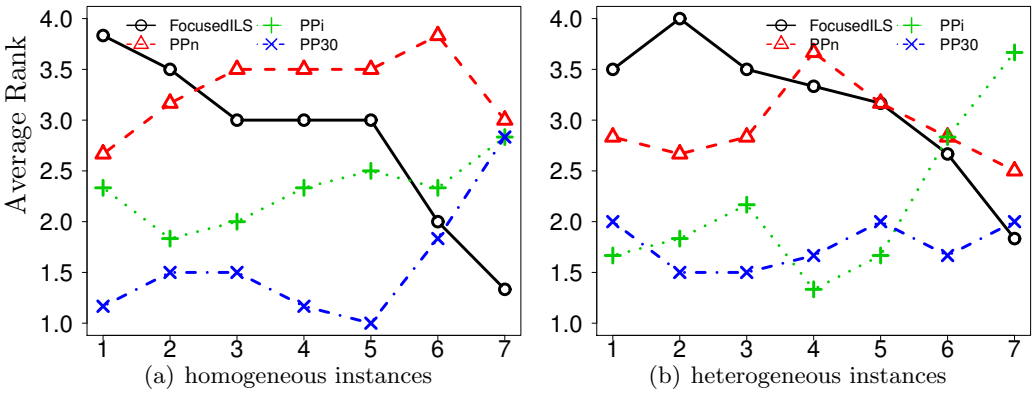


Figure 6.6: Average ranks of four ParamILS-based configurators over seven budget levels for configuring six case studies of $\mathcal{MM}$AS with either (a) homogeneous or (b) heterogeneous instance set. The four ParamILS versions include FocusedILS, and three post-selection ParamILS variants `PPn`, `PP30`, and `PPi`.

a t-Race without adjustment for multiple comparisons [33] instead of F-Race.

The results are presented in Figure 6.6. They show that post-selection with early restart, especially `PP30`, is clearly the best configurator in budget levels B_1 to B_6. FocusedILS performs better than `PP30` only in the highest budget level B_7. `PPn` does not perform very well as expected, since it usually takes around 100 to 400 evaluations to reach a natural restart; this leads to very few elite configurations, which greatly worsens the impact of post-selection. Enforcing early restart in `PP30` and `PPi` proves to be a more successful setting of post-selection than natural restart. However, frequent restart may weaken the exploitation ability in finding promising configurations during the elite qualification phase. A better approach than enforcing early restart is to use *early qualification* as done for CMA-ES in Sec. 6.2.3, qualifying elite configurations without interrupting the search procedure. However, we leave this possibility for future research.

6.2.6 Summary and outlook

Post-selection adopts two distinct phases in the automatic algorithm configuration process. In the elite qualification phase, a number of elite configurations are identified, for example, by independent runs of some algorithm configurator. The subsequent elite selection phase tries to identify then the best of these elite configurations, for example,

by a racing method. In this section, we have examined in more detail such a post-selection mechanism, proposed earlier in [233, 242], using the example application of algorithm configurators for setting numerical parameters of $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System applied to the traveling salesperson problem (TSP). Our analysis of post-selection showed that it is enough to evaluate candidate configurations on rather few instances during the elite qualification stage. In our case studies only one instance was even enough, but we expect that on other configuration tasks with more heterogeneous instances than in the TSP a larger number of instances in the elite qualification stage may be better. If the configurator in the elite qualification phase cannot gather many elite configurations, enforcing early restarts or an *early qualification* mechanism, as proposed in this chapter, may be useful. Overall, our results showed that post-selection is a promising approach that should receive further attention. In addition, we identified a post-selection CMA-ES configurator with alternating instances and early qualification, as a high-performing configurator for setting numerical parameters.

6.3 Advanced settings of post-selection

Post-selection is shown in the previous section to be effective for automatic algorithm configuration. In this section, we explore various possibilities to further improve the performance of post-selection. To this end, we first extend post-selection configurators by including as search method also the famous Nelder-Mead Simplex algorithm [175], which is one of the earliest yet still state-of-the-art derivative-free algorithms for continuous optimization. Besides, we follow the best settings of post-selection in the previous section, including one instance per qualification iteration ($nr = 1$) and alternating instances, and further explore additional advanced settings for post-selection.

Our main focus is the stochasticity handling setting in the elite qualification phase. As we learned from the experimental results in the last section, as long as a careful elite selection phase exists, we can drastically reduce the evaluation budget in each elite qualification by evaluating each search point by very few instances, typically one instance ($nr = 1$). However, this also leads to the risk that the iteration-best configuration may be lost during the search, and hence, may not be correctly identified as the elite, especially when the elite qualification needs to generate many search points before convergence. One such example is the CMA-ES in the previous section, where we need to use early qualification to shorten each elite qualification iteration. To this end, we explore the setting of *second evaluation*, which

evaluates a number of best candidate configurations in each elite qualification with an additional instance. Applying second evaluation to few iteration best configurations can help increase the probability of identifying the best configuration as the elite configuration with very little evaluation overhead. In this work, we study different number of iteration best configurations to which second evaluation is applied.

As post-selection CMA-ES is found to be the most effective configurator in the previous section, we also focus our study here on the elite qualification for CMA-ES. More specifically, we study the setting of different population sizes for CMA-ES, and population variation schemes during CMA-ES search.

We have implemented these racing based post-selection configurators into a Java package dubbed JRace. Within JRace, we have reimplemented the statistical racing methods such as F-Race and T-Race [33], which is used by our post-selection configurators. In addition, we have reimplemented the iterated racing [37] into JRace, and have empirically verified its performance. All experiments in this section ran on a computing node with a 12-core Intel Xeon X5675 CPU at 3.07 GHz and 48 GB RAM. Only a single thread is used for each configuration experiment, and a maximum of 12 threads will be simultaneously loaded on a node. The JRace is compiled using Java 1.7.0.

Two classes of benchmark algorithm configuration problems are adopted from [233], namely, 15 case studies configuring $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) for the travelling salesman problem, and 10 case studies configuring constricted particle swarm optimization (cPSO) for the Rastrigin functions.

6.3.1 Post-selection Nelder-Mead Simplex configurators

The Nelder-Mead Simplex method [175] is one of the earliest and most influential derivative-free optimization algorithms. To adapt simplex to handle the bound constraint, we start the simplex method from a random point, and let it be the first vertex of the simplex. The second to the $(n+1)$-th vertices are constructed by cumulatively changing one dimension of the previous simplex vertex by 50% of the search range. Every time the simplex samples a point outside the boundary, a large value will be returned as evaluation without running the algorithm. We further extend the simplex method by a post-selection mechanism [242] for handling the stochasticity in the evaluation of algorithm configurations. We follow the best post-selection setting in Section 6.2 [242] with $nr = 1$ (one instance per elite qualification iteration), alternating instances, and one elite qualification iteration corresponds to one restart of the simplex method. Besides, further post-selection settings for the

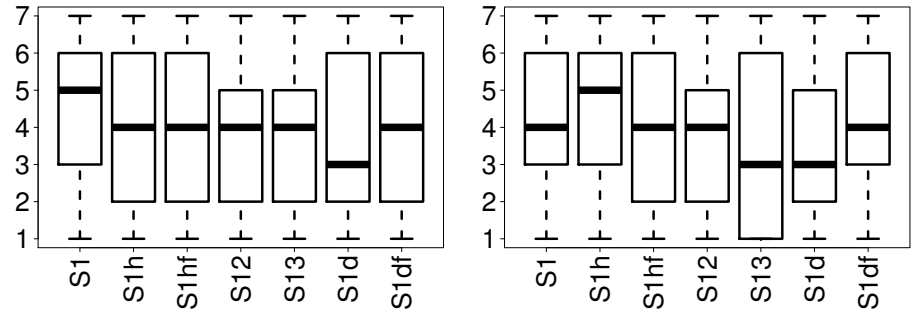
post-selection simplex method have been experimented, including a *second evaluation* technique that evaluates a number of candidates in each qualification iteration by an additional instance, and *multi-elite* technique that qualifies multiple elites per iteration. In particular, the following post-selection simplex variants are tested.

- S1: The default post-selection simplex with $nr = 1$, i.e., one instance for each qualification iteration.
- S1h: S1 with second evaluation for all the historical best configurations (that once improve best-so-far point) in each run of the simplex method except the first $n+1$ configurations that are used to construct a simplex.
- S12: S1 with second evaluation for the best two configurations in each iteration.
- S13: S1 with second evaluation for the best three configurations in each iteration.
- S1d: S1 with second evaluation for the best d configurations in each iteration, where d is the dimension of search space.

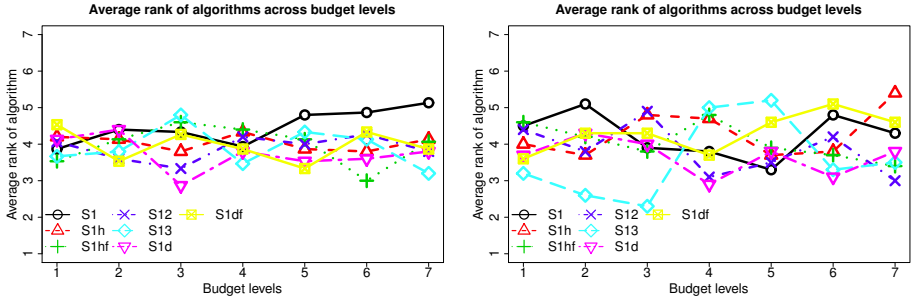
Besides second evaluation, we also study another mechanism for handling stochasticity in elite qualification called *multi-elite* mechanism, which qualifies more than one elites per elite qualification. As simplex converges more slowly than the other search methods under consideration, i.e., simplex generates more configurations in each restart, which makes it a good testbed for the multi-elite mechanism. In particular, the two following variants are considered:

- S1hf: S1h that qualifies $\lfloor \frac{d}{2} \rfloor$ best elites from each qualification iteration instead of only one. d is the dimension of the parameter space, i.e., the number of parameters to be configured.
- S1df: S1d that qualifies $\lfloor \frac{d}{2} \rfloor$ best elites per iteration.

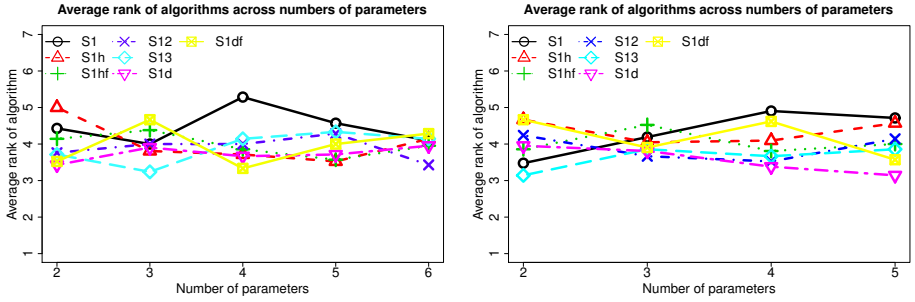
As shown in Figure 6.7, the relative performance difference between all the Simplex settings are rather close. Most of the configurators with second evaluation outperforms the default post-selection setting. Picking the iteration-best configurations for second evaluation as done in S12, S13, and S1d appears to perform better than picking the historical best configurations as in S1h. The performance difference among S12, S13, and S1d is small, and S1d appears to be most robust across all dimensions and budget levels. The multiple elite technique, especially S1df, fails to improve its counterpart S1d.



(a) $\mathcal{MM}$AS ranks (b) cPSO ranks



(c) $\mathcal{MM}$AS average rank by budget (d) cPSO average rank by budget



(e) $\mathcal{MM}$AS average rank by dimension (f) cPSO average rank by dimension

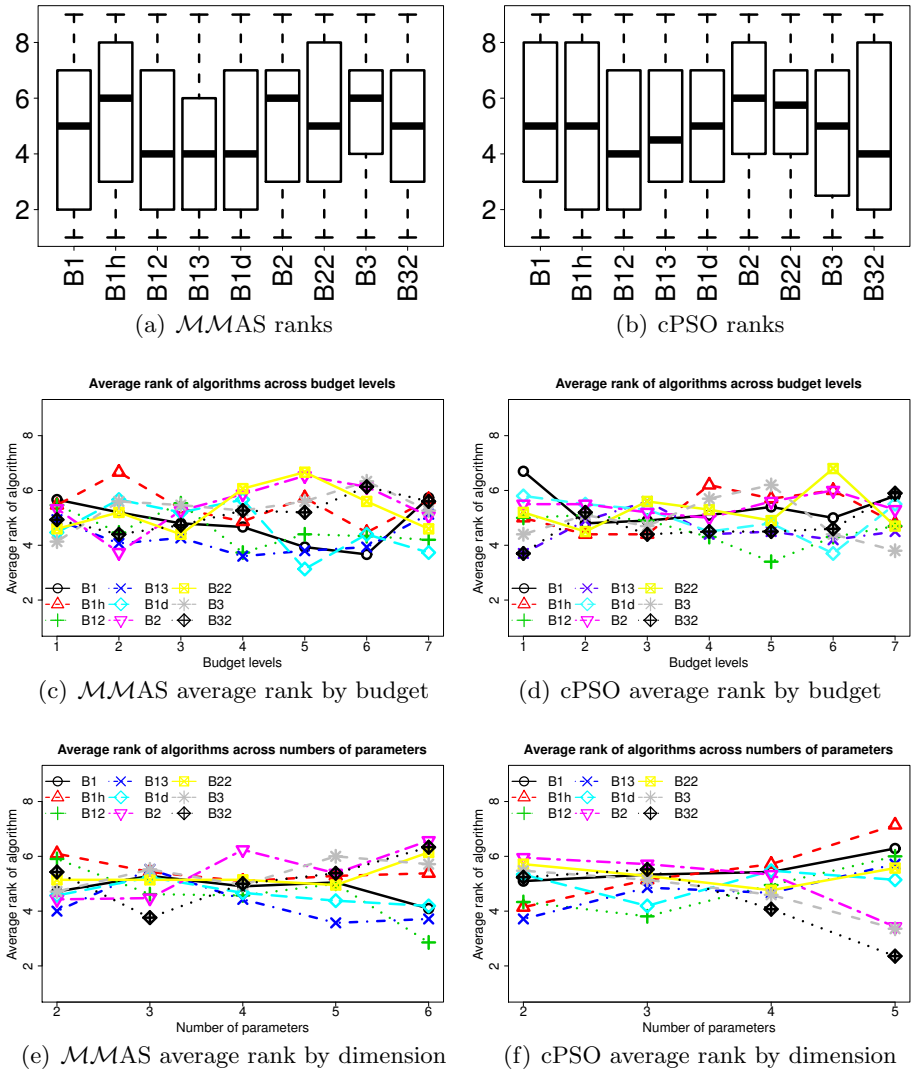
Figure 6.7: The ranking performance of seven post-selection simplex configurators in JRace: S1 with default setting; S1h with second evaluation for historical best; S1hf with second evaluation for historical best and multiple elites; S12, S13, and S1d with second evaluation for two, three, and d iteration best; S1df with d second evaluation candidates and multiple elites.

6.3.2 Post-selection BOBYQA settings

BOBYQA [189] is a model-based derivative-free search method that iteratively builds and refines a quadratic surrogate model to approximate the search space. The basic setting of Post-selection BOBYQA was already studied in Section 6.2 [233, 242] for the algorithm configuration problem. Here, we further study advanced settings for post-selection BOBYQA, including the use of second evaluation, and different restart schemes. These include:

- B1: The default post-selection BOBYQA as in [242], with initial step-size r set to 0.5 in the first restart, which covers the whole search space, and $r = 0.2$ in all the subsequent restarts. The starting point for each restart is chosen at random.
- B1h: B1 with second evaluation for all the historical best configurations (that once improve best-so-far configuration) in each run of the BOBYQA method except the first $2n + 1$ configurations in the initial phase.
- B12: B1 with second evaluation for two iteration best configurations.
- B13: B1 with second evaluation for three best iteration configurations.
- B1d: B1 with second evaluation for d iteration best configurations.
- B2: restart BOBYQA with a gradually reduced step-size r, from maximum $r_{max} = 0.5$ in the first restart, to minimum $r_{min} = 0.2$, with a reduction rate of $\rho = \sqrt[d]{0.8}$, where d is the dimension of the parameter space;
- B22: B2 with second evaluation for two iteration best configurations.
- B3: a mixed restart setting of B1 and B2. It uses B2 at each odd restarts, and uses a fixed setting of $r = 0.2$ at each even restart that chooses the best point of the previous restart as its starting point.
- B32: B3 with second evaluation for two iteration best configurations.

As shown in Figure 6.8, the relative difference between different post-selection settings for BOBYQA are quite close. The second evaluation settings, such as B12, B13, and B1d appears to improve its counterpart without second evaluation B1 noticeably; the same trend



(a) $\mathcal{MM}$AS ranks (b) cPSO ranks

(c) $\mathcal{MM}$AS average rank by budget (d) cPSO average rank by budget

(e) $\mathcal{MM}$AS average rank by dimension (f) cPSO average rank by dimension

Figure 6.8: The ranking performance of nine post-selection simplex configurators in JRace: B1 with default restart setting; B1h with second evaluation for historical best; B12, B13, and B1d with second evaluation for two, three, and d iteration best; B2 with restart scheme of gradually reduced step size; B22 is B2 with second evaluation for two iteration best; B3 with a mixed restart scheme of B1 and B2; B32 is B3 with second evaluation for two iteration best.

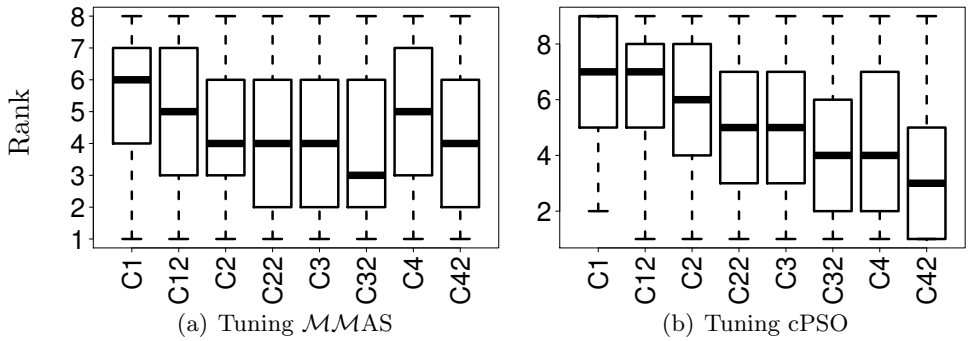
can also be observed as B22 improves over B2, and B32 improves over B3. The two restart schemes B2* and B3* appear to be outperformed by the default restart scheme B1* in most of the case studies. Applying second evaluation on historical best such as B1h does not perform as well as on the iteration best such as B12 or B13, as also observed in Section 6.3.1. Although B13 appears to be slightly better than B12 in the $\mathcal{MM}$AS problem class, B12 performs best in configuring the problem class cPSO, and appears to be relatively robust across the both problem classes.

Comparing the best setting B12 of BOBYQA to the best setting S1d of simplex, it appears that the best setting of the number of second evaluation configurations depends on the number of generated configurations in an elite qualification. As BOBYQA generates significantly less configurations than simplex within a restart, it requires less second evaluation configurations than simplex.

6.3.3 Post-selection CMA-ES settings

We follow the best basic setting of post-selection CMA-ES in Section 6.2.3 including the early qualification for CMA-ES, i.e., to qualify elites from each CMA-ES iteration. We study here further settings for post-selection CMA-ES configurators, including different population sizes, and whether to use the second evaluation technique. The population size λ of CMA-ES is set to $\lambda = 4 + a \cdot ln(d)$, where d is the dimension of the search space, and a is a parameter with a default value of 3.0 [98]. It is recommended in [98, page 25] to increase the population size. Therefore, we tested four values of a: $a = 3, 6, 12, 24$, and name the corresponding CMA-ES configurators C1, C2, C3, and C4, respectively. Besides, we also study the use of second evaluation technique, i.e., to determine the iteration best by an additional evaluation of the best candidate configurations in each iteration. Based on the previous study of second evaluation setting on BOBYQA and simplex, we have considered for CMA-ES only the second evaluation setting with two best candidate configurations, as each search iteration of CMA-ES generates significantly less candidate configurations than a restart of BOBYQA and simplex. The corresponding CMA-ES configurators with second evaluation are C12, C22, C32, and C42, respectively.

As shown in Figure 6.9, the post-selection CMA-ES variants that use the second evaluation always improve their corresponding counterparts without second evaluation. Besides, the default population size in C1* with $a = 3$ can always be improved by using a larger population in these configuration benchmarks. However, the best population



(a) Tuning $\mathcal{MM}$AS (b) Tuning cPSO

Figure 6.9: The ranking performance of eight post-selection CMA-ES configurators in JRace: each of C1, C2, C3, and C4 uses one of the four different population sizes from small to large; each of C12, C22, C32, and C42 uses second evaluation with one of the four population sizes, correspondingly.

setting differs in the two benchmarks: C32 with $a = 12$ appears best in tuning $\mathcal{MM}$AS, while C42 with $a = 24$ appears best in tuning cPSO. We further tested a larger population setting $a = 48$ in C52 for cPSO, and it is outperformed by C42.

6.3.4 Population variation strategy for CMA-ES configurators

As the best population setting of CMA-ES changes from benchmark to benchmark, we further explore the possibility of varying the population size during the run of CMA-ES, such that its performance is robust subject to the change of the search space. Such online adaptation of algorithm settings are classified into three types [218]: *prescheduled* approach, which varies the setting by a predefined schedule, *adaptive* approach, which varies according to a rule that depends on the performance of different settings measured during the run, and a *search-based* approach, which applies a search algorithm in the parameter space during the algorithm run.

In this work, we will focus on the prescheduled adaptation. A well-known prescheduled population variation scheme is the IPOP-CMA-ES [13], which increases the population size in each restart of CMA-ES. However, due to the limited number of search samples allowed for CMA-ES in an algorithm configuration experiment, most of the CMA-ES runs cannot have enough budget for restarts. This leaves us two choices: either enforce fast convergence of CMA-ES for population variation, or perform the population variation within each CMA-ES run. We chose the latter, and studied two basic prescheduled schemes:

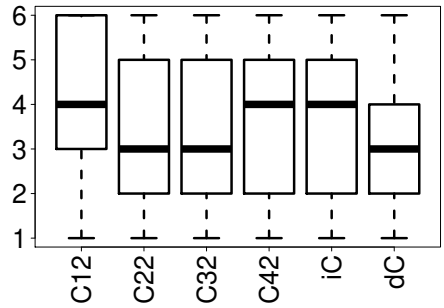
increasing population (iC), or decreasing population (dC). In order to be comparable to the fixed population size CMA-ES configurators in Section 6.3.3, we adopted the four levels of fixed population settings $a = 3, 6, 12, 24$, and change the population setting according to available budget. The iC starts with smallest population setting $a = 3$, and increases its population setting to the next level after every 25% of the budget is consumed; the dC starts with the largest population setting $a = 24$, and decreases to the next lower level after consuming every 25% of its budget. Besides, the second evaluation technique from Section 6.3.3 is also adopted in dC and iC.

As is shown in Figure 6.10(a-b), the decreasing population CMA-ES dC is in general the best-performing configurators for the both benchmark problem classes. Comparing to the best fixed-population CMA-ES configurators, i.e., the C42 in cPSO benchmarks, and C22 in the smallest three dimensions 2, 3, and 4 in $\mathcal{MM}$AS, and C32 in the largest two dimensions 5 and 6 in $\mathcal{MM}$AS, as shown in Figure 6.10(e-f), dC is usually comparable, if not better. dC performs especially well in the case studies with largest configuration budget, e.g., the largest budget levels 5, 6, and 7 as shown in Figure 6.10(c-d). Besides, decreasing population performs substantially better than the increasing population setting. iC only stands out in few case studies with lower budget levels.

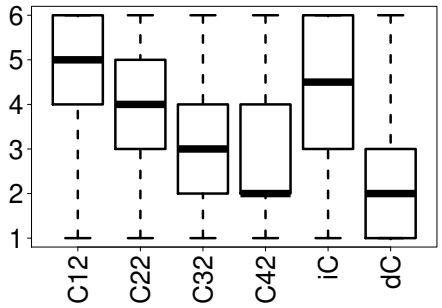
There are two possible reasons for the decreasing population to become a good post-selection setting for CMA-ES. On the one hand, this is due to the stochastic nature of the automatic algorithm configuration problem. The evaluation of the candidate configurations in each CMA-ES iteration becomes noisier from iteration to iteration, as the search gradually narrows down to a promising region. The probability of selecting the right elite out of the iteration configurations are higher when the number of candidates are fewer. On the other hand, due to the use of the post-selection mechanism, as the population slowly decreases, more elites will be qualified in the latter iterations than in the earlier iterations, while the candidates in the latter iterations are usually of higher quality as the search gradually converges to a more promising region.

6.3.5 Comparison to existing automatic configuration software

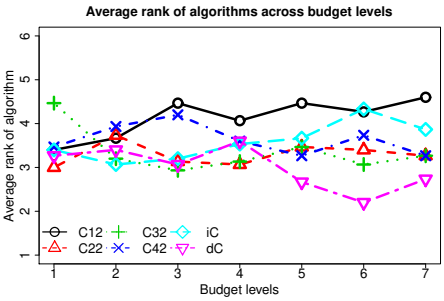
Firstly, JRace configurators are compared with existing automatic configurators. In particular, the following three implementations of iterated racing are compared:



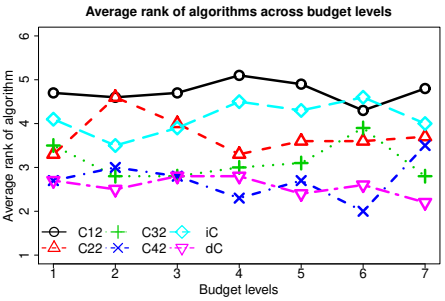
(a) $\mathcal{MM}$AS ranks



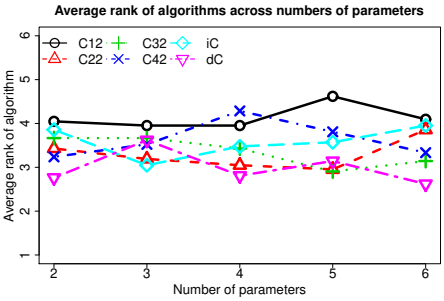
(b) cPSO ranks



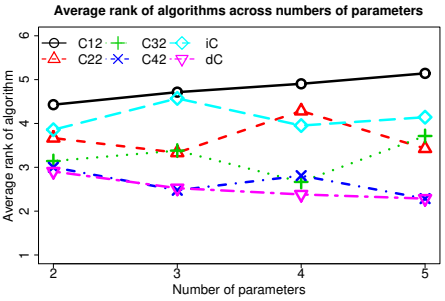
(c) $\mathcal{MM}$AS average rank by budget



(d) cPSO average rank by budget



(e) $\mathcal{MM}$AS average rank by dimension



(f) cPSO average rank by dimension

Figure 6.10: The ranking performance of seven configurators in JRace: four CMA-ES configurator with second evaluation (C12–C42), CMA-ES with increasing population (iC) and CMA-ES with decreasing population (dC).

- I15: The Java implementation of iterated racing in JRace;
- I11: The R implementation of iterated racing in irace package [151];
- I09: The R implementation of iterated racing in [37].

The JRace implementation in iterated racing I15 is compared with previous R implementations of the iterated racing to validate that it does not perform worse than existing implementations.

Note that the default setting of the irace package rounds every numerical parameter setting to two decimal places, while we round to two significant digit in our experiments. We adapted I11 to using rounding of two significant digits to be consistent to our experiments. Besides iterated racing, the uniform random sampling racing implemented in JRace, dubbed U, is also included as a baseline for comparison.

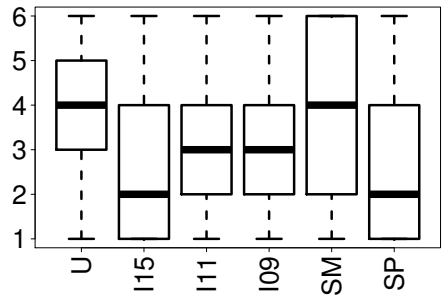
Besides, the following two versions of SMAC [112] are compared:

- SM: The default setting of SMAC;
- SP: The SMAC with post-selection using a fixed early restart of 30 evaluations per iteration, following the setting in [242].

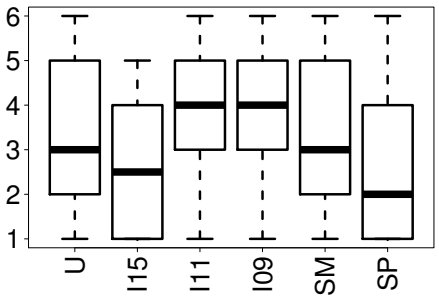
The ranking performance of the six configurators are shown in Figure 6.11. The JRace implementation of iterated racing I15 clearly outperforms I11 and I09. I11 and I09 performs similarly in $\mathcal{MM}$AS, as shown in Fig. 6.12(a-d); and I09 performs noticeably better than I11 in cPSO, especially in budget levels 5 to 7, as shown in Fig. 6.12(e-h). To rule out the factor whether the inferior performance of I11 is due to the change of the default setting of using two decimal places to two significant digits, we further set up experiments to compare I15 and I11 with two decimal places, and I15 is again substantially better than I11 across all case studies.

As for the two SMAC variants, SMAC with post-selection SP substantially outperforms the default setting of SMAC SM.

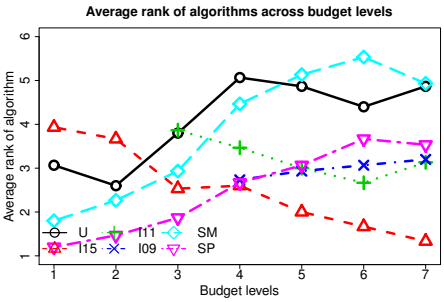
In the both benchmark problems, the two SMAC variants, especially SP, perform relatively well in low budget levels, namely budget levels 1 to 3. The iterated racing configurators are relatively poor in these lowest budget levels, and are usually outperformed by U-race. However, in budget levels 4 to 7, I15 stands out as the best-performing configurator, as also shown in Figure 6.12. The performance of SMAC (SM) is only comparable to U/Race. Surprisingly, the two older version of I/Race implementations, especially I11, performs poorly in tuning cPSO and is even outperformed by U/Race.



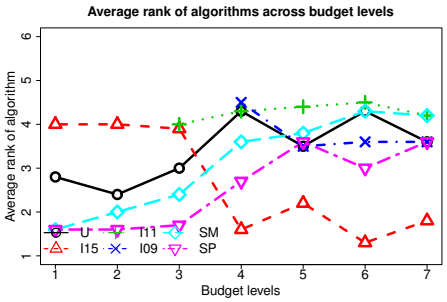
(a) $\mathcal{MM}$AS ranks



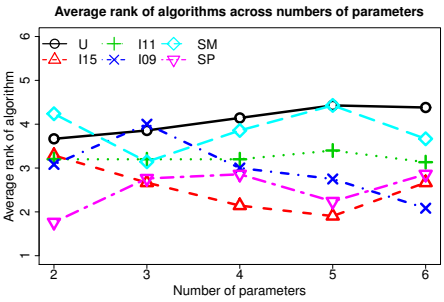
(b) cPSO ranks



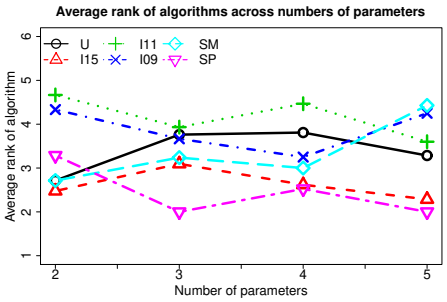
(c) $\mathcal{MM}$AS average rank by budget



(d) cPSO average rank by budget

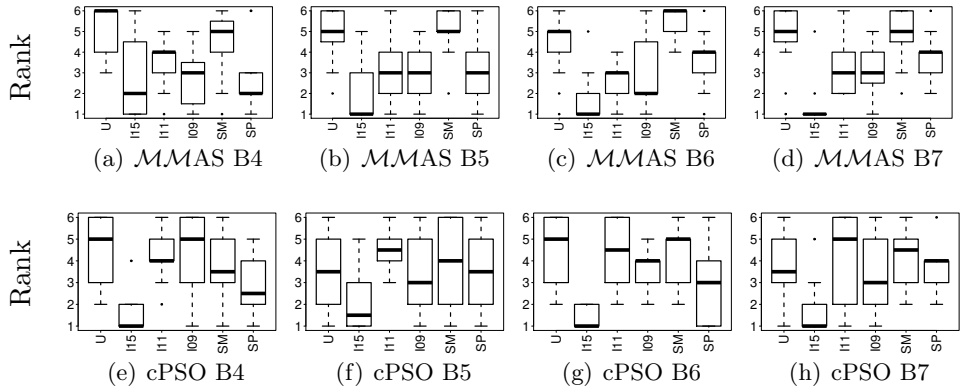


(e) $\mathcal{MM}$AS average rank by dimension



(f) cPSO average rank by dimension

Figure 6.11: The ranking performance of six configurators: the uniform random sampling with racing U; three iterated racing configurators I15, I11, and I09; two SMAC configurotors SM with default SMAC setting, and SP intergrating SMAC and post-selection.



(a) $\mathcal{MM}$AS B4 (b) $\mathcal{MM}$AS B5 (c) $\mathcal{MM}$AS B6 (d) $\mathcal{MM}$AS B7

(e) cPSO B4 (f) cPSO B5 (g) cPSO B6 (h) cPSO B7

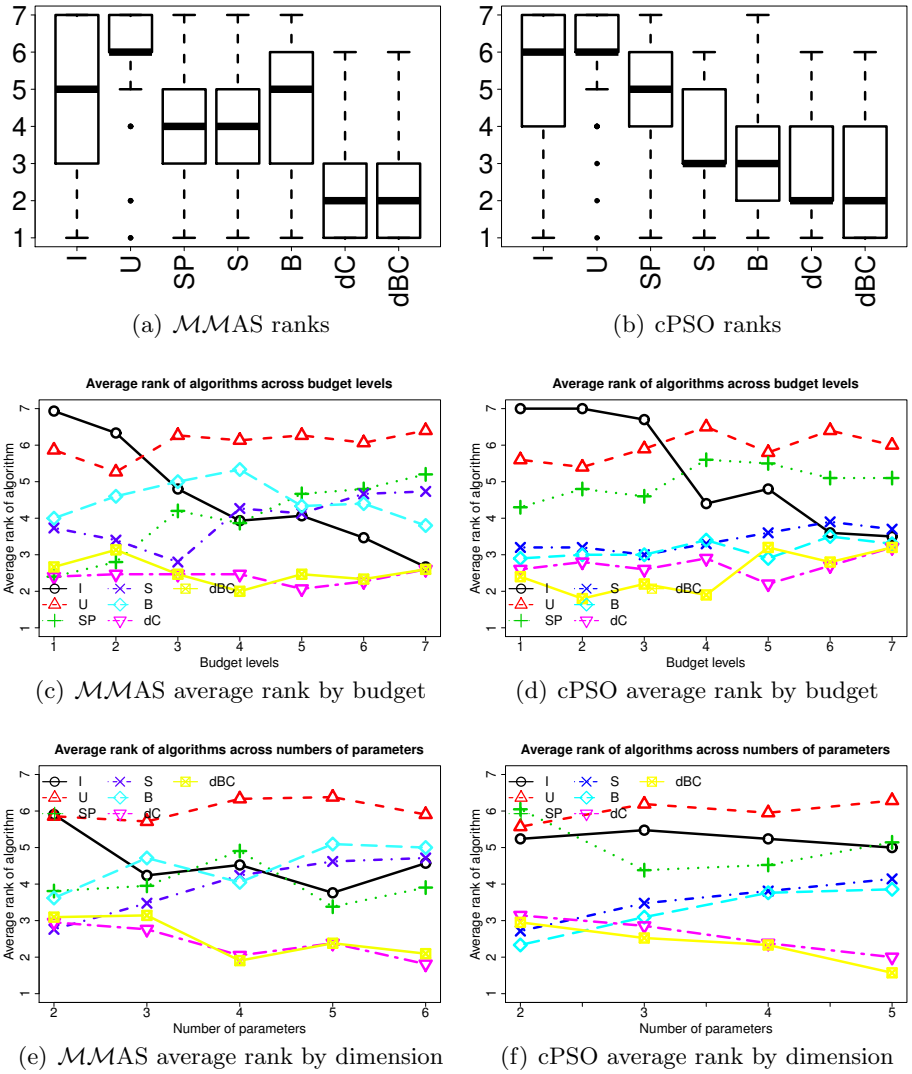
Figure 6.12: The ranking distribution of six configurators including uniform random racing U, three iterated racing configurators I15, I11, and I09, and original SMAC SM and SMAC with post-selection SP on (a-d) tuning $\mathcal{MM}$AS with budget levels 4 to 7; (e-h) tuning cPSO with budget levels 4 to 7.

6.3.6 Comparison of the best configurators

The best configurators are selected for the performance comparison on tuning $\mathcal{MM}$AS and cPSO. These best configurators include:

- I: The best iterated racing configurator I15 in JRace;
- U: The racing with uniform random sampled configurations as baseline;
- SP: The post-selection SMAC hybrid;
- S: S1d, simplex with second evaluation for d iteration best configurations.
- B: B12, BOBYQA with second evaluation for two best iteration configurations.
- B3: B32, BOBYQA with restart scheme 3 and second evaluation for 2 iteration best configurations.
- dC: The decreasing population CMA-ES with post-selection;
- dBC: The decreasing population CMA-ES, using BOBYQA in the first iteration to replace the uniform random sampling;

As shown in Figure 6.13, the decreasing population (DP) CMA-ES configurators, both dC and dBC, perform overall the best in almost all the problem dimensions and budget levels. The performance difference between dC and dBC are rather close, especially in $\mathcal{MM}$AS; in tuning



(a) $\mathcal{MM}$AS ranks (b) cPSO ranks

(c) $\mathcal{MM}$AS average rank by budget (d) cPSO average rank by budget

(e) $\mathcal{MM}$AS average rank by dimension (f) cPSO average rank by dimension

Figure 6.13: The ranking performance of best configurators in JRace: iterated racing (I), uniform random sampling racing (U), post-selection SMAC (SP), simplex with second evaluation of d iteration best (S), BOBYQA with second evaluation of two iteration best (B), BOBYQA with restart scheme 3 and second evaluation (B3), CMA-ES with decreasing population (dC), decreasing population CMA-ES starting from the best point of BOBYQA (dBC).

benchmark cPSO where BOBYQA configurator also performs well, replacing uniform sampling by BOBYQA as done in dBC may give a slight edge in determining the starting point for CMA-ES, especially when in the low budget levels. All the post-selection configurators have outperformed U/Race in all budget levels. The post-selection BOBYQA and Simplex are usually well performing for the smallest configuration case studies with 2 parameters to tune. The I/Race configurator performs better as the configuration budget increases. In the largest budget levels, and especially in the relatively more stochastic benchmark $\mathcal{MM}$AS, it is competitive with the post-selection configurators with BOBYQA, Simplex and SMAC, but is constantly outperformed by the DP-CMA-ES post-selection configurator.

6.4 Comparisons on further tuning benchmarks

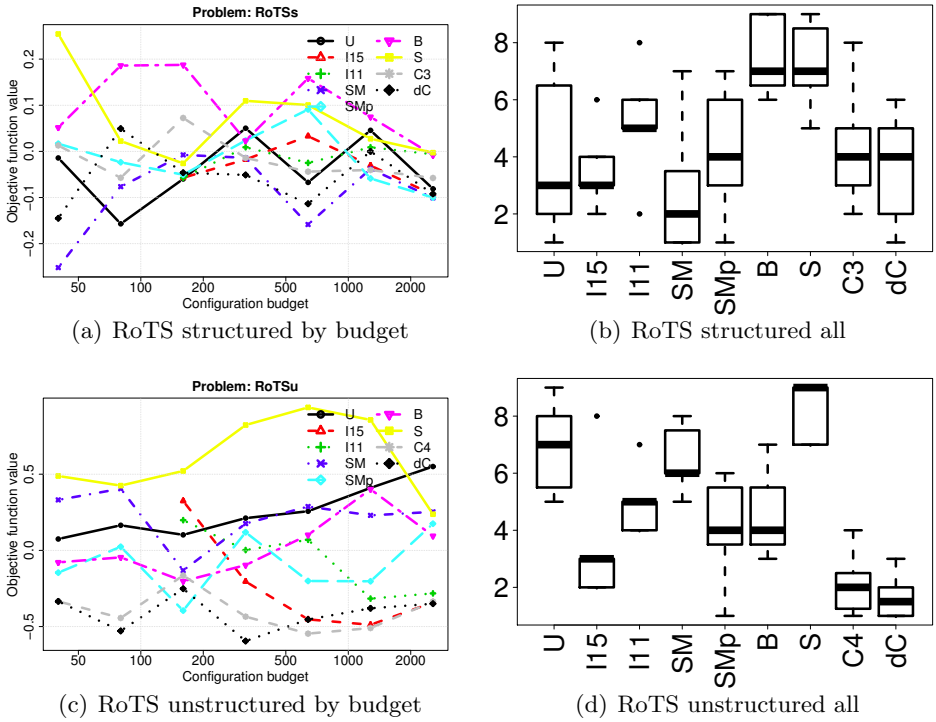
We further test our configurators on some further tuning benchmark problems, including configuring robust tabu search for quadratic assignment problem (QAP), configuring $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System with local search for QAP, and an increasing population CMA-ES with iterated local search for continuous functions.

6.4.1 Configuring robust tabu search

There are two instance sets: structured and unstructured. For each instance set, we first separated the 100 instances into training and testing set of 50 instances each. Then we replicated the 50 training instances into 300 training instances by assigning 6 different seeds to each instance, and also replicate 50 testing instances into 300 for testing. There are three parameters to be configured:

- m – the minimum factor for the tabu list length tt, to be configured within the range $[0.0, 3.0]$;
- w – the window factor for tabu list length tt to be configured in the range $[0.0, 3.0]$, such that $tt = (m + w)\times$ instance size;
- a – the length factor for iterations before forced aspiration (diversification mechanism), to be configured in the range $[0.5, 10.0]$.

The results is listed in Figure 6.14. The CMA-ES configurators, especially the decreasing population (DPop) CMA-ES configurator dC performs clearly best in the unstructured instance set, followed by the I/Race implementation I15 in the JRace package, which also clearly outperforms the other counterparts. The two winners dC and I15 from the unstructured instance set still performs among the best candidates



(a) RoTS structured by budget

(b) RoTS structured all

(c) RoTS unstructured by budget

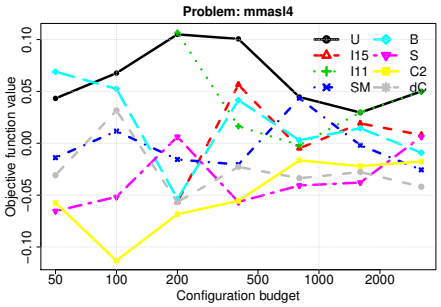
(d) RoTS unstructured all

Figure 6.14: The comparison of configurators: uniform random sampling racing (U), iterated racing in irace (I11) and in JRace (I15), SMAC (SM), post-selection SMAC (SMp), post-selection simplex (S), post-selection BOBYQA (B), post-selection CMA-ES with the best fixed population size (C), decreasing population CMA-ES (dC).

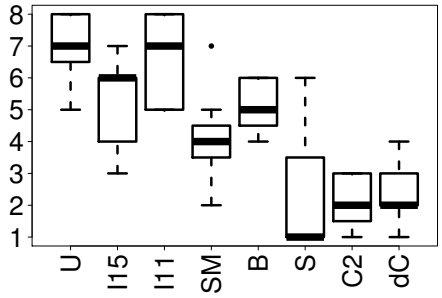
in the structured instance set, but they are slightly outperformed by SMAC. However, the performance of SMAC is rather fluctuating: it is the best performing configurator in the structured set, but is also among the worst performing ones in the unstructured set. In comparison, the DPop-CMA-ES configurator and I/Race are relatively robust in configuring Robust Tabu Search among different instance sets. Besides, the CMA-ES configurator with the best fixed population setting, i.e., C3 with a fixed population setting $a = 12$, and C4 fixing $a = 24$, are both outperformed by dC that uses a pre-scheduled decreasing population setting for CMA-ES.

6.4.2 Configuring $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System for QAP

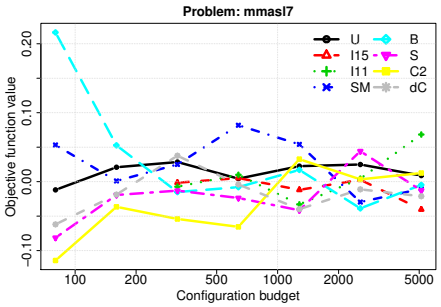
The next configuration benchmark is to configure the $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) algorithm with the 2-opt local search for the QAP problem. The same QAP instance sets used in the Section 6.4.1



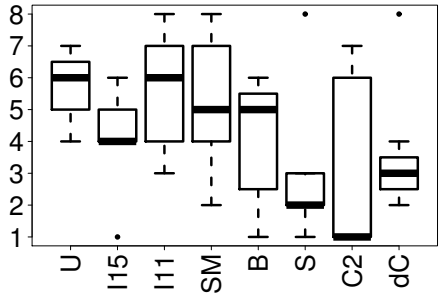
(a) acoqap structured 4 by budget



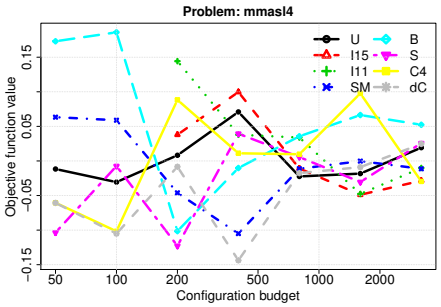
(b) acoqap structured 4 all



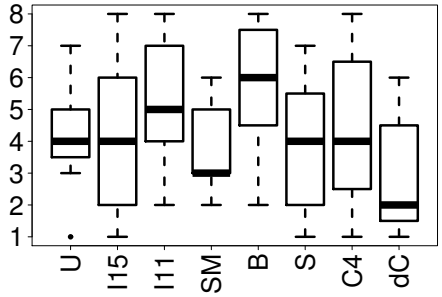
(c) acoqap structured 7 by budget



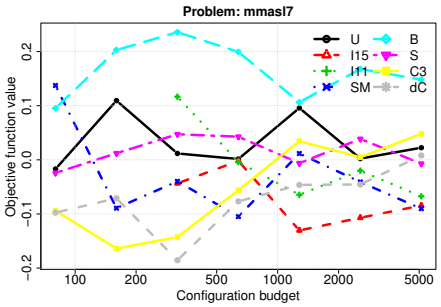
(d) acoqap structured 7 all



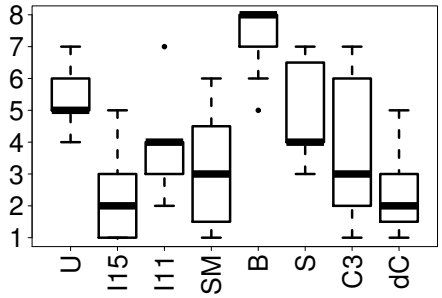
(e) acoqap unstructured 4 by budget



(f) acoqap unstructured 4 all



(g) acoqap unstructured 7 by budget



(h) acoqap unstructured 7 all

Figure 6.15: ACOQAP with local search on structured or unstructured instances with 4 or 7 parameters to tune.

are used here: a structured set and an unstructured set. Two case studies per instance set are used for this benchmark: using 4 parameters or 7 parameters of $\mathcal{MM}$AS. More specifically, the 4-parameter case study includes:

m – the number of ants;

α – the weight factor of pheromone trail;

ρ – the evaporation rate;

q_0 – the probability used in the pseudo-random proportional action choice rule.

Note that the β parameter in the standard $\mathcal{MM}$AS algorithm is not used here, since no heuristic information is used for solving QAP. For the details of the $\mathcal{MM}$AS parameters above please refer to the Section 2.2.1. The 7-parameter case study includes in addition:

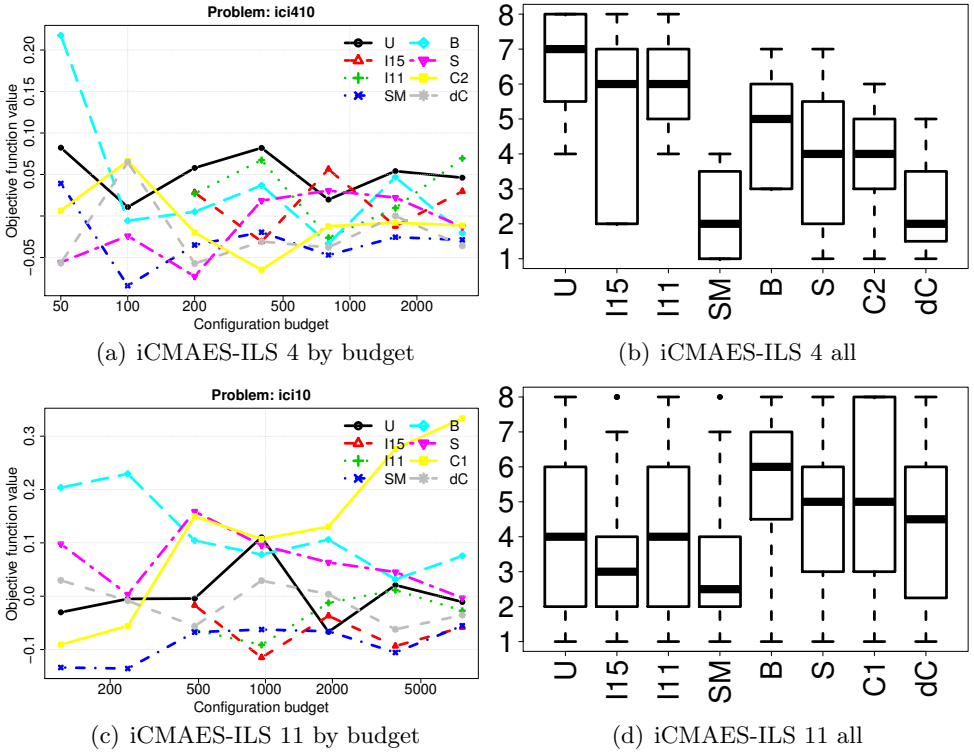
p – the pdec factor for updating the pheromone limits, to be configured in $(0, 1]$;

l – schedule length, the length of the update schedule in $\mathcal{MM}$AS, to be configured in $[5, 100]$;

r – which restarts if the branch factor is less than this restart branch factor, to be configured in $[1, 3]$;

More details about the parameters above can be found in [150].

The configurator comparison results on the structured instance set with 4 or 7 parameters to be tuned are illustrated in Figure 6.15(a-d), and configuring for the unstructured instance set is shown in Figure 6.15(e-h). The post-selection DPop-CMA-ES (dC) performs constantly best and robust across all the four case studies. It also constantly outperforms CMA-ES with the best fixed population setting, especially in the high budget levels. Other configurators also excel in certain individual case studies. For example, post-selection Simplex (S) performs especially well in the 2 case studies with structured set, while it is also one of the worst-performing in configuring 7 parameters for the unstructured set. Iterated racing implementation from JRace (I15) also stands out as one of the best in configuring seven parameters in the unstructured set. Besides, I15 constantly outperforms the other iterated racing implementation I11 in the R package of irace. It is interesting to note that comparing I15 and SMAC (SM), I15 appears to perform better in the 7-parameter case, while SMAC performs slightly better in the 4-parameter case.



(a) iCMAES-ILS 4 by budget

(b) iCMAES-ILS 4 all

(c) iCMAES-ILS 11 by budget

(d) iCMAES-ILS 11 all

Figure 6.16: iCMAES-ILS on rastrigin functions with 4 or 11 parameters to tune.

6.4.3 Configuring iCMAES-ILS

In the next benchmark, we configure a hybrid continuous optimization algorithm of increasing population CMA-ES with iterated local search (iCMAES-ils) for the class of Rastrigin functions. Two case studies are set up, one with four parameters to be configured, and one with 11 parameters to be configured. The results of the both case studies are shown in Figure 6.16. In the 4-parameter case study, the decreasing population CMA-ES configurator dC and SMAC are the best performing configuration algorithms. In the 11-parameter case study, the performance difference between different configurators are very close. SMAC and the iterated racing implemented in JRace (I15) are the best performing ones.

6.5 Summary

In this chapter, an in-depth investigation into the setting of post-selection mechanism is provided. The problem of finding good algo-

rithm configuration is stochastic: assessing the goodness of a configuration may require multiple evaluations. Post-selection is a stochasticity handling mechanism that divides the algorithm configuration process into two phases: an elite qualification phase that identifies a number of elite configurations, which will be evaluated in the second phase of elite selection to determine the best of the elites. We have found out in this chapter that having a careful elite selection process, the search algorithm in the elite qualification will require much less evaluations on each candidate configuration. Besides, elites can be qualified in different ways, e.g., by either enforcing fast restart and selecting the restart best, or selecting the iteration best from an iterative search algorithm. Each qualification can use alternating instance(s), and use a second evaluation mechanism to double-evaluate the best candidates for determining an elite. In particular, a post-selection CMA-ES algorithm stands out as the best and robust algorithm configurator across different benchmarks. The best setting of the post-selection CMA-ES uses most of the studied setting in this chapter, including low number of repetition ($nr = 1$), alternating instance, early qualification by iteration best, second evaluation. Besides, decreasing population (DPop) within one restart of CMA-ES exhibits especially good and robust performance across various configuration benchmarks against state-of-the-art algorithm configurators such as SMAC and various implementations of iterated racing.

Chapter 7

Conclusions and future work

In this chapter, we summarize the results and contributions presented in this thesis. We also offer some ideas for future work that we believe can contribute to the design and configuration of algorithm for hard optimization problems.

7.1 Conclusions

The algorithm configuration problem is a stochastic black-box mixed-variable optimization problem. An algorithm configurator usually contains two components: a search method that generates candidate configurations, and an evaluation method that compares and selects the best among candidate configurations. A common way to combine the search method and evaluation method for algorithm configuration is to follow the iterated selection framework: use an iterative search method to generate candidate configurations in each iteration, then apply the evaluation method to compare the incumbent with the newly generated configurations, so as to select the best to bias the search in the next iterations.

In this thesis, we have designed and developed a number of high-performing algorithms for the task of automatic algorithm configuration.

Iterated selection configurators: Iterated racing and MADS racing

We extended the statistical racing method [36] with an effective iterated search method for categorical and conditional parameters, and improved the performance of the first version of iterated racing [14] for numerical parameters. The extended version of iterated racing [37]

is the first version of a full functioning iterated racing configurator, and it is currently one of the most widely used automatic algorithm configuration tools, also known as `irace` [151].

Besides, we have also extended the family of iterated racing by hybridizing the statistical racing method with an existing search method mesh adaptive direct search (MADS) [236], which significantly improves the existing MADS configurator with fixed number of evaluation instances [9]. We further observed that the higher the variability of the parameter space, the higher the advantage of using racing as evaluation method. Besides, an incumbent protection mechanism is proved to be effective, and it substantially improves the performance of MADS/F-Race.

State-of-the-art black-box optimizer for searching parameter space

We further studied and adopted a number of state-of-the-art black-box optimizers, such as BOBYQA, CMA-ES, and MADS, to configure numerical parameters [235, 233]. In order to achieve better performance of search methods BOBYQA and CMA-ES, we have found that simple post-selection mechanisms to be quite effective in handling the stochastic nature of algorithm configuration problem. Post-selection is an alternative mechanism to iterated selection. It divides the algorithm configuration procedure into two phases: a first elite qualification phase that iteratively identifies a set of elite configurations which are then compared and selected carefully in the second elite selection phase. Our experiments based on the configuration benchmarks of swarm intelligence algorithms have shown that both BOBYQA and CMA-ES perform well in searching numerical parameter space. BOBYQA performs especially well in lower dimensions, while CMA-ES performs best and robust across different benchmark scenarios.

In-depth analysis of post-selection

The good performance of the post-selection has motivated us to investigate in more depth various post-selection settings [242]. We have found out that with a careful elite selection, typically very few (usually one in our work) evaluations per configuration is required to qualify elites. Each qualification can use alternating instance(s), and elites can be qualified by enforcing fast convergence of the algorithm and taking the restart best, or by early qualification that accepts each iteration best of an iterative algorithm such as CMA-ES. Besides, a second evaluation mechanism can be used to double-evaluate the restart-best

or iteration-best solutions before qualifying the elite. Our experiments have identified the CMA-ES configurator that uses low number of evaluations, alternating instance, early qualification, second evaluation, and decreasing population setting, performs particularly well across extensive benchmark configuration problems. In fact, it usually outperforms state-of-the-art configurators such as iterated racing and SMAC.

Summary

The algorithm configuration problem is a stochastic black-box mixed-variable optimization problem. An algorithm configurator usually contains two components that solves the two subtasks: a black-box search method that generates candidate configurations, and an evaluation method that compares the generated candidate configurations under stochasticity and selects the best among them. In this work, we define two frameworks to combine search method and evaluation method for algorithm configuration. Iterated selection uses an iterated search method to generate candidate configurations, and uses an evaluation method in each iteration to evaluate the candidates, and select the best to bias the further search. We classify that all the existing algorithm configurators into the iterated selection framework. We developed iterated racing for categorical and conditional parameters, as well as MADS and CMA-ES racing following the iterated selection framework. We proposed the post-selection framework to combine search and evaluation method in two phase: in the first elite qualification phase, a number of elite configurations are qualified from running a configurator, and in the second elite selection phase, an evaluation method is used to carefully select from qualified elite configurations. We introduced state-of-the-art black-box optimization algorithms such as CMA-ES, BOBYQA, and Nelder-Mead Simplex as search methods for algorithm configuration, following the post-selection framework. Post-selection settings such as few alternating instance in qualification, second evaluation to double-evaluate restart-best or iteration-best for elite qualification are shown to make the black-box optimizers become state-of-the-art algorithm configurators. In particular, a post-selection decreasing population CMA-ES is found to perform best and robust across extensive configuration benchmark problems with various dimensions and budget.

7.2 Future work

There are a number of research directions to extend the work presented in this thesis. Potential promising directions can be divided into two parts: the further methodological development of the automatic algorithm configuration algorithms, and the further application of the algorithm configurators developed in this work.

Methodological development

From the methodological aspect, an algorithm configurator consists of a search method, and an evaluation method. In the following, we will present the potential future work in these two parts.

Both iterated selection and post-selection are simple yet effective frameworks for extending established search methods to handle stochasticity. Established search methods for continuous optimization or mixed-variable optimization could be easily adapted based on these frameworks to become effective method for algorithm configuration or for other noisy optimization problems. Our work presented in this thesis, especially post-selection, focuses mainly on configuring numeric parameters, including real-valued or integer valued parameters. One interesting and fruitful future research direction is to apply post-selection with state-of-the-art optimization methods for categorical variables, to configure in addition the categorical algorithm parameters. Another challenge in the search aspect of an algorithm configurator is to effectively handle conditional parameters. To this end, a multi-level search approach can be a promising direction to pursue. The multi-level search divides the search of the parameter space by multiple levels depending on the conditional hierarchy, and searches in a top-down fashion. The search can first fix the value of the higher-level master parameters, determine the eligibility of the rest of the parameters, and then search in the space of eligible parameters.

Throughout the thesis, we have used statistical racing methods such as F-Race or T-Race as evaluation method for devising our algorithm configurators. There exists alternative in the field of simulation optimization for ranking and selection under noisy evaluations, known as optimal computing budget allocation (OCBA) [47]. It would be interesting to bridge the both fields of automatic algorithm configuration and simulation optimization by studying the methods from the both fields to their application problems. Potential limitation of applying OCBA directly on the algorithm configuration problem is that its default version makes certain distribution assumptions that may not hold in algorithm configuration problems. This may not be a major issue

in practice, as for example T-Race that also makes such distribution assumption works well on many algorithm configuration tasks. On the other hand, various transformation approaches such as rank transformation can be applied.

Application

Automated algorithm configuration has received more and more attention and has been applied more widely in the last decade. The application potential of the algorithm configurator is vast. Examples for successful applications of automated configuration tool to obtain high-performing algorithms for real-world optimization problems can be found in Appendix A and D. It is interesting to see automated algorithm configuration become a necessary tool for algorithm developers as well as algorithm users in the future. In fact, with the automated algorithm configuration tools, an algorithm developer can focus on the more interesting tasks such as defining algorithm components and parameter ranges, and leave the tedious parameter tuning to be carried out by computers. Besides configuring algorithms, the configurators developed in this work can be seen as general-purpose algorithms for stochastic optimization problems. For example, it may be applied for optimizing noisy functions such as those of the black-box optimization benchmarking workshop series [99], or further optimization problems with noisy objectives, such as the ones from the field of simulation optimization [84]. The positive results obtained in this work indicate that the directions outlined above are promising ideas to pursue.

Appendices

Appendix A

Automatically Tuned Iterated Greedy Algorithms for a Real-world Cyclic Train Scheduling Problem

A.1 Introduction

The problem we are tackling in this chapter arises in the strategic planning of the Deutsche Bahn AG (DB), the largest railway company in Germany. In particular, the problem arises in the context of a complex simulation tool that is used at DB to provide long-term simulations and future predictions of the load of the railway network. The tool can be seen as a chain of modules, where information between the modules is exchanged through data files.

Our particular problem arises in a module called *train scheduler*, which is responsible for the buildup of trains from cars. A train starts as soon as it is built, that is, when enough cars are assembled. Hence, this also means that the starting times of the trains do not follow a specific timetable; rather they follow the estimated customers demand or production. Since the locomotives are among the most expensive resources of the operation of railroad companies, their efficient scheduling is of high importance.

The locomotive scheduling, with which we deal here, can be characterized as a vehicle routing problem with time windows, a heterogeneous fleet of vehicles (due to different types of locomotives), and cyclic departures of trains. It also includes two important additional aspects: network-load dependent travel times and the transfer of cars between trains. In earlier work [85], we have developed an integer linear pro-

gramming (ILP) formulation of the problem and proposed a solution approach for the problem using a commercial ILP solver (ILOG CPLEX 10). In the corresponding experimental campaign [85], we noticed that (i) the minimum number of missed car transfers is relatively easy to find, (ii) with fixed starting times, relatively large instances with up to about 1 500 trips could be solved to optimality, (iii) the models that allowed the flexible choice of starting times within some predefined time windows made the problem much more difficult to solve: the size of the instances that could successfully be tackled by the commercial ILP solver (after some additional improvements such as providing good initial feasible solutions and problem-specific cutting planes) was limited to medium sized instances with a few hundreds of trips [85]. Despite the added computational difficulty, the hardest model with flexible starting times has highly desirable properties: allowing to vary the starting times within small time windows results in a considerable reduction of the required number of locomotives and, hence, a strong reduction in the total costs [85].

We tackle the most difficult problem variant studied in [85], namely the one that uses time windows and network-load dependent travel times. For this variant, which is also the most realistic and interesting one, heuristic algorithms are required to generate good quality solutions to large instance sizes, but also ILP approaches can benefit from improved initial upper bounds for medium sized instances. In this chapter, we therefore report on our research for improving upon the performance of the greedy construction heuristics that have been used in [85]. Our development process departs from these greedy heuristics, extending them step-by-step. The first extensions comprise a direct modification of the greedy heuristic by changing the way solutions are constructed. Next, we extend the construction heuristic to an iterated greedy (IG) algorithm [197] and further hybridize it with a simple iterative improvement algorithm. Our experimental results clearly show that for computation times ranging from a few seconds to a few minutes on current CPUs, very strong improvements over the initial greedy heuristic can be obtained. A further hybridization of the IG algorithm with ant colony optimization (ACO) algorithms, however, gave rather mixed results and no further significant improvement in performance.

A.2 The freight train scheduling problem

It is convenient to first describe the nomenclature used in the context where this freight train scheduling problem arises.

A.2.1 Problem setting

Cars. A *car* is the smallest unit to be moved; cars have to be moved from a source to a destination within the railway network. A train is composed of a set of cars. Large customers require the transport of large amounts of goods so that they order whole trains; in such a case, the route of the cars is the same as the route of the train. Individual or small sets of cars are used by smaller customers. Several such cars are then assembled into a train, moved to some intermediate destination, which is called *shunting yard*, where trains are disassembled and re-assembled into new trains. We assume that that the place and timing of these transfers is known for individual cars. Note that the scheduling of locomotives needs to take care that these transfers remain feasible.

Trains. A *freight train* (also called *active trip*) consists of several *cars*. Each train has a *start* and a *destination*, which are goods stations or railroad shunting yards, and *starting times* and *arrival times*. Times can be either fixed times or be taken from some interval. In our case, trips start cyclically every 24 hours. The *trip duration* is the difference between start and arrival times. Trains vary in lengths and weight and, depending on these two characteristics, different needs on the driving power of locomotives arise. Typically, a single locomotive is enough and only rarely two pulling locomotives are required. The handling of a train involves attaching a locomotive to a train at the start and the decoupling of the locomotive at the destination. For both *coupling* processes, some time (up to 30 minutes, depending on the trip) is required for technical checks or refueling.

Locomotives. The around 30 different models of locomotives that are used at DB share many similarities and so they can be classified into a small number of different *classes*. Differences between the classes concern mainly the driving power and the motor type, diesel or electrical. Electrical locomotives can only be used on electrical tracks, whereas diesel locomotives, in principle, can drive everywhere. However, diesel soots the electrical wires, so one wants to avoid their deployment on such tracks. Hence, it is only possible to assign such locomotives to trains that have a sufficient power and the right motor type for the track.

Deadheads. A locomotive is either active, that is, pulling a train, or deadheading, that is, driving alone, without pulling a train, from the destination station of one train to the start of another train. The distance between these points and the class of locomotive determine the duration of a deadhead trip.

A.2.2 Formulation of the problem

Our locomotive scheduling problem can be formulated as a cyclic Vehicle Scheduling Problem (CVSP), more specifically, as a CVSP with hard time windows and a heterogeneous fleet of vehicles (locomotives); we use the abbreviation CVSPTW in what follows. The locomotives differ in starting cost and capability, that is, the subset of customers that can be potentially served by the locomotives. Instead of starting and terminating at a depot as in standard vehicle routing problems, the locomotives are scheduled in a cyclic fashion every 24 hours. Given is also a set of customers, more specifically in our case a set of trips, that expect exactly one of the locomotives to pull the train obeying restrictions on the starting and arrival times (or time windows).

More formally, the problem can be described as follows. We denote by $\mathcal{V}$ the set of active trips and by $\mathcal{B}$ the set of locomotive types. With each trip is associated a list of locomotive types which can serve it. $\mathcal{A} = \mathcal{V} \times \mathcal{V}$ denotes the set of potential connections of pairs of trips. Note that trips are *cyclic*, which for our problem means that the trips occur every day.

The time intervals for the starting and arrival times of trips are input data. In practice, the train scheduler of the tool chain defines fixed starting and arrival times. In our model, we let the starting time t_i of trip i vary in a time window $\underline{t}_i \leq t_i \leq \bar{t}_i$. The times are defined as the minutes passed since some time zero and, hence, one has that $0 \leq t_i \leq 1439$ (a day has 1440 minutes). Note that a further constraint imposes that a trip has to start on the first day; hence, in case $\bar{t}_i$ is larger than 1439, the time interval $\underline{t}_i \leq t_i \leq \bar{t}_i$ is replaced by two time intervals $([\underline{t}_i, 1439] \cup [0, \bar{t}_i]) \cap \mathbb{Z}$.

Moreover, trains $i, j \in \mathcal{V}$ that require a car transfer from one to the other need to be synchronized. Denote $\mathcal{P}$ the set of pairs (i, j) where cars transfer from i to j. We assume $\mathcal{P}$ is valid and given by the earlier computation module. For all $(i, j) \in \mathcal{P}$ the transferred car must be picked up by j within 12 hours after the arrival of i at the shunting yard.

Our model includes network-load dependent travel times. This is important since typically at day time the average traveling speed of a freight train is lower than during nighttime; the main reason is that passenger trains have a higher priority than most freight trains, leading to frequent waiting times for the latter. To model this aspect, we partition a day into a number of time slices $\mathcal{H} = \{1, \ldots, H\}$, that is, $[0, 1439] = \dot{\bigcup}_{h \in \mathcal{H}} [\underline{\psi}_h, \overline{\psi}_h]$. The starting time of a train then falls into one of the slices and the travel times are considered accordingly.

The total trip and deadhead durations $\delta_{b,(i,j)}$ for a locomotive of

type b that serves both trips i and j are computed as

$$\delta_{b,(i,j)} := \delta_i^{trp} + \delta_i^{uncpl} + \delta_{b,(i,j)}^{dhd} + \delta_j^{cpl}, \tag{A.1}$$

where δ_i^{trp} denotes the trip duration, that is, the time the locomotive is active while pulling train i; δ_i^{uncpl} denotes the time for uncoupling the locomotive from the train at the arrival; $\delta_{b,(i,j)}^{dhd}$ denotes the time for deadheading from the end of i to the start of train j; δ_j^{cpl} denotes the time for coupling the locomotive to the train at the start of j.

Note that the driving time δ_i^{trp} is assumed to be independent of the actual locomotive class, whereas the deadhead time $\delta_{b,(i,j)}^{dhd}$ is class dependent (since diesel and electrical might use different routes). In the case of network-load dependent travel times, δ_i^{trp} gets replaced by $\delta_{i,h}^{trp}$, which gives the trip duration of train i when starting in slice h. Finally, in the case of car transfers between trains, the time for shunting a car from i to j, $\delta_{i,j}^{shnt}$, needs to be taken into account for the computation of the trip and deadhead durations.

The goal of the problem is to compute feasible starting and arrival times of the trains such that the operational costs are minimized. More concretely, two cost components are considered. Let γ_b^{cls} be the cost for a locomotive of class b, $\gamma_{b,(i,j)}^{dhd}$ by the cost of a deadhead trip from i to j for locomotive class b. Typically, the following ordering holds between the costs: $\gamma_b^{cls} \gg \gamma_{b,(i,j)}^{dhd}$, that is, the most important objective is to reduce the number of used locomotives, and then at a subsidiary level, to minimize the total distance of all the deadhead trips.

A mathematical description of the constraints and objective as a mixed-integer programming model can be found in [85].

Finally, recall that in this chapter we tackle the most difficult variant of the train scheduling problem from [85], that is, the variant with time windows (instead of fixed starting and arrival times) and with network-load dependent travel times, in part because the variant without these two features were well solved by the ILP approach. For the approach described in this chpater, we do not consider the preliminary step of the minimization of the number of missed car transfers as in [85], but we fix the set of feasible car transfers as determined through some preliminary computation. The reason is that the minimum number of missed car transfers can, in practice, easily be done by an ILP approach or by a greedy heuristic [85]. Hence, we focus on the minimization of the operational costs, that is, the costs for the locomotive usage and the deadhead trips.

A.2.3 Benchmark instances

For our computational tests we used eight instances, five of which were also used as test instances in [85]. The names and the characteristics of these five instances are as follows.

- `A`, with 42 trains, 3 locomotive classes
- `B`, with 82 trains, 3 locomotive classes
- `C`, with 120 trains, 4 locomotive classes
- `KV`, with 340 trains, 6 locomotive classes
- `EW`, with 727 trains, 6 locomotive classes

From each of these instances, we obtain additional ones by imposing different ranges for the time windows. In particular, for each of the fixed starting and arrival times we allow symmetric time windows centered at these values from the set $\{\pm 0, \pm 10, \pm 30, \pm 60, \pm 120\}$ minutes.

An additional three instances have been used for tuning the parameters of the heuristic algorithms. The characteristics of these instances are as follows.

- `R1-101-3`, with 101 trains, 3 locomotive classes
- `R2-137-6`, with 137 trains, 6 locomotive classes
- `R3-295-5`, with 295 trains, 5 locomotive classes

Since we used iterated F-race [14], an automated tuning tool, it was desirable to derive a larger set of instances from these three. Hence, we selected the time windows as follows. For `R3-295-5`, `R2-137-6`, and `R1-101-3`, we selected as time windows the integers from $[0, 150]$ that can be divided by 2, 3, and 6, respectively; this results in 75, 50, and 25 instances, respectively. Note that the larger the base instance the more derived instances are available. This was done in order to bias the choice of the parameter settings towards better solving large instances.

A.3 Greedy algorithms

For effectively tackling the CVSPTW variant with network-load dependent travel times, an essential ingredient for the ILP-based approach [85] has been a randomized PGreedy-type algorithm [87]. This algorithm, called `g-CVSPTW`, forms the basis for the further developments and it is described next.

A.3.1 The g-CVSPTW heuristic

g-CVSPTW assumes a fixed set of car transfers as input and it seeks to minimize the operating costs. g-CVSPTW iteratively generates cyclic schedules for locomotives repeating the following series of steps until all trips are scheduled.

1. **Step 1: Locomotive selection.** First, a locomotive class is chosen. The choice is based on two factors: (i) the capability N_b, which is the number of trains that can be served by a locomotive from class b, and γ_b^{cls}. Each locomotive class is scored by the ratio N_b/γ_b^{cls}. A locomotive class with the highest ratio is chosen.

2. **Step 2: Start trip selection.** Among the still unscheduled trips, we select one with the highest number of deadhead trips that use a locomotive of the same class b as selected in step 1; ties are broken randomly.

3. **Step 3: Starting time selection.** Next, the starting time t_i of the trip is taken from the interval $[\underline{t}_i, \overline{t}_i]$. t_i is chosen such that the train arrival time is the earliest possible. (Recall that since we have netload dependent driving times, we need to subdivide the interval $[\underline{t}_i, \overline{t}_i]$ in dependence of the time slices.) After fixing the starting time, the impact of this decision on other starting time windows needs to be propagated. This update is necessary, if there is some trip j with (i, j) or $(j, i) \in \mathcal{P}$. For this task, a constraint propagation algorithm was implemented to propagate the effect of the decisions through the network.

4. **Step 4: Deadhead trip selection.** Next, a shortest deadhead trip to the start of the next train is chosen for the locomotive.

g-CVSPTW starts with steps 1 and 2 and then cycles through steps 3 and 4 until no trip can be added anymore to the locomotive's schedule. In particular, the construction is stopped if no trip can be accommodated within the 24 hours cycle (recall that g-CVSPTW uses a maximum 24 hour cycle length for each locomotive).

Instead of making deterministic choices in steps 1, 2, and 4, g-CVSPTW actually uses randomized greedy values, in spirit similar to the noising method [46]. Before doing the choice, the concerned greedy values are multiplied by a value that is generated randomly according to a uniform distribution in $[10000 * (1 - \mathit{NOISE}), 10000]$, where NOISE is a parameter from the range $[0, 1]$ that indicates the degree of noise in the random selection. Decisions are then taken deterministically based on the perturbed greedy values, breaking remaining ties randomly. The

main rationale for using the randomization in the greedy heuristic is to allow the generation of a large set of different solutions.

A.3.2 Modified greedy heuristic

As a first step in the road to improved heuristic algorithms, we modified `g-CVSPTW`, based on the previous experience with it, resulting in the `mg-CVSPTW` heuristic.

The *first modification* is to add a parameter v to weigh the importance of the locomotive capability and we now use the ratio N_b^v/γ_b^{cls} for the computation of the greedy values.

The *second modification* derives an additional greedy value for biasing the solution construction. It is based on the observation that the choice of a trip restricts in very different ways the number of possible trips that feasibly could be added to a locomotive cycle. In the extreme case, locomotives contain only one trip, since it is infeasible to assign more trips within the 24 hour cycle of a locomotive. (We call such trips *isolated*.) Hence, we define for each trip i its *sociability score* s_i, which counts the number of other trips that could be served by a locomotive in a same cycle with trip i; for isolated trips, this score is zero.

A *third modification* is to change the order of steps 1 and 2 in `g-CVSPTW`, together with a more deterministic solution construction. For this modification, now trips are pre-ordered and chosen deterministically in the given order. The ordering is defined in a lexicographic way by (i) considering the number of locomotive classes able to serve the trip (the smaller, the higher the rank) and (ii) the sociability score of each trip (the smaller the score the higher the rank). In the construction process, `mg-CVSPTW` first chooses the starting trip in non-increasing order of the ranks and then fixes the locomotive class. There are two reasons why this modification was deemed to be helpful. Firstly, in this way the most constraining decisions are done first, postponing the ones with more flexibility; secondly, by selecting among a smaller set of candidates, we expected to speed-up the construction process. In fact, we observed that the computation times for the solutions construction decreased by a factor of about five.

Once the first trip and the locomotive class selection is done, the modified greedy heuristic continues iteratively with steps 3 and 4 as `g-CVSPTW`.

A.3.3 Locomotive type exchange heuristic

A common way of improving solutions returned by a construction method is to improve them by some type of local search. Since the

```
procedure Iterated Greedy
    s = GenerateInitialSolution
    repeat
        s_p = DestructionPhase(s)
        s' = ConstructionPhase(s_p)
        s'' = LocalSearch(s')          % Optional local search phase
        s = AcceptanceCriterion(s, s'')
    until termination condition met
```

Figure A.1: Algorithmic scheme of iterated greedy with optional local search phase.

development of very effective local search algorithms can be rather time consuming, we limited the local search to locomotive class exchanges: the idea is to try to replace the locomotive of each tour by a cheaper one. The local search procedure works as follows. First, all locomotive classes are sorted according to non-decreasing costs into a list $c = \langle c_0, c_1, c_{|\mathcal{B}|} \rangle$. Then, for all cycles that use a locomotive of class i we check whether the locomotive of cost c_i can be replaced by a cheaper locomotive, that is, by one of costs $c_0, \ldots, c_{i-1}$ (starting with index 0). If this is possible, the substitution is applied and the next cyclic tour is considered.

A.4 Iterated greedy algorithms

As said previously, a main motivation for the randomization of the greedy steps in `g-CVSPTW` and in `mg-CVSPTW` is to allow to construct many different solutions. However, this results in independent applications of a heuristic, an approach which for many problems does not lead to excellent performance. Hence, we use *Iterated Greedy* (IG) as another way for iterating over construction heuristics. The central idea of IG is to avoid to repeat a greedy construction from scratch, but rather to keep parts of solutions between successive solution constructions [108, 197]. This is done by first destructing a part of a current complete solutions and then to reconstruct from the resulting partial solution a new complete solution. This new solution is accepted as the new incumbent solution according to some acceptance criterion. For a generic algorithmic outline of IG see Figure A.1.

The IG algorithm adds additional parameters. Directly linked to the IG heuristic is the choice how much a complete candidate solution should be destructed; here, we use the parameter *destruction ratio d*.

It determines the number of locomotive cycles that are removed from the current solution: if N_l is the number of locomotives in the current solution, then $\lfloor d \cdot N_l \rfloor$ randomly selected locomotives are removed from the current solution. For the re-construction of a complete solution (and the construction of the initial solution), we use the `mg-CVSPTW` heuristic, which proved to be more effective than `g-CVSPTW`.

The acceptance criterion uses the well-known Metropolis rule known from simulated annealing. A candidate solution that is better or of equal quality to the current candidate solution is deterministically accepted; a worse solution is accepted with a probability given by $\exp\{-\Delta/T\}$, where Δ is the cost difference between the new and the current solution and $T = \nu \cdot f(s^*)$, where s^* is the best solution found so far. Note that in the acceptance criterion no annealing is used.

The resulting algorithm we call `IG-CVSPTW`. Our algorithm uses one additional new feature, which was previously not used in IG algorithms. Instead of only reconstructing one complete solution, `IG-CVSPTW` generates from one partial solution $I_r \geq 1$ new solutions. This is done since generating the correct time windows for the partial solutions is relatively computation time intensive due to the constraint propagation for narrowing the time windows.

Clearly, as for `g-CVSPTW`, it is also desirable for `IG-CVSPTW` to consider the hybridization of the algorithm with a local search, and, hence, we also study this feature. Adding local search to IG is straightforward by improving each reconstructed solution, as indicated by the optional $\mathsf{LocalSearch}(s')$ procedure in Figure A.1. As for `mg-CVSPTW`, we use the locomotive class exchange local search.

A.5 Experimental results for greedy and iterated greedy

In this section, we report on the experimental results and the range of improvements that we obtained with the extensions of the greedy heuristic. Since the clearest differences in performance have been observed on the two largest base instances, we focus on these two in the discussion of the experimental results.

A.5.1 Experimental setup

All the codes were written in Java and share the same data structures. The code was compiled and executed using JDK 1.6.0_05. The experiments were run on computing nodes each with two quad-core XEON

algorithm	parameter	range	selected value	
			without LS	with LS
`mg-CVSPTW`	*NOISE*	$[0, 1]$	0.67	0.15
	v	$[0, 10]$	1.8	6.3
IG	*NOISE*	$[0, 1]$	0.59	0.57
	v	$[0, 10]$	9.4	2.9
	d	$[0.01, 1]$	0.12	0.074
	ν	$[0, 0.05]$	0.046	0.019
	I_r	$[1, 50]$	9	25

Table A.1: Parameter settings obtained from iterative F-race for greedy and IG heuristics. See the text for more details.

E5410 CPUs running at 2.33 GHz and 8 GB RAM. Due to the sequential implementation of the algorithms, each execution makes only use of one single core.

Each algorithm was first tuned by an automatic tuning algorithm called iterated F-race [37]; the only exception is `g-CVSPTW`, for which the pre-fixed setting of *NOISE* was used. The three instances `R1-101-3`, `R2-137-6`, and `R3-295-5` were used as training instances for tuning, using the time windows as explained in Section A.2.3. Before tuning, the order of the instances is randomized. The computation time for all tuning instances was set to 300 CPU seconds, which corresponds approximately to the time `g-CVSPTW` requires to generate 10 000 solutions for base instance `R-295-5`. Iterative F-race is then run for a maximum of five iterations and in each iteration 100 candidate configurations are generated, that is, a total of 500 configurations for each algorithm are tested. Table A.1 gives the range of values considered and the finally chosen ones by iterative F-race.

A.5.2 Experimental results

For an experimental evaluation of the five algorithms, we have run each of them 30 times on the 25 test instances (for each of the five base instances 5 settings of time windows have been considered). Each algorithm was run for the same maximum computation time as required by `g-CVSPTW` to construct 10 000 solutions. In Figure A.2, we show for each of the algorithms the development of the average solution quality across 30 independent trials over time. The plots show that initially the solution quality improves very quickly and then the curves flatten off. However, especially the IG variants still show further improvements over time. The differences of the average solution quality reached by the algorithms are rather strong with the best performing one being `IG-LS-CVSPTW`. The boxplots, which are given in Figure A.3, also indi-

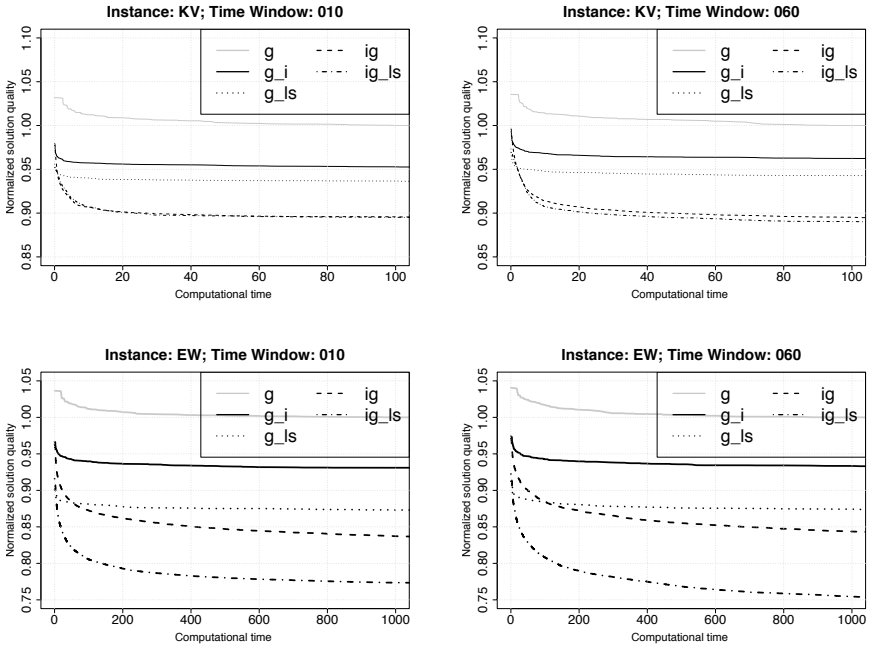


Figure A.2: Development of the solution quality over time for `g-CVSPTW` (g), `mg-CVSPTW` (g_i), `mg-LS-CVSPTW` (g_ls), `IG-CVSPTW` (ig), and `IG-LS-CVSPTW` (ig_ls) for instances `KV-10` (top left), `KV-60` (top right), `EW-10` (bottom left), and `EW-60` (bottom right); the numbers after the instance identifier indicate the time window range chosen. The variability of the averages is low as shown for the solutions after 100 CPU seconds (instances `KV-*`) and 1000 CPU seconds (instances `EW-*`) by the boxplots in Figure A.3.

cate that the variability of each algorithm's final performance is very low. Hence, the significance of the differences could also be confirmed by statistical tests: all pairwise differences between the final solution quality reached by the algorithms are statistically significant according to the Wilcoxon test using Holms correction for the multiple comparisons; the only exception is for instance KV-10, where the differences between `IG-CVSPTW` and `IG-LS-CVSPTW` were not significant. In fact, also on few smaller instances, the differences between `IG-CVSPTW` and `IG-LS-CVSPTW` are not significant.

An important result of this comparison is that the improvement of the computational results over the repeated application of `g-CVSPTW` is very strong. For most instances, the final costs found by `IG-LS-CVSPTW` were about 10% to 20% lower than those reached by `g-CVSPTW`. Typically, the same final solution quality as that of `g-CVSPTW`, was reached by `IG-CVSPTW` or `IG-LS-CVSPTW` in less than a hundredth of the maxi-

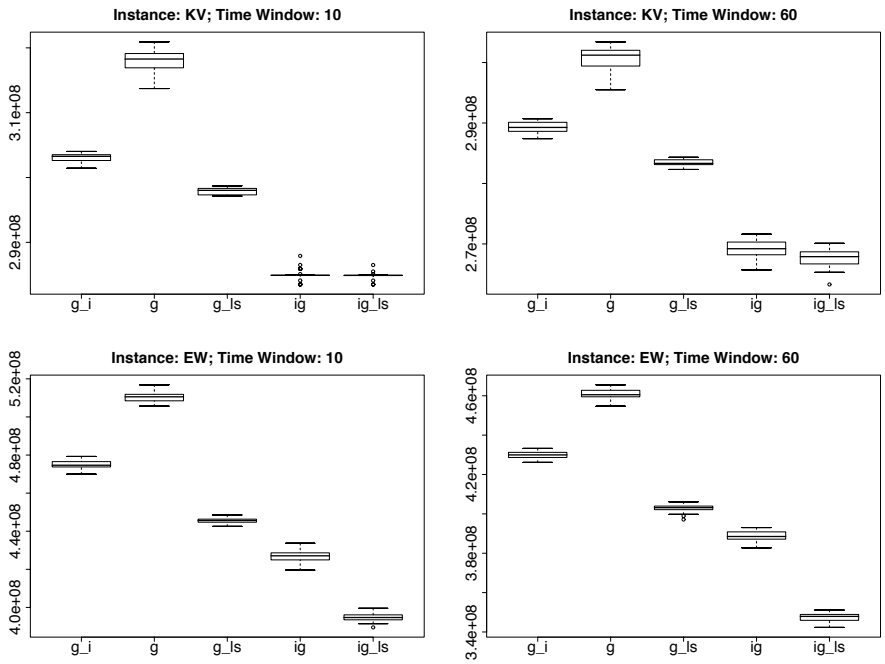


Figure A.3: Boxplots of the final performance after 100 CPU seconds for instances `KV-10` (top left) and `KV-60` (top right), and 1000 CPU seconds for instances `EW-10` (bottom left) and `EW-60` (bottom right). The y-axis shows the solution cost reached. For the explanation of the abbreviations see the caption of Figure A.2.

mum time taken by `g-CVSPTW`.

A.6 Iterated Ants

The strong improvements by the IG algorithms motivated us to consider extensions that might yield further performance improvements. Given the strong importance of the constructive part, we decided to consider a hybrid between two constructive SLS methods: IG and ant colony optimization (ACO) algorithms; the resulting hybrid algorithm is called *iterated ants* [225]. The central idea of iterated ants is to make each ant in the ACO algorithm follow the steps of the IG algorithm. The construction phase of the IG algorithm is then biased, as usual in ACO algorithms, by pheromones and heuristic information. The ACO approach we followed was based on a version of $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) that uses the pseudo-random proportional action choice rule known from Ant Colony System (see [217] for more details) and each ant follows the steps of the modified greedy construction heuristic. The pheromone trails have been associated to the choice of the next

trip to be added to a locomotive's schedule; that is, a pheromone trail τ_{ij} refers to the desirability of serving trip j after having done trip i; the heuristic information is the inverse of the length of the deadhead trip from i to j. This choice matches the usual ACO approach to the TSP.

Without going into further details on the parameters involved (which were the usual ones arising in the ACO context plus the ones relevant for the IG part), we summarize our main observations. The first is deceptive, in the sense that the final configuration returned from iterated F-race had the importance of the pheromone trails set to zero (Note that the action choice rules in ACO algorithms are positively biased to choose components j for which the term $\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}$ is high, where α is a parameter that weighs the influence of the pheromone trails. Setting $\alpha = 0$ actually has the effect of not considering the pheromone trails at all in the solution construction.) However, the parameter weighting the influence of the heuristic information was set to its maximum value in the considered range. This is true for the variants with and without local search. Nevertheless, it is still interesting to compare the performance of the "iterated ants" algorithms to the IG ones since both use different rules for constructing complete candidate solutions. Exemplary results for this comparison with plots of the development of the solution quality over computation time are in Figure A.4.

The overall best performing variant is `IG-LS-CVSPTW`. Across the 25 test instances, it is statistically better than "iterated ants" with local search on 12 instances and on none statistically worse. The situation is less clear, when considering the variants without local search; here on some instances `IG-CVSPTW` performs better than the "iterated ants" variant (such as the `KV` instance), while the opposite is true on others (such as on the largest instance tested here, `EW`).

A.7 Discussion

A direct comparison of `IG-LS-CVSPTW` to the performance of the best ILP-based approach in [85] would certainly be interesting. However, a direct comparison is, at least with the current heuristic algorithm code, not straightforward because of some minor differences in the constraints considered. In fact, in the greedy heuristics, a maximum cycle length of 24 hours is imposed for the schedule of each locomotive, which is not done in the ILP formulation: in the ILP formulation a schedule cycle of a locomotive can be an arbitrary multiple of 24 hours. Hence, the heuristic solutions generated within the 24 hour limit for each locomotive is a subset of the ones considered by the ILP approach. Anyway,

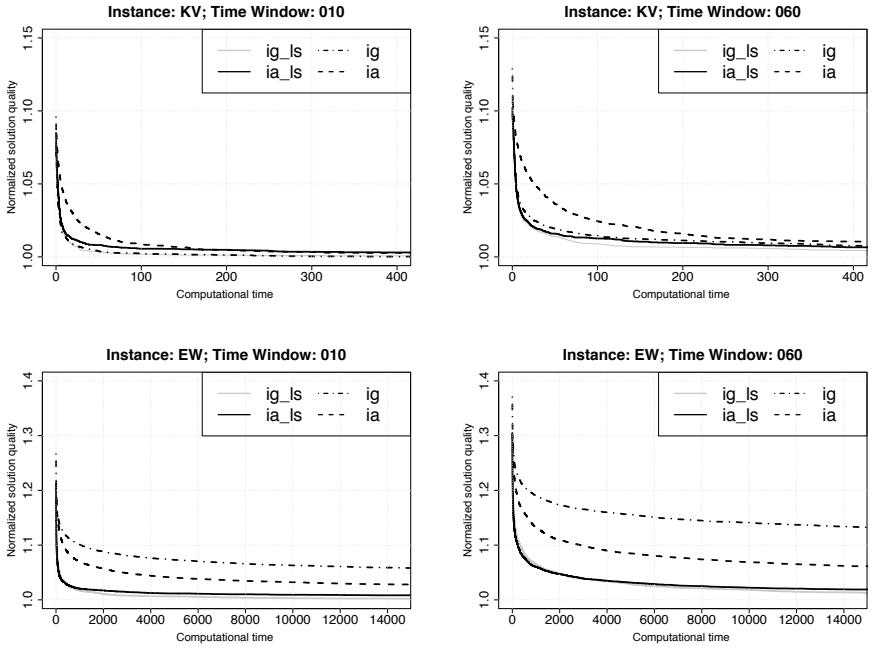


Figure A.4: Development of the solution quality over time for iterated greedy, iterated greedy plus local search, iterated ants and iterated ants plus local search for instances `KV-10` (top left), `KV-60` (top right), `EW-10` (bottom left), and `EW-60` (bottom right).

the solutions generated by the heuristics are feasible for the ILP formulation, that is, despite this difference, the heuristic solutions are valid upper bounds for the ILP. The difficulty in comparing our solutions to the ILP ones is that the ILP formulation allows possibly much better solutions to be reached. If we anyway compare the solutions, then we can see that for the same instance–time window combination, the `IG-LS-CVSPTW` algorithm typically finds solutions which use the same number of locomotives or one or two more than in available optimal solutions. Additionally, our heuristics have the advantage that they are applicable to very large instances, which are beyond the reach of the ILP-based approach. In fact, already for the largest instance for which the ILP solver could still deliver solutions in [85] (instance `EW` with time windows ±30), improved upper bounds could be found by `IG-LS-CVSPTW`.

Our heuristic algorithms can generate good quality in relatively short computation times on the instances tested here, say a few minutes for the largest instance tested here. Given more computation time, `IG-CVSPTW` and `IG-LS-CVSPTW` can use this time effectively and further

improve the quality of the solutions over time. On the largest instance tested here, computation times up to four hours have been considered to test the limiting behavior of the heuristics (see, for example, the results given in Figure A.4 on instance `EW`). The computational effort is, however, still lower than the one given to the ILP approach, which was run for one CPU hour in parallel on eight cores of a similar machine as ours (CPLEX 10 makes effective use of multiple cores by parallelizing the branch-and-bound tree computations.). In addition, there are still a number of possibilities for improving the speed of the heuristic algorithms, ranging from the adoption of pre-processing techniques to reduce instance size (which is actually done for the ILP model) to more fine-tuned implementations.

Concerning the results for the iterated ants, one may wonder, whether the setting of $\alpha = 0$ is an artifact of the tuning and whether on larger instances the usage of pheromone trails would result into better performance. (Note that the `KV` and `EW` instances are in part much larger than the ones used for tuning.) To test this, we took the second ranked algorithm configuration from the race, which had for both cases–with and without local search–settings of α around one. However, on the two largest instances no significant improvements by using pheromone trails in the solution construction have been identified.

The negative results of the tentative extension of IG to iterated ants allows two different interpretations. On one side, it does not exclude that an iterated ants approach or another population-based extension may further improve performance. For example, different ways of defining the meaning of the pheromone trails may be tested, such as associating locomotive types to trips. However, significant further developments and tests would have to be done, time which could be also used to further improve the simpler algorithm, for example, by using more elaborate construction heuristics or improved local search algorithms. On the other side, these tests are also a confirmation that conceptually rather simple algorithms are a good means to improve the performance of basic heuristics especially in the context of real-world problems. In this sense, the results here give also an example of how a bottom-up development of SLS algorithms, which adds algorithm features in a step-by-step manner, is a viable approach to obtain high performing yet conceptually simple algorithms.

A.8 Conclusions

In this chapter we have developed a high performing stochastic local search algorithm for a freight train scheduling problem arising in

the strategical planning of Deutsche Bahn AG. In particular, we have shown that a combination of an iterated greedy heuristic with a simple local search algorithm yields very promising performance. With this algorithm we now can obtain high quality solutions to large problem instances, for which an approach based on a commercial solver ILP solver is not effective anymore.

There are a number of directions in which this research could be extended. A first is certainly the application and comparison of the iterated greedy and iterated ants algorithms on very large instances. For tackling large instances, computation time reductions may be obtained by adopting a pre-processing phase to reduce the effective instance size tackled (this was also indicated by initial tests). Another attractive possibility is to consider hybrids between the exact solution methods and the iterated greedy algorithms. One way to do this is by exploiting the very good performance of the commercial solver for small sized instances: One may use the iterated greedy algorithm to define partial solutions and use the ILP solver to compute their optimal extensions to complete ones.

Appendix B

ScaLa: Scaling large instances

Solving large instances of optimization problems to a satisfactory level usually requires long computation time, sometimes hours or even days. Tuning target algorithms for such instances usually requires a large number of evaluations on similar training instances. This quickly makes automatic tuning computationally prohibitive. We referred to these instances that requires a long computation time to solve as *large* instances throughout this chapter, since typically, instances with larger size require longer computation time to solve. However, the *largeness* referred here may also depend on other instance features and whether the target algorithm is effective in tackling them. These features may include, e.g. problem-independent features such as ruggedness of the fitness landscape, fitness distance correlation, and so on; and problem-specific instance features such as dominance and sparsity of the instance matrix in QAP, etc. More detailed survey on measuring instance difficulty can be found in [205]. Our preliminary experiments here consider only instance size as a measure of instance "largeness", but incorporating other features is straightforward.

To make automatic tuning applicable for large instances, one idea is to tune on smaller instances [35, 219]. Styles et al. proposed in [219] to run multiple tuning processes on small instances, validate the independently tuned configurations on medium instances, and use the best validated configuration for solving the large instances. Our current work takes a different direction. The goal is to tune on small instances with short runtime, such that the tuned configuration performs similarly on large instances with long runtime. In order to realize this goal, a number of questions have to be addressed:

1. how to measure similarity among different instances?

Table B.1: Parameters for SA-TS on QAP

Parameter	Description	Type	Range	Default
Temp	Initial temperature of SA algorithm	Continuous	[100, 5000]	5000
Alpha	Cooling rate	Continuous	[0.1, 0.9]	0.1
Length	Length of tabu list	Integer	[1, 10]	10
Limit	maximum number of non-improving iterations prior to intensification strategy (re-start the search from best-found solution)	Integer	[1, 10]	10

2. Does there exist similarity between small instances and large instances at all?

3. Do good configurations on small instances perform well on similar large instances?

After a brief description of our experimental setup, each of the three questions above will be addressed in one of the following sections.

B.1 Experimental setup

We refer the algorithm whose performance is being optimized as *target algorithm* and the one that is used to tune it as the *tuner*. As target algorithm, we use a hybrid Simulated Annealing and Tabu Search (SA-TS) algorithm [177]. It uses the Greedy Randomized Adaptive Search Procedure (GRASP) [74] to obtain an initial solution, and then uses a combined Simulated Annealing (SA) [131] and Tabu Search (TS) [93] algorithm to improve the solution. There are four numerical parameters, real-valued or integer-valued, to be tuned as described in Table B.1.

The QAP instances are generated by the generator described in [133]. Both the flow matrix and the distance matrix are integers generated from a uniform distribution. 120 instances are generated for training with sizes 50, 70, 90, and 150, and another 100 testing instances are generated with size 150.

All experiments were performed on a 1.7GHz Pentium-4 machine running Windows XP for SufTra and on a 3.30GHz Intel Core i5-3550 running Windows 7 for ScaLa. We measured runtime as the CPU time needed by these machine.

B.2 Measuring instance similarity

To answer question 1, there exist two different approaches to finding similarities among instances. One is based on instance features, e.g. instance size, fitness distance correlation, search trajectory patterns etc. Both instance-specific tuners ISAC [125] and CluPaTra [145] cluster instances based on features, then tune on each cluster, and confirm that tuning on separate clustered instance set leads to better performance than tuning on all instances. Unlike our approach in this work, they did not take into account the computation time as instance feature. Another approach is based on empirical algorithm performance [201]. Schneider *et al.* introduced in [201] two measures, a *ratio measure* and a *variance measure*, for measuring instance similarity based on relative performance of different algorithms (or same algorithm with different configurations). However, the performance-based similarity measure depends on two folds: computation time and solution quality. Although [201] considered computation time, but did not consider scaling among different instances by, e.g. considering different solution quality threshold. In this work, we adopt the performance-based similarity measures proposed in [201], more specifically, the *variance measure* that is described in more details in the next section, and use them to find similarities among different instances with different runtime.

B.3 Finding similarities between large and small instances

To answer question 2, we set up experiments to test the hypothesis whether large instances could be similar to small instances at all given different computation time. Here, the goal is to join instances with different features, given different runtime. We take QAP as our target problem, and an implementation of SA-TS algorithm as our target algorithm (see Section B.1 for a description and parameter ranges). 30 instances are generated for each of the four instance sizes 50, 70, 90, and 150. The SA-TS has four parameters as described in Table B.1. In order to use the performance-based measure, 100 parameter configurations are sampled uniformly within the parameter range. Each parameter configuration runs once on each instance. The solution cost $c_\theta(n, t_n)$ of a configuration $\theta \in \Theta$ on an instance size $n \in N = \{50, 70, 90, 150\}$ with a given runtime t_n is computed by taking the mean solution cost across the 30 instances with size n, and $C_\Theta(n, t_n) = \{c_\theta(n, t_n), \theta \in \Theta\}$. For each instance size n, a set of runtime T_n is determined as follows:

let minimum runtime $t_{min} = 0$, maximum runtime t_{max} takes value of the maximum natural stopping time of the algorithm (no restart), and T_n takes values in a logarithmically spaced sequence between t_{min} and t_{max}, excluding t_{min}. Following [201], we perform a standardized z-score normalization for each cost vector $C_\Theta(n, t_n)$, and use the variance measure

$$Q_{var}(\Theta, N', T_{N'}) = \frac{1}{|\Theta|} \sum_{\theta \in \Theta} Var(c_\theta(N', T_{N'})), \text{ for } N' \subseteq N \qquad \text{(B.1)}$$

for measuring similarity (more precisely, dissimilarity) among different pairs of $(n, t_n) \in (N, T_N)$. Based on this measure Q_{var}, instances of different size and computation time can be clustered with the goal of optimizing similarity of the resulting subsets. A classical clustering approach Hierarchical Agglomerative Clustering or AGNES [126] is adopted in our preliminary experiments (an alternative clustering method K-mean also gives very similar clustering result). For illustrative purpose, 5 logarithmic time intervals for each instance size n are used, excluding t_{min}, this makes $|T_n| = 4$. The clustering results are shown in Figure B.1. Interestingly, the most similar two subgroups turn out to be the longest runtime (natural stopping time) of each of the four instance sizes $n \in N$, and the second longest logarithmic runtime level of each $n \in N$. More specifically, the four (n, t_n)-pairs $(50, 23.6), (70, 29.8), (90, 39.2), (150, 127.6)$ form the most similar group, while $(50, 6.7), (70, 8.0), (90, 9.8), (150, 23.8)$ comprise the second most similar group. In the two shorter levels of runtime, the similarities across the four instance sizes are less obvious. Nevertheless, this interesting clustering result confirms our hypothesis raised in question 2: using performance-based similarity measure, given the right runtime, different instances, small or large, can become similar to each other.

B.4 Solving large instances by tuning on small instances

How can automatic tuning benefit from this automatically detected instance similarity? One straightforward follow-up idea is to use the best parameter configuration tuned on small instances with short runtime to solve similar large instances with long runtime. However, it remains unjustified that how good these tuned-on-small parameter configurations are, compared with, for example, parameter configuration tuned directly on instances of the same size with the same runtime. In this

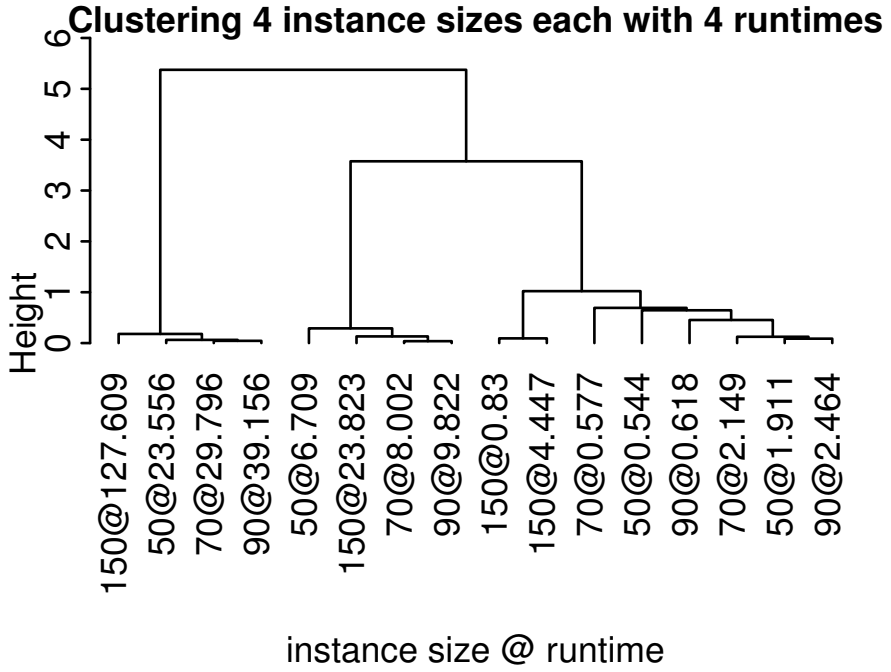


Figure B.1: Clustering four different instance sizes each with four different computation times by hierarchical agglomerative based on variance measure.

experiment, two most similar groups of size-runtime pairs (see Figure B.1 of Section B.3) are used: the first group includes (50, 6.709), (70, 8.002), (90, 9.822), and (150, 24.823); the second group includes (50, 23.556), (70, 29.796), (90, 39.156), (150, 127.609). For each of the two groups, two sets of experiments are set up: 1) tuned by `oracle`: as a quick proof-of-concept, we take the best configuration from 100 configurations based on 30 instances on each instance size as found in Section B.3, and test them on another 100 testing instances of size 150 with corresponding runtime; 2) tuned by ParamILS [118]: we tune the target algorithm using three independent runs of ParamILS for each instance size using the size-runtime pairs mentioned above, each run was assigned maximum 300 calls of target algorithm on new randomly generated training instances, and each tuned configuration is tested on 100 same testing instances as in 1). The second experiment set is to test generalizability of the similarity information detected in Section B.3. The goal is to see how good these best configurations tuned on small instances such as 50, 70, and 90 with shorter runtime, compared with the best configuration tuned on instance size 150, when tested on instance size 150 with the same runtime.

The results are listed in Table B.2. The results confirm that, firstly

as expected, a large amount of tuning time is saved by tuning on small instances, ranging from 59 to 81% in our experiments; and secondly, in general, parameter configurations tuned on smaller instances with shorter runtime do not differ significantly from the ones directly tuned on large instances, as long as similarities between them are confirmed. In both groups of both experiment sets, there is no statistical difference between the configurations tuned on 50, 70, 90, and 150, tested by Wilcoxon's rank-sum test. In the first experiment set tuned by `oracle`, configurations tuned on size 70 and 90 sometimes perform even better than tuned on 150. The mean performance difference from the tuned-on-150 configuration in the first group is usually less than 0.1%, and even less than 0.01% in the second group. In the second experiment set tuned by ParamILS, although configuration tuned on size 150 performs best, the difference is not significant: the mean performance difference is usually less than 0.1% in the first group, and less than 0.05% in the second group. This shows the similarity information detected from Section B.3 can be actually generalized to tuners with different training instances. As reference, the performance of the default parameter configuration (listed in Table B.1) is presented in Table B.2, and it is statistically significantly outperformed by almost all the above tuned configurations in both groups, which proves the necessity and success of tuning process. We also include as reference the best configuration tuned on instance size 150 with runtime 23.556 (127.609) seconds to be tested on instance size 150 with different runtime, i.e. 127.609 (23.556) seconds, respectively (in row 150' of Table B.2). Although the tuning and testing instances are of the same size, different runtime makes a great performance difference, resulting in almost one order of magnitude worse than tuning on the small instances with appropriate runtime. The 150' performance is statistically significantly worse than all the above tuned configurations belonging to the same group, and it is even significantly worse than the default configuration in the second group. This contrasts with the fact that the difference among the similar size-runtime pairs (the first four rows of Table B.2) is indeed very minor, and it also shows the risk of tuning on algorithm solution quality without assigning the right runtime, which in fact proves the necessity of our automatic similarity detection procedure in ScaLa.

B.5 Summary

In this work, we constructed a proof-of-concept approach ScaLa for tuning large instances, and tested it on the Quadratic Assignment Problem (QAP). In the development of ScaLa, we used performance-based sim-

Table B.2: Results for the performance of the best parameter configurations tuned on sizes 50, 70, 90, 150, and tested on instances of size 150. Two most similar groups of size-runtime pairs (see text or Figure B.1) are used. Two experiment sets are presented, `oracle` and ParamILS (see text). Each column of `%oracle` and `%ParamILS` shows the mean percentage deviation from the reference cost. In each column, $+x$ ($-x$) means that the tuned configuration performance is $x\%$ more (less) than the reference cost. `%time.saved` shows the percentage of tuning time saved comparing with tuning on instances of size 150. The performance of default parameter configuration is shown in row "def.". The last row 150' used the best parameter configuration tuned on instance size 150 with runtime 127.609 (23.556) seconds, and tested on instance size 150 with runtime 23.556 (127.609) seconds, respectively. Results marked with † refers to statistically significantly worse results compared to tuned-on-150 using Wilcoxon's rank-sum test.

	23.556 seconds			127.609 seconds		
tuned.on	%oracle	%ParamILS	%time.saved	%oracle	%ParamILS	%time.saved
50	−0.48	−0.047	72	−0.048	−0.048	81
70	−0.65	−0.053	66	−0.060	−0.027	76
90	−0.61	−0.093	59	−0.057	−0.040	69
150	−0.58	−0.151	0	−0.060	−0.070	0
def.	$+1.17^{\dagger}$	$+0.150^{\dagger}$	-	$-0.008^{\dagger}$	−0.024	-
150'	$+1.16^{\dagger}$	$+0.195^{\dagger}$	-	$+0.232^{\dagger}$	$+0.208^{\dagger}$	-

ilarity measure and clustering technique to automatically detect and group similar instances with different sizes by assigning different runtime, such that one can tune on small instances with much less runtime and apply the tuned configuration to solve large instances with long runtime. This greatly reduces computation time required when tuning large instances. Through our preliminary experiments, we empirically showed that small instances and large instances can be similar when given the right runtime, and in such case, the good configurations tuned on small instances can also perform well on large instances.

ScaLa is still an actively ongoing work. Future works include largely extending the amount of experiments, consider also testing on problems other than QAP, and extend our studies to other state-of-the-art algorithms. The correlation between computation time and instance size may be algorithm-specific, therefore, an automatic approach to detecting it is practically valuable. Our current approach is still a proof-of-concept, since it is computationally expensive for computing

the performance-based measure. In future work, we plan to investigate how to reduce the computation expenses by, e.g. taking fewer instances and fewer but good configurations found during the tuning process. In particular, we plan to investigate the possibility of predicting the "right" runtime for an unseen instance such that it is similar to a known group of instances.

Appendix C

Offline Configuration Meets Online Adaptation: An Experimental Investigation in Operator Selection

C.1 Introduction

The performance of metaheuristic algorithms for hard optimization problems usually depends on their parameter settings [91]. How these algorithm parameters can be properly set has been a challenge posed in front of every designer as well as user of the algorithm. In recent years, many works on using automatic algorithm configuration to replace the conventional rule-of-thumb or trial-and-error approaches have been proposed, and automating the process of finding good algorithm parameter settings has attracted more and more research attentions [97, 107, 22].

The automatic algorithm configuration methods can generally be categorized into two classes: offline method and online method. The offline configuration method, also referred to as parameter tuning, consists in finding a good parameter configuration for the target algorithm in a training phase *before* the algorithm run [33]. It learns and selects the best parameter configuration based on a set of available training instances that resembles the future unseen instances. These training instances in practice can be obtained from, e.g., a simulated instance generator or historical data if the target optimization problem happens in a recurring manner, for example, to optimize logistic plans on weekly delivery demand, to arrange university course timetable on a semester basis, or to schedule school busses on a yearly basis, etc. There exist a

number of established approaches for the task of offline configuration, for example, statistical racing techniques [36] and iterated racing [37], an iterated local search approach (ParamILS) [118], genetic algorithms [5, 176], model-based approach SMAC [111], and continuous optimizers with post-selection techniques [233, 242] specifically for numerical parameters. They regard the target algorithm to be configured as a black box, and can be applied to any algorithms without human intervention. Once the training phase is finished, the target algorithm is deployed using the best parameter configuration found through the training instances, and usually fixes it when solving each instance, and across different instances encountered.

On the other hand, instead of keeping a static parameter configuration during the algorithm run, an online configuration method varies the parameter setting *during* the algorithm run. Such approaches are also referred to as parameter adaptation [4] or parameter control [73]. There are two potential advantages of varying parameter settings at run time. Firstly, a good-performing fixed parameter setting across a class of problem may not perform well on each particular instance, especially when the instances are heterogeneous. Secondly, different search stages may prefer different parameter settings, for example, a rule of thumb is to use more explorative settings at the beginning of the search and then switch to more exploitive settings at the later stage to converge the search. The online parameter adaptation problem has attracted many attentions and research efforts, especially in the field of evolutionary algorithm [148]. The usage of machine learning techniques in parameter adaptation is also the unifying research theme of reactive search [22].

Although the online and offline methods approach the automatic algorithm configuration problem differently, they can be regarded as complementary to each other, and thus can be incorporated together. For example, the online methods usually also have a number of hyperparameter to be configured, and this can be fine-tuned by an offline method. [78] discretized the continuous parameter space of different adaptive operator selection methods, and adopted a statistical racing technique to select the best setting. Besides, offline methods can provide a good starting parameter configuration, which is then further adapted by an online mechanism once the instances to be solved are given. [185] provide an in-depth analysis in this direction in the context of ant colony optimization algorithms (ACO) [71]. They compared offline configuration and various online adaptation methods in ACO, and explored the possibility to adapt the offline tuned configuration. Their results have shown that offline configuration is more beneficial

than online adaptation, and also adapting the offline tuned configuration usually performs worse than fixing the tuned configuration. [82] compared online adaptation in operator selection with a static operator tuned offline, and found that using a tuned static operator is more preferable in their context. A preliminary study by [241] further showed that offline configuration could be applied not only to select a static operator, it could also be used to tune a static probability distribution from which the operators are dynamically generated and varied iteration by iteration.

In this work, we focus on the operator selection problem [60], since it was reported as influential to the performance of evolutionary algorithms [75]. We employ the state-of-the-art offline configuration tools to the adaptive operator selection strategies. To the best of our knowledge, the hyper-parameters of the adaptive operator selection methods are never tuned by a search-based configuration tool designed for continuous parameters (e.g. [233]). Secondly, we experimentally investigate the two potential advantages in adaptive operator selection strategies, i.e. being able to adapt to per instance setting and adapt to search stages. Thirdly, we develop a simple non-adaptive approach to vary parameter values by an offline tuned probability distribution, and use it to challenge: (1) the conventional offline configuration approach that fixes an operator during algorithm run; (2) the adaptive operator selection mechanisms.

The remainder of this article first gives an overview of further related works in Section C.2, describes the target problem and target algorithms in Section C.3 and C.4, introduces different operator selection strategies in Section C.5. The experimental setup and computational results are presented in Section C.6 and C.7, respectively, and Section C.8 concludes and provides outlooks for future work.

C.2 Related works

Most of the applications of offline configuration fixes the configuration during the algorithm run, regardless of the instances encountered. There exist also specific offline configuration approaches for heterogeneous instances. One of such approaches is called portfolio-based algorithm selection [194, 137]: e.g., SATzilla [228], builds a portfolio of algorithm trains an empirical hardness model [142, 223] that predicts runtime for a portfolio of algorithm configurations on instances with different features, and then selects a configuration from the portfolio with the best predicted performance for a given instance to be solved. A limitation in SATzilla is that it requires a priori domain knowledge

to build an uncorrelated static portfolio. A follow-up work to overcome this limitation, Hydra [229], uses offline configuration to dynamically generate complementary configurations to update the portfolio. Another approach for handling heterogeneous instances are the instance-specific algorithm configuration (ISAC) [125, 146]. The basic idea is to cluster the instances based on their features and apply offline configuration of the algorithm on each cluster of instances. Both approaches have reported promising results, especially SATzilla is reported to be the most successful solver in the SAT competition. However, both portfolio-based selection and instance-specific configuration approaches keep their configuration fixed when solving an instance.

Fixing a configuration during the entire algorithm run may not be a best choice. There are various ways of varying the configuration over time. For example, [161] has reported a uniformly random variation of parameter values over time improves the performance and robustness of an ACO algorithm with a fixed setting. [218] discuss parameter adaptation techniques for ACO algorithms and show the usefulness of a simple parameter variation scheme by a pre-scheduled function. Most of the other works focus on more sophisticated approaches to vary parameter settings over different search stages adaptively based on runtime feedback from the search.[1] Such examples include reactive search [22], adaptive operator selection [60, 75], etc. However, there exists no strong experimental evidence to support the assumption that adaptive approaches vary the parameter values at different search stages better than the non-adaptive approaches. [160] has conducted an empirical study on reactive tabu search (RTS) [23], one of the most prominent online adaptive approach. They have observed that the adaptive mechanism in RTS does not adapt well to different search stages. In their experiment, setting the tabu list length randomly with the same empirical distribution of the RTS adaptation scheme leads to very close performance as RTS. They further used offline configuration to tune a distribution for randomly setting tabu list length at each search stage in the robust tabu search [220], which significantly outperforms RTS. In adaptive operator selection, [75](pages 161–170) also conducts some preliminary experiments that randomly generates operator by an offline tuned distribution, and found rather equivalent performance comparing with his best adaptive operator selection method applied to differential evolution.

[1] A more detailed classification of online adaptation schemes can be found in [73].

C.3 Target problem

We adopt the quadratic assignment problem (QAP) [136, 199, 183, 41] as target problem for our study. The QAP is one of the most widely studied NP-hard combinatorial optimization problems. It is the abstract model of many real-life facility layout problems, including layout design of campus, hospital, typewriter keyboard layout, etc. The task is to assign n interrelated facilities to n locations, such that the most related facilities are assigned as close as possible. The distance between each pair of locations i and j is given by d_{ij}, an entry of an $n \times n$ distance matrix D; and the relatedness between each pair of facilities k and l is given by f_{kl} from a so-called flow matrix F. A solution of QAP is represented by a permutation vector π, such that $\pi_i = j$ assigns to a location i a unique facility j. The objective is to find a permutation π that minimizes the total cost defined as follows:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} \cdot f_{\pi(i)\pi(j)}. \tag{C.1}$$

That is, the total cost is a sum of unit cost for each pair of locations i and j, which is defined by multiplying their distance d_{ij} with the flow $f_{\pi(i)\pi(j)}$ of the two facilities assigned to them.

C.4 Target algorithm

The operator selection [75] in the evolutionary algorithm (EA) is chosen as our target algorithm for this empirical study of offline configuration and online adaptation. Our EA implementation is inspired by the work of [165] for QAP. Each individual in EA represents a QAP solution, encoded as a vector of the permutation π where $\pi_i = j$ means facility j is assigned to location i. A population of p individuals are evolved from iteration to iteration by applying variation operators such as crossover and mutation. Initially, the p individuals are generated uniformly at random. Then at each iteration, p_c new individuals, dubbed offspring, will be generated by applying a crossover operator, and p_m new individuals will be generated by applying a mutation operator. All these new individuals may be refined by applying an optional local search procedure. The best p individuals among the old and newly generated individuals will be selected to continue to the next iteration.

A crossover operator generates a new individual based on two existing individuals which are named parents. The two parents are selected uniformly at random from the existing individuals. In this study, we look into the following four different crossover operators:

Cycle crossover (CX) [167] first passes down all solution components that are shared by both parents, I_{p_1} and I_{p_2}, to the offspring, I_o. The remaining components of the offspring are assigned starting from a random one, $I_o(j)$. CX first sets $I_o(j) = I_{p_1}(j)$. Then, denoting $I_{p_1}(j')$ as the component where $I_{p_1}(j') = I_{p_2}(j)$, CX sets $I_o(j') = I_{p_1}(j')$ and substitutes the index j with j'. This procedure is repeated until all components of I_o are assigned.

Distance-preserving crossover (DPX) [166] generates an offspring that has the same distance from both parents. DPX simply passes down to I_o all the components that are shared by both I_{p_1} and I_{p_2}. Each of the remaining components, $I_o(j)$, is assigned randomly provided that $I_o(j)$ is a permutation and it is different from both $I_{p_1}(j)$ and $I_{p_2}(j)$ in some approximate sense.

Partially-mapped crossover (PMX) [95] randomly draws two components locations of I_o, namely j and j' where $j < j'$. PMX then sets $I_o(k) = I_{p_1}(k)$ for all k outside the range of $[j, j']$ and $I_o(k) = I_{p_2}(k)$ for all $j \leq k \leq j'$. If the offspring generated is not a valid permutation, then for each component location pair k and z where $I_o(k) = I_o(z)$ and $j \leq z \leq j'$, PMX sets $I_o(k) = I_{p_1}(z)$. This process is repeated until a valid permutation is obtained.

Order crossover (OX) [59] randomly draws two component locations of I_o, namely j and j' where $j < j'$. OX then sets $I_o(k) = I_{p_1}(k)$ for all $j \leq k \leq j'$ and assigns in the k-th unassigned component of I_o the k-th component of I_{p_2} that differs from any $I_o(z)$, $j \leq z \leq j'$.

C.5 Operator selection strategies

In this section, we group different operator selection strategies into three categories: the *static operator strategy* that selects one operator and fixes it during the algorithm run; the *mixed operator strategy* that generates an operator at each generation by a fixed probability distribution; and *adaptive operator selection strategy* that adapts the operator probability during the algorithm run. The connections between them can be illustrated in Figure C.1.

C.5.1 Static operator strategy

The static operator strategy (SOS) refers to fixing one operator when solving an instance. This can be regarded as assigning a value to a categorical parameter, with the selectable operators being its possible

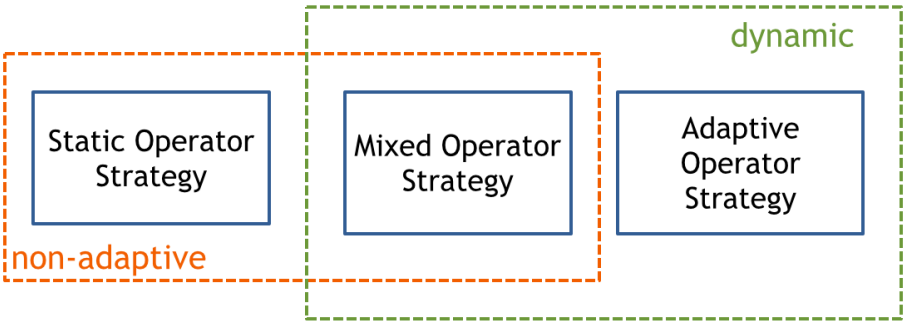


Figure C.1: Different categories of operator selection strategies.

values. This is most commonly used in the design of evolutionary algorithms. [82] showed that using a static operator tuned offline often performs better than the adaptive operator selection strategies in their context.

C.5.2 Mixed operator strategy

In contrast with fixing one operator to use, the mixed operator strategy[2] (MOS) assigns a selection probability to each operator. This allows an operator to be selected at each iteration of the algorithm under a certain probability. This strategy is usually designed with a uniform probability distribution for each possible operators, and is referred to as "naive" strategy in the literature, e.g., [76, 138, 82], as a baseline opposed to the "adaptive" operator selection strategies introduced in Section C.5.3. Of course, the selection probability of each operator can be set in other ways than uniform, and can be regarded as a real-valued parameter. One can even regard the static operator strategy as a degenerate case of a mixed strategy, in which one operator is selected with probability 1, and each of the others with probability 0. The selection probability can potentially be fine-tuned in the offline training phase. In this study, we will experimentally explore this possibility.

C.5.3 Adaptive operator selection

Different from the two approaches above, adaptive operator selection strategy (AOS) try to adjust the parameter values while running the target algorithm for each instance. As an online method, it is able to adapt parameter values according to different instances as well as different search stages.

[2]The term is syntactically and semantically analogous to the term *mixed strategy* widely used in game theory.

We further group the SOS and MOS as *non-adaptive* strategy, since their operator selection mechanisms, either the operator choice or the operator probability distribution, do not change during algorithm run. MOS and AOS can be also grouped as *dynamic* strategies, since their choice of operator can change dynamically in each generation, as opposed to using a static operator. The categorization of the operator selection strategies is illustrated in Figure C.1.

The development of an AOS needs to address two issues:

- the reward function, or credit assignment mechanism [77], which concerns the measurement of operator quality according to operator performance;
- and the adaptation mechanism, which concerns which operator to use at each time step according to the performance measurement.

In this work, we used two different reward functions described in Section C.5.3, and three different online adaptation methods described in Section C.5.3.

Reward function

Two reward functions were used in our work. Both versions make reference of the cost of the offspring c_o to the cost of the current best solution c_b and the better parent solution c_p. The first reward function is adopted from the study of [82] as follows. For an operator i that is used to generate a set $\mathcal{I}_i$ of offspring at the current iteration, its reward R_i^1 is defined as:

$$R_i^1 = \frac{1}{|\mathcal{I}_i|} \cdot \sum_{o \in \mathcal{I}_i} \frac{c_b}{c_o} \max\{0, \frac{c_p - c_o}{c_p}\}. \tag{C.2}$$

Note that the reward is only given if the offspring is better than both of its parents. A drawback in the reward function R^1 is that the relative improvement of the offspring over its better parent will bias the multiplicative reward value much stronger than its relative performance to the current best solution. This may not be effective especially when the parents are drawn uniformly randomly. We modify (C.2) by making the reference to the parent solution and the current best solution to contribute the same magnitude to the reward function as follows:

$$R_i^2 = \frac{1}{|\mathcal{I}_i|} \cdot \sum_{o \in \mathcal{I}_i} \frac{c_b}{c_o} \cdot \frac{c_p}{c_o} \cdot sign(c_p - c_o), \tag{C.3}$$

where $sign(x)$ function returns 1 when $x > 0$, and returns 0 otherwise, so that only the crossover operators that result in improved offspring over its both parents are given credits.

Online adaptation mechanisms

We considered the three online algorithm adaptation methods studied by [82] for the operator selection problem. These adaption methods include: Probability Matching (PM) [55], Adaptive Pursuit (AP) [221], and Multi-Armed Bandit (MAB) [11].

Probability Matching. It selects operators stochastically based on their quality. The quality Q_i of each candidate operator i can be interpreted as an exponential, recency-weighted average of rewards. It is updated at each iteration by a fraction of the difference between the current rewards of the iteration and the cumulative rewards:

$$Q_i = Q_i + \alpha \cdot (R_i - Q_i) \tag{C.4}$$

where $0 \leq \alpha \leq 1$ is a parameter called *adaptation rate* that controls how fast the adaptation takes place. A high value of α makes the adaptation react quickly to recent reward changes, at the risk of being misled by evaluation noise; while a low value of α adapts more slowly. Q_i is by default initialized to 1 for each operator i. Then the probability of choosing an operator i is given by the following formula:

$$P_i = P_{min} + (1 - |I|P_{min})\frac{Q_i}{\sum_{i' \in I} Q_{i'}}, \tag{C.5}$$

where I is the set of all possible operators. The minimum probability $0 < P_{min} < \frac{1}{|I|}$ is a parameter to guarantee that every operator has a chance to be selected. Note that having a non-zero P_{min} is important in a non-stationary environment, which means the operator rewards change over time. For example, an undesirable operator at the beginning of the algorithm run may become the most effective one at a later stage of the algorithm. But once a probability P_i drops to 0, this operator i will no longer be selected. The selection probability increases as the relative quality increases.

Adaptive Pursuit. The second adaptation method AP is also a stochastic operator selection method based on the quality of each operator. The quality measure Q_i of an operator i is updated in the same way as in PM by (C.4). And AP differs from PM by a different

probability update formula than (C.5):

$$P_i = \begin{cases} P_i + \beta(P_{max} - P_i), & \text{if } Q_i = \max_{i'} Q_{i'} \\ P_i + \beta(P_{min} - P_i), & \text{otherwise,} \end{cases} \tag{C.6}$$

where

$$P_{max} = 1 - (|I| - 1)P_{min},$$

and the *learning rate* $0 \leq \beta \leq 1$ is a parameter that controls how fast the selection probability converges. Over time, the probability of choosing a promising operator converge to P_{max} while all others descend to P_{min}.

Multi-Arm Bandit. The third adaptation method MAB selects an operator $\bar{i}$ deterministically by

$$\bar{i} = \arg\max_{i \in I} \left\{ \bar{R}^i + \gamma(\sqrt{\frac{2 \ln \sum_{i'} n_{i'}}{n_i}}) \right\}, \tag{C.7}$$

where $\bar{R}^i$ is the average reward computed since the beginning of the search and n_i is the number of times the crossover operator i is chosen.

C.6 Experimental setup

All experiments were conducted on a computing node with 12-core Intel Xeon X5675 CPU at 3.07 GHz sharing 48 GB RAM. Each run used a single thread.

C.6.1 Instance setup

Three classes of QAP instances are considered in our experiments: one heterogeneous and two homogeneous sets of different difficulty. For the heterogeneous set (`het`), we followed the experimental setup in [82], and selected 32 instances from QAPLIB [42] with size from 50 to 100.[3] They are heterogeneous not only in different sizes, but they also stem from different background with different structures. For the homogeneous sets, we generated 32 relatively easy homogeneous instances (`hom-easy`) and 32 harder homogeneous instances (`hom-hard`) using instance generator described in [216]. The instances in `hom-easy` are unisize 80, with Manhattan distance matrix and random (unstructured)

[3]There are in total 33 instances found in the QAPLIB with size from 50 to 100. We further exclude one of them, esc64a, which is too simple and each algorithm considered in this work will solve it to optimum. Then it results in a total number of 32 instances in the heterogeneous set.

flow matrix generated with the same distribution with 50% sparsity. The `hom-hard` instances are uni-size 100 with zero sparsity. Both homogeneous instance sets are chosen with large size (80 and 100), so that the computational overhead of the online adaptation mechanisms can be ignored.[4] All three instance classes are divided in half, 16 instances for training and 16 others for testing. Each instance was run 30 times, resulting in 480 instance runs. Each of the 480 runs is assigned with a unique random seed. Note that during each run, different algorithms will use the same random seed. This setting of common random seed is for the purpose of reducing evaluation variance [162].

C.6.2 Target algorithm setup

In the work by [82], three memetic algorithm (MA) schemes were used for experiments: a simple MA with crossover only; an intermediate MA with crossover and local search; and a full MA with crossover, mutation, and local search. We followed their default parameter values in our implemented MA, setting population size $p = 40$, crossover population $p_c = p/2 = 20$. A restart is triggered when the average distance over all pairs of individuals in the population has dropped below 10 or the average fitness of the population has remained unchanged for the past 30 generations. In such case, each individual except the current best one will be changed by a mutation operator until it is 30% of the instance size differ from itself. Five levels of computation time are considered in our experiments, 1, 3.1, 10, 31, and 100 seconds. From our initial experiments, we found that local search is time-consuming for the instance size we considered. For an instance of size 100, one local search took about 1 second. The intermediate and full MA thus performed no crossover within 40 seconds, and less than 5 crossover generations in 100 seconds.[5] On the other hand, incorporating local search (and mutation for the same reason) introduces "noise" into the evaluation of crossover operations, making it more difficult to distinguish the contribution due to the usage of the crossover operators. Therefore, usually many more generations are required to average out the evaluation noise due to the influence of other operators. With these observations, we excluded the local search as well as mutation[6], and focused on the crossover operation in this study. In such case,

[4]Comparing with the non-adaptive operator strategy (fixed or mixed strategy), the computational overhead of the online adaptation mechanisms in our implementation is around 1% on instances of size 100, and around 3% on instances of size 50.

[5]More sophisticated techniques such as neighborhood candidate list or *don't look bit* [158, 27] may speed up local search. However, the development of these techniques is out of the scope of this study.

[6]However, mutation will be used in restart when the population converges.

the computation time chosen corresponds to around 900, 3 000, 9 000, 30 000, 90 000 crossover generations (each generation with 20 crossover operations), respectively.

C.6.3 Offline configuration setup

An offline configuration procedure requires the following to be given:

- a set of training instances that are assumed to resemble the instances to be solved;
- a target algorithm with parameters to be tuned;
- and an offline configuration tool.

Then it embarks the offline configuration tool to find the best parameter setting of the target algorithm based on their performance on the training instances. Our experimental separation of training and testing instances are described in Section C.6.1. This section details how different operator selection strategies were configured, their parameter value ranges, and our implemented offline configuration tool.

Configuring SOS

The task is to choose one of the four crossover operators, namely, CX, DPX, PMX, and OX, based on the training set. Since the parameter space is small, we assess each static operator by an exhaustive evaluation in the training set, which comprises 480 evaluations of the 16 instances.

Configuring MOS

Three versions of MOS are presented in this work: an untrained MOS with uniform probability distribution for each operator, denoted MOS-u and two automatically tuned versions of MOS, denoted MOS-b and MOS-w. The two tuned versions differ in how the configuration experiment is designed, more specifically, in which reference operator to choose: MOS-b chooses the best static operator as reference, while MOS-w chooses the worst. Note that finding the best or the worst operator requires a priori knowledge such as studied in Section C.7.2, or additional tuning effort. However, this additional tuning effort is usually small, since the parameter space (e.g., four operators to be chosen) is much smaller comparing with the rest of the tuning task (with a configuration budget of 4000 evaluations). Suppose there are n operators, each of which is assigned a parameter $q_i, i = 1, \ldots, n$, indicating its relative quality in the operator portfolio. After a reference

Table C.1: The hyper-parameters of the online adaptive operator selection: their default values and their ranges for offline configuration.

param. name	used in	default	range	comment
α	PM, AP	0.3	[0.0, 1.0]	adaptation rate
P_{min}	PM, AP	0.05	[0.0, 0.2]	minimum probability
β	AP	0.3	[0.0, 1.0]	learning rate
γ	MAB	1.0	[0.0, 5.0]	scaling factor

operator r is chosen, in our case, either the best or worst operator, we fix $q_r = 1$, and try to tune the $n-1$ parameters $q_i, i \in \{1, \ldots, n\} \setminus \{r\}$. The range of these $n-1$ parameters is set to $[0, 1.2]$ in MOS-b, while in MOS-w, the range is set to $[0, 100]$. Note that it may not be optimal to apply the best performing static operator most frequently in the operator portfolio, similarly, the worst static operator may not have to be the least frequently used operator. After the tuned configuration q_i of each operator i is obtained, the probability p_i of operator i is set proportionally to q_i, i.e., $p_i = \frac{q_i}{\sum_{j=1}^{n} q_j}$.

Configuring AOS

As introduced in Section C.5.3, the three AOS adaptation mechanisms, PM, AP, and MAB, also have parameters that can be fine-tuned offline. These AOS parameters with their default parameter values and ranges for offline configuration are listed in Table C.1.

Racing-based offline configuration tool

Our configuration tasks described in Section C.6.3 and C.6.3 consists of real-valued parameters. In such case, exhaustive evaluation is not feasible, thus we used two state-of-the-art automatic configuration tools, namely iterated racing [37] and BOBYQA post-selection [233, 242]. We reimplemented both configuration methods in Java, making it modular and platform independent, and integrated them into the framework of AutoParTune [146]. For each of the two configuration methods, a maximum of 4000 target algorithm runs were allowed as configuration budget. Then the best configurations found by the both configurators are compared based on their training performance (median rank in the training instance-runs), and the one with the better training performance is selected. Note that the BOBYQA optimizer is only ap-

plicable for tuning more than one parameters [189], thus it cannot be applied to tuning MAB, then only iterated racing is applied.

C.7 Experimental results

This empirical study starts by first explaining the result presentation in Section C.7.1, and examines the performance of each static crossover operator in Section C.7.2; then the performance of online adaptive operator selection methods with two reward functions and with or without offline configuration in Section C.7.3; followed by an analysis of the effectiveness of the adaptive operator selection methods in Section C.7.4. The performance of using mixed crossover operators is presented in Section C.7.5, and a further hybrid of mixed operator strategy and adaptive operator selection is investigated in Section C.7.6.

C.7.1 Result presentation

The empirical results for the three instance classes, namely, the heterogeneous class `het`, homogeneous easy class `hom-easy`, and homogeneous hard class `hom-hard` are illustrated in Figures C.2, C.3, and C.4, respectively. There are six plots in each figure, presenting results in one of the five computation time levels (1, 3.1, 10, 31, and 100 seconds), as well as the overall comparison aggregating all the five computation time levels. We name an instance class with one computation time level a *case study*, and a total of 15 case studies are presented in this work (three instance classes by five computation time levels). Each case study comprises 16 instances with 30 runs each, and we further name each instance with a computation time level a *scenario*, and a total of 240 scenarios are presented here. In each plot of in Figures C.2, C.3, and C.4, the following candidate algorithms are included for comparison:

- four static operator strategies CX, DPX, OX, and PMX;
- four AP variants: AP-1 and AP-t1, being the untrained and tuned AP with the first reward function R^1 as (C.2), as well as AP-2 and AP-t2, the untrained and tuned AP with the second reward function R^2 as (C.3);
- four MAB variants, similarly, MAB-1, MAB-t1, MAB-2, and MAB-t2, differing in untrained or tuned, as well as the reward functions used;
- four PM variants: PM-1, PM-t1, PM-2, and PM-t2;

- two MOS methods PM-2r and PM-2f based on the empirical operator distribution of PM-2;
- three MOS methods: untrained MOS-u with uniform operator probability, and two tuned MOS methods MOS-b and MOS-w;
- MOS-PM, a hybrid of MOS and PM.

Each type of the algorithms listed above are grouped into one distinct block in each plot of Figures f:result-het, C.3, and C.4. For statistically comparing different candidate algorithms in a case study across 16 different instances, we adopted a non-parametric test, namely, Friedman's two-way analysis of variance by ranks [83] with a α level of 0.05. Each plot in Figures C.2, C.3, and C.4 shows the median and the 95% simultaneous confidence intervals of candidate algorithm regarding the Friedman's test comparisons. If the intervals of two candidate algorithms overlap, then the difference between them is not statistically significant.[7]

In addition, we also compared candidate algorithms on each scenario with 30 independent trials in Tables C.2, C.3, C.4, and C.5. In these tables, each entry lists three numbers, one for each of the three instance classes, `het`/`hom-easy`/`hom-hard`, respectively. There are in total 80 scenarios in each instance class, including 16 instances by 5 computation time levels. A number in each entry indicates in how many out of the 80 scenarios, the method named on the left of the row is statistically significantly better than the method named on the top of the column. The statistical significance is tested using the pairwise Wilcoxon's paired signed rank test [226] with Holm's adjustment for multiple test correction at an α level of 0.05.

C.7.2 Static operator strategy

We first applied each of the four crossover operators, CX, DPX, OX, and PMX exhaustively on the training set. In the training set, each instance class contains 16 instances with five stopping times, totalling 80 scenarios. For each scenario, we ranked the four operators on each of the 30 runs and compared their median rank. In the `het` set, PMX is best performing in 64 scenarios, followed by CX in 11 scenarios and OX in 5 scenarios (for the five stopping time levels of instance `tai64c`).

[7]We further generated the box-plot of the median ranks across 30 trials of each instance, and the performance comparison in this median-rank box-plots and the presented confidence-interval plots are almost identical. The confidence-interval plot is shown here instead of median-rank box-plot since it displays additional information of statistical significance by the Friedman test.

We further applied Wilcoxon's signed rank test for the pairwise difference among the four operators, and the statistical significance results ($\alpha = 0.05$) are listed on the left in Table C.2. PMX is the significantly best (with significant level at 0.05) in 62 scenarios, OX performs significantly best in 5 scenarios (for instance `tai64c`), while none of CX's best performing scenarios proves to be significant. In the `hom-easy` set, PMX stands out in 75 out of 80 scenarios, where 65 of them are statistically significant; and CX excels in the other five scenarios with none of them being statistically significant. PMX is most dominant in the `hom-hard` set, topping all 80 scenarios, where 78 of them are statistically significant. This clearly shows PMX's dominance in all the training set, hence is selected offline as the best static crossover operator. CX performs significantly better in 235 out of 240 scenarios than OX, which in turn performs significantly better in all the 240 scenarios than DPX.

We further applied all the four static operators to the testing set. The testing set comprises three instance classes and five stopping time, totalling 15 case studies, or 240 scenarios. The relative performance by ranks in each of the 15 case studies is shown in the first block of each plot in Figures C.2, C.3, and C.4. As clearly shown, PMX is dominantly best performing compared to the other three operators, and the difference is statistical significant in 14 case studies except only the 100-second case of the `hom-easy` set. The pairwise statistical comparisons in the 240 scenarios are also presented on the right of Table C.2, where PMX statistically outperforms CX in 205 out of 240 scenarios; CX significantly outperforms OX in all the 240 scenarios; and OX in turn significantly outperforms DPX in all the 240 scenarios. PMX is also chosen to be the reference in each plot of Figure C.2, C.3, and C.4 (vertical dotted line), since it is found to be preferable in [82].

C.7.3 Online adaptive operator selection

The ranking performance of the three online adaptive operator selection strategies, AP, MAB, and PM, are presented at the second, third, and fourth block of each plot in Figures C.2, C.3, and C.4, respectively. Within each block, there are four boxes for comparison: the first two boxes refer to untrained and tuned version with the first reward function R^1 in Equation (C.2), while the latter two boxes with the second reward function R^2 in Equation (C.3).

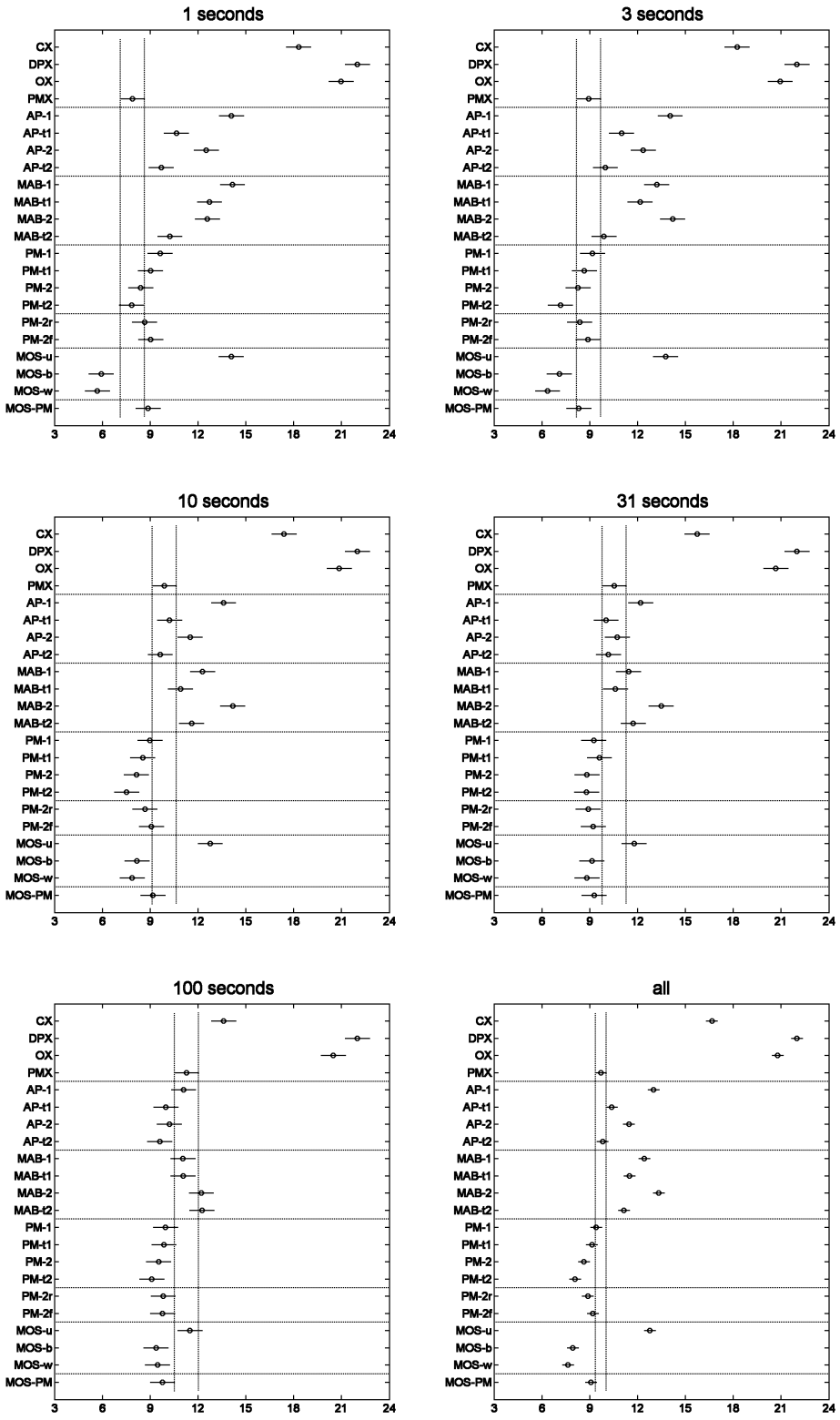


Figure C.2: The ranking performance of different operator selection methods (acronyms see text) with different computation time levels 1, 3.1, 10, 31, and 100 seconds on heterogeneous instance set.

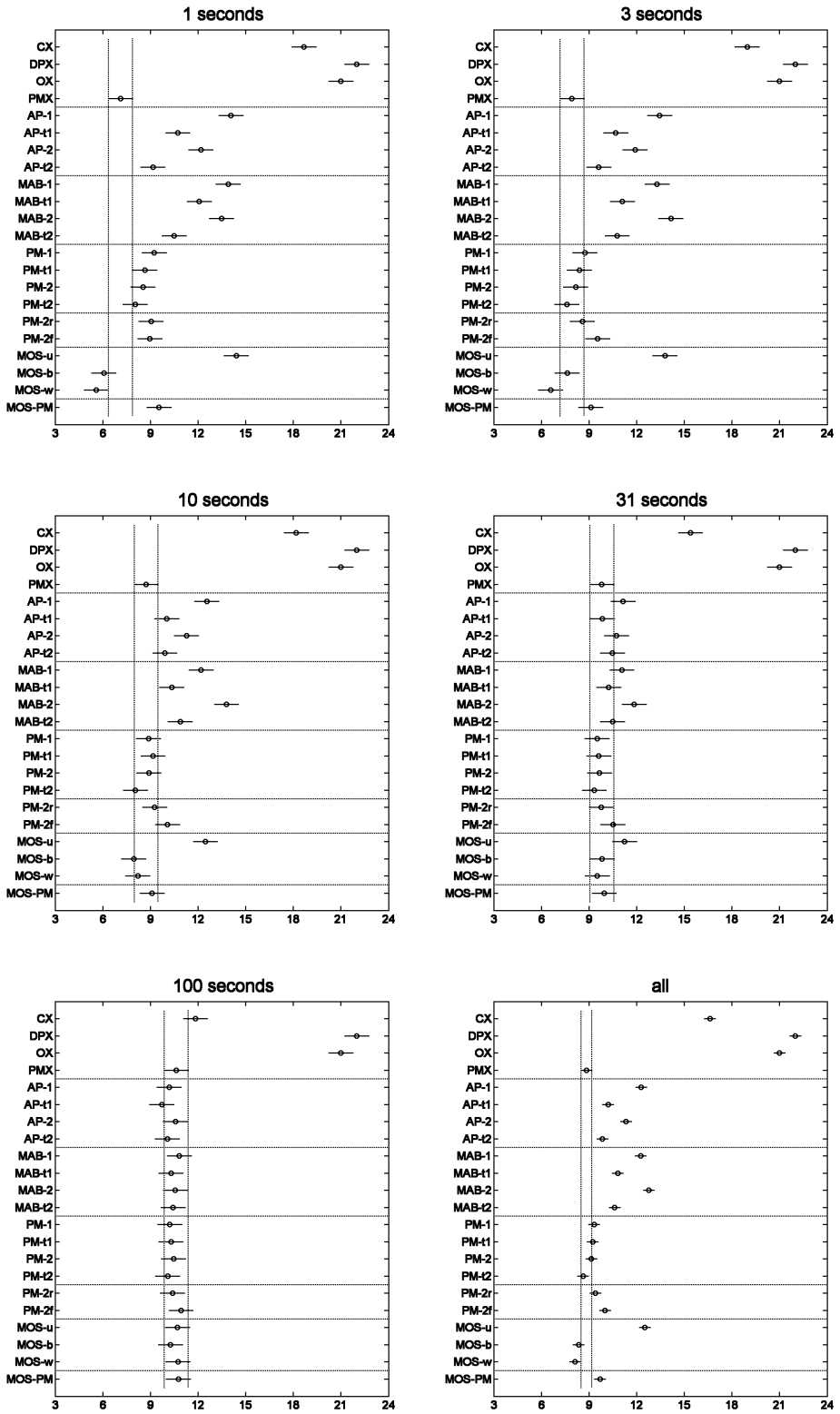


Figure C.3: The ranking performance of different operator selection methods (acronyms see text) with different computation time levels 1, 3.1, 10, 31, and 100 seconds on homogeneous easy instance set.

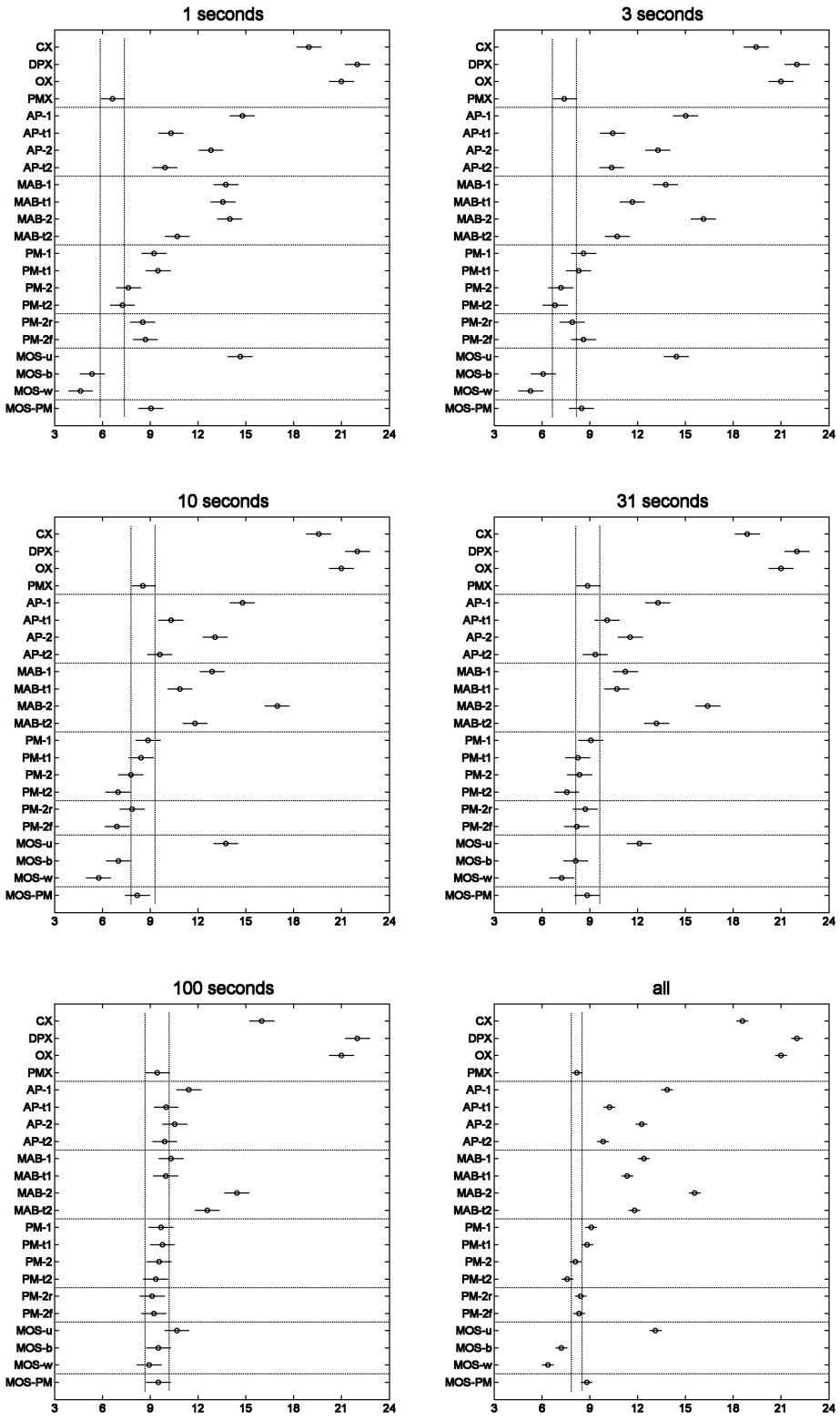


Figure C.4: The ranking performance of different operator selection methods (acronyms see text) with different computation time levels 1, 3.1, 10, 31, and 100 seconds on homogeneous hard instance set.

Table C.2: The number of statistically significant scenarios among the four SOS methods (CX, DPX, OX, and PMX) in the training instance set (left) and testing instance set (right). Each entry lists three numbers, one in each of the three instance classes, `het`/`hom-easy`/`hom-hard`, respectively. A number in each entry indicates in how many out of the 80 scenarios in each instance class, the method named on the left of the row is statistically significantly better than the method named on the top of the column. The statistical significance is tested by the Wilcoxon's signed rank test with Holm's adjustment for multiple test correction at an α level of 0.05.

	Training Performance				Testing Performance			
	CX	DPX	OX	PMX	CX	DPX	OX	PMX
CX	–	80/80/80	75/80/80	0/0/0	–	80/80/80	80/80/80	0/0/0
DPX	0/0/0	–	0/0/0	0/0/0	0/0/0	–	0/0/0	0/0/0
OX	5/0/0	80/80/80	–	5/0/0	0/0/0	80/80/80	–	0/0/0
PMX	62/65/78	80/80/80	75/80/80	–	61/64/80	80/80/80	80/80/80	–

Online adaptation methods with default setting

First we take a look at the untrained versions of the online adaptation methods with default settings. It appears that the new reward function R^2 benefits AP and PM, while worsens MAB. AP-2 significantly improves AP-1 in 7 out of 15 case studies by Friedman's test, and 25 out of 240 scenarios by Wilcoxon's test as shown in Table C.3; PM-2 significantly improves PM-1 in only one case study and 9 scenarios, all of which are in either the `het` set or the `hom-hard` set. Using the new reward function in MAB-2 results in significantly worse performance than MAB-1 in 6 case studies and 76 scenarios. In general, PM is the best performing adaptation method, especially PM-2. PM-2 significantly outperforms the best untrained MAB method MAB-1 in 11 out of 15 case studies, and it also significantly outperforms the best untrained AP method AP-2 in 10 case studies.

Comparing with the best performing static operator PMX, the adaptation methods in general perform better in the heterogeneous set than in the homogeneous sets, and perform better as the computation time increases. Take the untrained PM variants for example, PM-1 never significantly outperforms PMX in all case studies, which is consistent with the empirical studies in [82]. Besides, PM-1 is significantly outperformed by PMX in all the 1-second case studies. Another PM variant PM-2 is never significantly outperformed by PMX in all the 15 case studies; and it performs significantly better than PMX in 3 out of 15

Table C.3: The number of statistically significant scenarios among the four versions of AOS methods (using reward function R^1 untrained -1 and trained -t1; using reward function R^2 untrained -2 and trained -t2) in Adaptive Pursuit (AP), Multi-Arm Bandit (MAB), and Probability Matching (PM). Each entry lists three numbers, one in each of the three instance classes, `het`/`hom-easy`/`hom-hard`, respectively. A number in each entry indicates in how many out of the 80 scenarios, the method named on the left of the row is statistically significantly better than the method named on the top of the column. The statistical significance is tested using the Wilcoxon's signed rank test with Holm's adjustment for multiple test correction at an α level of 0.05.

	AP				MAB				PM			
	-1	-t1	-2	-t2	-1	-t1	-2	-t2	-1	-t1	-2	-t2
-1	–	0/0/0	0/0/0	0/0/0	–	0/0/0	24/3/49	6/1/8	–	1/1/0	0/0/0	0/0/0
-t1	35/23/50	–	18/3/22	0/1/1	9/9/12	–	27/25/56	7/1/13	4/1/0	–	0/2/0	0/0/0
-2	11/5/9	0/0/0	–	0/0/0	1/1/0	1/0/0	–	0/0/0	5/0/4	3/0/4	–	0/1/0
-t2	40/32/58	1/1/2	23/12/25	–	22/19/17	12/4/4	32/23/51	–	12/0/6	7/2/5	1/0/1	–

case studies, as the instances are heterogeneous as in the `het` set, and the computation time is long enough (10, 31, and 100 seconds).

Offline configuration of online adaption

Instead of using a fixed default setting, the hyper-parameters of online adaptation methods can also be fine-tuned for each case study by offline configuration. As shown in the Figures C.2, C.3, and C.4 for each case study, as well as in Table C.3 for each scenario, the offline tuned parameter settings usually improve the performance of the online adaption methods with default settings. The improvement is especially noticeable for the poor performing adaptation methods such as AP and MAB. For example, the MAB-t2 statistically significantly improves the performance of MAB-2 in 12 out of 15 case studies and 105 out of 240 scenarios. Likewise, AP-t1, AP-t2, and MAB-t1 also significantly improves AP-1, AP-2, and MAB-1, in 11, 9 and 5 case studies, and in 108, 60, and 30 scenarios, respectively. The performance improvement by offline configuration is particularly noticeable when computation time is under 10 seconds (amounts to 9 000 generations). It appears that the performance of AP and MAB are more sensitive to their parameters and the reward function used; especially the scaling factor γ in MAB needs to be fine-tuned when the reward function is changed. For our best online method PM, although applying offline configuration usually

improves its default setting by slightly better ranking, the improvement is usually not statistically significant: both PM-1t and PM-2t do not significantly improve their respective default settings in any of the 15 case studies. Scenario-wise, our best untrained AOS method PM-2 is significantly improved by offline configuration in only 2 out of 240 scenarios, and it also significantly outperforms PM-t2 in 1 scenario; Similarly, PM-1 is significantly improved by tuning in 5 scenarios, and it significantly outperforms its tuned version in 2 scenarios. Nevertheless, note that our offline configuration approach assumes no a priori knowledge about the parameter space, while the default setting are usually determined by expert knowledge and extensive experiments. It is noteworthy that the offline tuned settings never perform significantly worse than the default setting in any of the 15 case studies by Friedman's test. Therefore, the offline configuration should be applied for setting hyper-parameters of the online adaptation methods, especially if one faces new problem domain or new setting of reward function.

Comparison of tuned online adaptation methods

As shown in Figures C.2, C.3, and C.4, the tuned PM variants, especially PM-t2, is clearly the best performing tuned online adaptation method. It significantly outperforms the two tuned MAB variants in 12 out of 15 case studies, except in the high computation time levels of the `hom-easy` and `hom-hard` set; and it also significantly outperforms the two tuned AP variants in 9 out of 15 case studies. The performance of PM-t2 and PM-t1 is quite close, PM-t2 significantly outperforms PM-t1 in only one case study in the `hom-hard` set with 1 second shown in Figure C.4, and in 14 scenarios as shown in Table C.3; and PM-t1 never shows an edge over PM-t2 in any of the 15 case studies or any of the 240 scenarios. PM-t2 performs significantly better than PMX in five out of 15 case studies, where four of them are in the heterogeneous set, and it performs never significantly worse. The tuned AP variants in general performs worse than PM variants and PMX, however, the automatically tuned AP-t1 becomes best performing for the case study of `hom-easy` set in 100 seconds as shown in Figure C.3. For the worst adaptation methods MAB, an interesting phenomenon has been observed. Although the untrained MAB-1 usually outperforms MAB-2, the tuned MAB-t2 usually significantly outperforms MAB-t1 in the short runtime cases such as 1 and 3.1 seconds by Friedman test as shown in Figures C.2, C.3, and C.4. This further confirms that MAB is very sensitive to its parameter setting, and offline configuration should always be applied when the problem domain, reward function, or computation time criterion has been changed.

Table C.4: The number of statistically significant scenarios among the three methods: PM-2, and two MOS methods PM-2r, and PM-2f based on the empirical distribution of PM-2, on the three instance classes `het`/`hom-easy`/`hom-hard`, respectively. Each instance class contains 80 scenarios. Statistical significance (left significantly better than above) is based on the Wilcoxon's signed rank test with Holm's adjustment for multiple test correction ($\alpha = 0.05$).

	PM-2	PM-2r	PM-2f
PM-2	–	1/0/1	3/2/2
PM-2r	0/2/0	–	3/4/0
PM-2f	1/1/2	0/0/4	–

C.7.4 Analysis on the effectiveness of online adaptation

It is generally believed that the online adaptation methods have two potential advantages over using a fixed parameter configuration:

1. **instance-specific adaptation.** Online adaptation mechanisms may adapt to the best setting for each particular instance. When the instances are heterogeneous, the best parameter setting across a whole class of instances may perform poorly for some individual instances.
2. **Search-stage adaptation.** Online adaptation may also adapt better to different search stages of the algorithm. At different search stages of an evolutionary algorithm, different operators with different explorative or exploitive abilities may be preferred.

In the following, we set up experiments to test which of the two factors above attribute the most to the effectiveness of the adaptation method. We take the most effective untrained adaptation method PM-2 for this study.

Adaptation to search stages

We first set up experiments inspired by [160] to test the second factor above. For each run r on each instance in the testing set, we keep track of the number of usage n_i^r of each operator i in PM-2, and then randomly generate operators based on the probability distribution $p_i^r = n_i^r / \sum_i n_i^r$. The operator probability p_i^r is unchanged through the whole algorithm run. This amounts to using a MOS approach based on the empirical probability distribution of operator usage in PM-2. We

further allowed for each MOS run exactly the same number of total operator generations as in PM-2. In such case, we observed that MOS may finish around 1% earlier than PM-2 (i.e., using around 99% of the time needed for PM-2) due to the ease of computational overhead caused by the adaptation mechanism in probability matching. This result of the MOS run is denoted as PM-2r as shown at the fifth block of each plot in Figure C.2, C.3, and C.4. Note that the empirical distribution p_i^r in PM-2r is learned for each run r on each instance, therefore, it is still instance-specific, while the sequence of crossover operator usage is randomized, and hence the search-stage adaptation is corrupted. If the second factor affects significantly the effectiveness of the adaptation methods, PM-2r should perform significantly worse than PM-2.

The results show that PM-2r performs slightly worse than PM-2 in terms of median ranks, but the difference between them is never significant in each of 15 case studies as shown in Figures C.2, C.3, and C.4. PM-2r performs noticeably worse than PM-2 in few case studies where the trained MOS such as MOS-w (see results in Section C.7.5) significantly outperform PM-2, including small computation time (1 or 3.1 seconds) in the two homogeneous sets. This may be because PM-2 cannot adapt to a good operator distribution in such a short computation time, and in such case, search stage adaptation shows a slight advantage. For the scenario-wise comparison shown in Table C.4, PM-2 is significantly better in only one out of 80 scenarios in the `het` set, one out of 80 scenarios in the `hom-hard` set, while it is significantly worse than PM-2r in two scenarios in the `hom-easy` set. To sum up, we observed only negligible advantage due to search stage adaptation in our best adaptation method.

Adaptation to specific instances

In the same block with PM-2r, the other method PM-2f is modified from PM-2r by corrupting also the first factor. This is done by using a fixed operator probability distribution for all instances in each case study. The fixed probability p_i is obtained by averaging the empirical operator probability p_i^r of PM-2 across all instances in each case study. This amounts to using MOS to randomly generating operators by the average empirical operator distribution of PM-2. Similar to PM-2r, we terminate each MOS run early by the same number of generation in PM-2. If the first factor, the instance-specific adaptation, influences significantly the effectiveness of the adaptation methods, PM-2f should perform significantly worse than the instance-specific PM-2r and PM-2.

In the comparison of PM-2f with PM-2r, it was expected that the advantage of PM-2r is more significant in the heterogeneous set than in

the homogeneous set. However, surprisingly, the advantage of PM-2r with instance-specific operator distribution is more significant in the hom-easy set than in the het set. It is also noteworthy that PM-2f even performs better than PM-2r and PM-2 in the hom-hard set, especially in 10 and 31 seconds case (see Figure C.4). This shows that the empirical operator distribution of PM-2 may be quite poor for some hom-hard instances, and it can be recovered by the average distribution. Scenario-wise comparison in Table C.4 also confirms that PM-2f performs significantly better than PM-2r in 4 scenarios in the hom-hard set, while being significantly worse than PM-2r in 3 scenarios in the het set and 4 scenarios in the hom-easy. Even comparing PM-2f to PM-2, no significant advantage of PM-2 can be observed in any of the 15 case studies; PM-2 significantly outperforms PM-2f in 7 scenarios, while being significantly outperformed in 4 scenarios. To sum up, no significant performance advantage is observed by using instance-specific adaptation. Slight performance improvement due to instance-specific adaptation can be observed only in the computationally easy instances, but the reverse may be true for the hard instances, where using instance-specific operator distribution performs even worse than a fixed operator distribution averaging across all instances. In fact, the performance difference due to search stage adaptation and instance-specific adaptation is smaller than, for example, using a different adaptation method, or using a different reward function, or even using a different parameter setting.

Since the two MOS methods PM-2f and PM-2r with the empirical distribution of the best online adaptation method PM-2 do not perform significantly worse than the PM-2, could we find a better operator distribution for MOS? In the next section, we explore the possibility of finding good operator distribution for MOS by offline configuration.

C.7.5 Mixed operator strategy

The observation in Section C.7.4 has motivated us to further investigate the simple non-adaptive operator selection strategy MOS. The ranking performance of the three MOS based approaches, an untrained MOS-u, and offline tuned MOS-b and MOS-w, are illustrated in the sixth block of each plot in Figures C.2, C.3, and C.4. We further compare the three MOS methods together with the best SOS method PMX and the best AOS methods AP-t2, MAB-t2, and PM-t2. Their pairwise statistical hypothesis test results are presented in Table C.5, and their average ranking by computation time is shown in Figure C.5.

Comparison of MOS and SOS

Firstly, the two tuned MOS methods MOS-b and MOS-w significantly improves over the default MOS-u with uniform probability in most of the case studies. MOS-w significantly outperforms MOS-u in 14 case studies except only `hom-easy` set with 100 seconds, and MOS-b significantly improves MOS-u in 12 out of 15 case studies by Friedman's test. The ranking difference is especially vast when the computation time is small. MOS-b and MOS-w significantly improve MOS-u in 139 and 146 out of 240 scenarios, respectively. MOS-w appears to be a slightly better way of tuning MOS compared to MOS-b but the difference between them is not statistically significant in any of the 15 case studies by Friedman's test. MOS-w significantly outperforms MOS-b in two scenarios in the `hom-easy` set and one scenario in the `hom-hard`, while MOS-b significantly outperforms MOS-w in one `het` scenario by Wilcoxon's test. Both MOS-w and MOS-b perform better than the offline tuned static operator PMX, especially when the instances are heterogeneous as in the `het` set, and when the instances are hard as in the `hom-hard` set. MOS-w significantly outperforms PMX in 10 out of 15 case studies, including all the five computation time levels in `het` set, and four computation time levels in `hom-hard` set, as well as `hom-easy` set with 1 second runtime. MOS-w significantly outperforms PMX in 13 scenarios in the `het` set, 6 scenarios in the `hom-hard` set, and 3 scenarios in the hom-easy set, while it is never significantly outperformed by PMX. As can be also observed in Figure C.5, the average rank of PMX quickly deteriorates to be one of the worst methods as computation time increases (in the `het` set and `hom-easy` set with 100 seconds), the ranking performance of MOS-b and MOS-w still remains one of the best methods as runtime increases. This interesting result indicates that, even if an operator that is dominantly better than the others exists, such as PMX in our case, varying the choice of operators at runtime can still result in significantly better and more robust strategy than a static choice. This also sheds some light on how offline configuration can be conducted: instead of finding one static configuration, varying the parameter values at runtime by a static distribution trained offline may be a better idea. In fact, varying parameter values at runtime by a non-adaptive distribution is also applied to set the tabu list length in robust tabu search [220], a state-of-the-art algorithm for QAP.

Comparison of MOS and AOS

It is interesting to see that even our fine-tuned best-performing online adaptation methods are outperformed by the tuned mixed operator

Table C.5: The number of statistically significant scenarios among the seven best methods: the best SOS method PMX; the best AOS methods AP-t2, MAB-t2, and PM-t2; three MOS methods, including an untrained MOS-u, and two trained MOS-b and MOS-w. Each entry shows how many out of 80 scenarios the left method is significantly better than the above method in one of the three instance classes `het`/`hom-easy`/`hom-hard`, respectively. Statistical significance is tested using the Wilcoxon's signed rank test with Holm's adjustment for multiple test correction ($\alpha = 0.05$).

	PMX	AP-t2	MAB-t2	PM-t2	MOS-u	MOS-b	MOS-w
PMX	–	2/2/13	12/10/39	0/1/0	32/36/49	0/0/0	0/0/0
AP-t2	1/0/0	–	7/3/8	0/0/0	30/26/38	0/0/0	0/1/0
MAB-t2	0/0/0	0/0/0	–	0/0/0	19/16/18	0/0/0	0/0/0
PM-t2	9/0/1	7/3/12	24/10/42	–	43/38/56	0/0/0	0/0/0
MOS-u	0/0/0	0/0/0	3/0/1	0/0/0	–	0/0/0	0/0/0
MOS-b	8/1/2	12/7/28	32/16/54	2/2/3	45/40/54	–	1/0/0
MOS-w	13/3/6	22/11/39	35/25/64	5/3/8	46/40/60	0/2/1	–

strategies. MOS-w performs noticeably better than PM-t2 especially in the hard instances such as in the hom-hard set, where MOS-w exhibits significant better overall performance over PM-t2; or when the computation time is small, such as in 1 or 3.1 seconds cases. The advantage of MOS-w over PM-t2 is statistically significant in five of 15 case studies, including three case studies (1, 3.1, 10 seconds) in hom-hard set, and 1 second case study in hom-easy and het set. MOS-b also significantly outperforms PM-t2 in all the 1-second case studies. MOS-w and MOS-b significantly outperform PM-t2 in 16 and 7 scenarios, respectively, while PM-t2 never performs significantly better than them in any of the 15 case studies or any of the 240 scenarios. Both MOS-b and MOS-w also significantly outperform all variants of AP and MAB in most of the case studies. MOS-w significantly outperforms AP-t2 in 10 out of 15 case studies by Friedman's test and 72 out of 240 scenarios by Wilcoxon's test; and it also significantly outperforms MAB-t2 in 12 case studies and 124 scenarios. The runtime development of these methods can be also visualized in Figure C.5. As can be shown, the relative ranking difference gets closer as time goes by. The relative performance of AOS methods tend to improve, while the non-adaptive methods, especially the static operator method PMX performs relatively worse as runtime increases. The untrained MOS-u, also known as "naive" approach in the online adaptation literature, appears to be

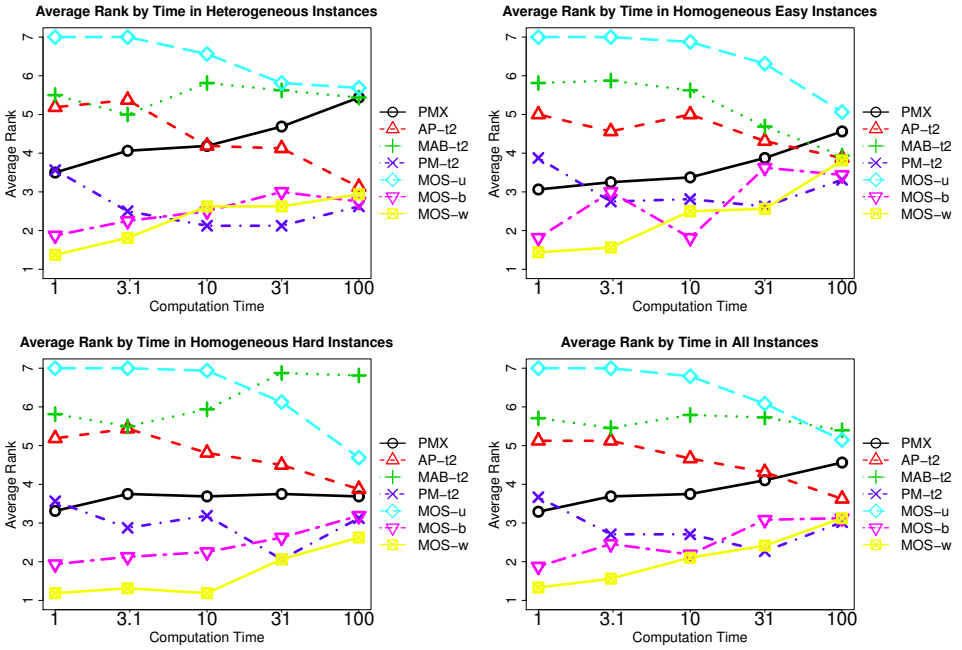


Figure C.5: The average ranks of the seven best operator selection methods over computation time for the heterogeneous (`het`), homogeneous easy (`hom-easy`), homogeneous hard (`hom-hard`), as well as across all instances. The seven methods listed here include the best SOS method PMX, the best AP method (AP-t2), the best MAB method MAB-t2, the best PM method PM-t2, and the three MOS methods: an untrained MOS-u, and two trained methods MOS-b and MOS-w. Five computation time 1, 3.1, 10, 31, and 100 seconds are considered, corresponding to ca. 900, 3 000, 9 000, 30 000, and 90 000 EA generations.

worse than the AOS methods, while MOS-w appears to be the overall best performing method, and its advantage is especially vast for hard instances and short runtime.

C.7.6 Combining MOS and AOS

We further investigate the possibility to incorporate both MOS and AOS together. The best MOS and AOS version found in this work, MOS-w and PM-2, respectively, are used for this study. This hybrid, named MOS-PM, is to tune the MOS-w parameters q_i (the quality vector of each operator i) in the training set, and then use this to initialize the quality vector Q_i for each operator i in PM-2 by setting $Q_i = q_i / \max_i q_i$. The scaling by setting the maximum initial Q_i to 1 is to make Q_i consistent with the magnitude in the reward function

R^2. This approach amounts to a biased initial condition for the online adaptation method by an operator distribution trained offline.

However, as shown in the last block of Figures C.2, C.3, and C.4, MOS-PM does not bring any improvement to

- MOS-w, which shows that the online adaptation method cannot further improve a well-tuned non-adaptive mixed operator strategy. This agrees with the study by [185] that online adaptation methods cannot improve an offline tuned static parameter configuration in ACO.
- PM-2, which shows that it may not be suitable to use the fine-tuned operator usage frequency during a whole algorithm run to initialize the operator probability distribution. In fact, at the initial stage of an algorithm, it may be more advantageous to use a more explorative parameter setting, while a more exploitive parameter setting may be preferred in later stages. We further confirm our hypothesis by tuning the initial operator quality Q_i together with α and P_{min} in PM-2. And indeed in the tuned Q_i, the average initial probabilities of the two more explorative operators OX and DPX are even higher than the two more exploitive operators CX and PMX. However, no noticeable performance improvement can be observed comparing with tuning only α and P_{min} as in PM-t2. The same observation can be obtained on MAB and AP, where tuning initial condition doesn't improve adaptation performance, as long as its hyper-parameters for adaptation are already fine-tuned.

C.8 Conclusions and future works

In this work, we provide an empirical investigation of applying offline parameter configuration into the online parameter adaptation on the operator selection problem in evolutionary algorithm. We extended [82] by incorporating offline configuration with the dynamic operator selection methods, including: (i) three adaptive operator selection (AOS) methods: Probability Matching, Adaptive Pursuit, and Multi-Armed Bandit; (ii) a non-adaptive mixed operator strategy (MOS), which assigns a probability distribution for each operator to be selected. State-of-the-art offline algorithm configuration tools are applied to this end, including iterated racing [37] and post-selection techniques [242]. One major contribution in this study is to identify an automatically tuned MOS as one of the best performing approaches for operator selection. The results show that even when a dominantly best choice of static

operator exists, randomizing the operator usage by an automatically tuned operator probability distribution still significantly outperforms the best static operator approach. This also sheds some light to the future design of offline algorithm configuration: instead of tuning for a static parameter configuration, it may be a better idea to tune a distribution from which the parameter configurations are randomly generated and changed during algorithm run. Besides, we also showed that the performance and robustness of online AOS methods can be improved by considering an offline configuration of its hyper-parameters. Our investigation also showed that the ability to adapt to particular instance and adapt to different search stages attributes only insignificantly to the effectiveness of the best adaptation method in our context.

One limitation of our current tuned MOS approach is that it cannot adapt to different instances. An interesting direction for future research is to use more advanced offline configuration techniques to obtain instance-aware operator distributions for MOS. The portfolio-based approach is one example: instead of tuning one probability distribution, we can dynamically build a portfolio of complementary distributions for instances with different features [229]. Another interesting alternative is the instance-specific approach [125], which automatically cluster different instances, and then apply offline configuration to obtain an operator distribution for each cluster of instances. Encoding instance-specific expert knowledge into the evolutionary algorithm [180] should be also useful. Applying such instance-aware configuration to MOS will further give it an edge over the AOS, and make it a promising direction for operator selection.

Appendix D

Automatic configuration of MIP solver: Case study in vertical flight planning

D.1 Introduction

Mixed Integer Programming (MIP) is a general way for modelling many real-world optimization problems. A general-purpose solver for MIP, such as CPLEX, Gurobi, or SCIP, can be regarded as a matheuristic algorithm that integrates the exact branch-and-cut algorithm with various MIP-based heuristics such as local branching, feasibility pump, among others (cf. [80] for a survey). Such MIP solvers usually contain a large number of parameters, e.g., IBM ILOG CPLEX 12.6 has a total of 159 parameters [120]. Most of these parameters are related to hardware or tolerance, e.g., available working memory, number of threads, random seed, time limit, optimum tolerance, constraint violation tolerance, etc. These should be fixed according to the hardware in use and the practicality of the problem. Besides, there are algorithmic parameters of different types, e.g., categorical parameters, such as options of branching strategies, LP method, whether to use certain heuristic; numerical parameters, such as how often a certain heuristic or perturbation is applied, cut limits, and so on; and conditional parameters, such as perturbation constant is only used when perturbation is switched on, or limit of strong candidate list or strong candidate iteration is only used when strong branching is selected. These algorithmic parameters could potentially be automatically configured for each particular class of problems. There exists work on automatic configuration of the MIP solvers [115] using a configuration software ParamILS [118]. Other general-purpose configurators for such task also include iterated

racing [37], GGA [5], SMAC [112], etc.

In this work, we empirically compare three classic piecewise linear formulations, as it arises from the real-world vertical flight planning problem [240, 238], by automatic configuration of MIP solver. We also investigate the performance variability of the MIP solver due to both random seed settings and parameter configurations.

D.2 Vertical flight planning

The vertical flight planning (VFP) problem concerns assigning optimal cruise altitude and speed to each trajectory-composing segment, such that the fuel consumption is minimized, and the arrival time constraints are satisfied. The original MIP models for VFP [240, 239] that assign continuous speed consist of second-order cone constraints, which lead to prohibitively long computation time. The speed discretization scheme proposed in [238] transforms the nonlinear model into linear, and significantly reduces the computation time to within seconds or minutes.

D.2.1 Mixed integer linear programming model

The MIP model of VFP using discrete speed [238] is as follows.

$$\min \quad w_0 - w_n \tag{D.1}$$

$$\text{s.t.} \quad t_0 = 0, \quad \underline{T} \leq t_n \leq \overline{T} \tag{D.2}$$

$$\forall i \in S: \quad \Delta t_i = t_i - t_{i-1} \tag{D.3}$$

$$\forall i \in S: \quad \sum_{v \in V} \mu_{i,v} = 1 \tag{D.4}$$

$$\forall i \in S: \quad \Delta t = \sum_{v \in V} \mu_{i,v} \cdot \Delta T_{i,v} \tag{D.5}$$

$$w_n = W^{dry} \tag{D.6}$$

$$\forall i \in S: \quad w_{i-1} = w_i + f_i \tag{D.7}$$

$$\forall i \in S: \quad f_i = \sum_{v \in V} \mu_{i,v} \cdot \widehat{F}_{i,v}(w_i). \tag{D.8}$$

The total fuel consumption (D.1) measured by the difference of aircraft weight before and after the flight is minimized; (D.2) ensures the flight duration within a given time window; (D.3) preserves the time consistency; Only one speed is assigned to each segment by (D.4), and the travel time on each segment depends on the speed assignment by (D.5). (D.6) initializes the weight vector by assuming all trip fuel is

burnt during the flight; weight consistency is ensured in (D.7) and the fuel consumption of each segment in (D.8) is calculated based on the speed selection μ and a piecewise linear function $\widehat{F} : [w_0, w_{max}] \to \mathbb{R}$ interpolating the fuel consumption F based on weight. The piecewise linear function can be modelled by three different formulations.

The Convex Combination (Lambda) Method. A variant of the *convex combination* (*Lambda*) method [57] can be formulated as follows. To interpolate F we introduce binary decision variables $\tau_k \in \{0, 1\}$ for each $k \in K$, and continuous decision variables $\lambda_k^l, \lambda_k^r \in [0, 1]$ for each $k \in K$.

$$\sum_{k \in K} \tau_k = 1 \tag{D.9a}$$

$$\forall\, k \in K: \quad \lambda_k^l + \lambda_k^r = \tau_k \tag{D.9b}$$

$$w = \sum_{k \in K} (w_{k-1} \cdot \lambda_k^l + w_k \cdot \lambda_k^r) \tag{D.9c}$$

$$\widehat{F}(w) = \sum_{k \in K} (F(w_{k-1}) \cdot \lambda_k^l + F(w_k) \cdot \lambda_k^r) \tag{D.9d}$$

The Incremental (Delta) Method. The *incremental (Delta) method* was introduced by Markowitz and Manne [155]. It uses binary decision variable $\tau_k \in \{0, 1\}$ for $k \in K$ and continuous decision variables $\delta_k \in [0, 1]$ for $k \in K$, and

$$\forall\, k \in K: \quad \tau_k \geq \delta_k \tag{D.10a}$$

$$\forall\, k \in K \setminus \{n\}: \quad \delta_k \geq \tau_{k+1} \tag{D.10b}$$

$$w = w_0 + \sum_{k \in K} (w_k - w_{k-1}) \cdot \delta_k \tag{D.10c}$$

$$\widehat{F}(w) = F(w_0) + \sum_{k \in K} (F(w_k) - F(w_{k-1})) \cdot \delta_k \tag{D.10d}$$

The Special Ordered Set of Type 2 (SOS) Method. Instead of introducing binary variables for the selection of a particular interval, we mark the lambda variables as belonging to a *special ordered set of type 2* (*SOS*). That is, at most two adjacent variables from an ordered set $(\lambda_0, \lambda_1, \ldots, \lambda_m)$ are positive. Such special ordered sets are treated by the solver with a special SOS branching [24]. We introduce continuous decision variables $0 \leq \lambda_k \leq 1$ for each $k \in K_0$, and:

$$\mathrm{SOS}(\lambda_0, \lambda_1, \ldots, \lambda_m) \tag{D.11a}$$

$$w = \sum_{k \in K} (w_{k-1} \cdot \lambda_{k-1} + w_k \cdot \lambda_k) \tag{D.11b}$$

$$\widehat{F}(w) = \sum_{k \in K} (F(w_{k-1}) \cdot \lambda_{k-1} + F(w_k) \cdot \lambda_k) \tag{D.11c}$$

The comparison of these classic piecewise linear formulations has been the topic in many scientific publications on various optimization problems, including gas network design [157], water network design [90], transportation [224], process engineering [86], flight planning [239], etc. SOS method was found the best in [157], while Delta method was best in [90], and mixed results among the three methods were presented in [224, 86, 239], despite the superior theoretical property of the Delta method over the Lambda method [181].

D.2.2 Problem instance

Two of the most common aircraft types, Airbus 320 (A320) and Boeing 737 (B737), are used for our empirical study. The aircraft performance data are provided by Lufthansa Systems AG. We generated random instances including three flight ranges for A320: 1000, 2000, and 3000 nautical miles (NM), and one flight range of 1500 NM for B737. Two types of segment lengths are generated, the homogeneous instances include segment lengths uniformly randomly generated from 40 to 60 NM, while the heterogeneous instances include segment lengths generated from a uniform distribution from 10 to 90 NM. The expected numbers of segments are 20, 30, 40, and 60 for flight ranges of 1000, 1500, 2000, and 3000 NM, respectively. Three time constraints are considered which require the aircraft to accelerate over its unconstrained optimal speed by factors of 2%, 4%, and 6%. For each of the four aircraft ranges with two homogeneities and three acceleration factors, 10 random instances were generated, totalling 240 instances for testing purpose. Besides, another 240 instances were generated in the same way with different random seeds for training purpose. It is worth noting that these instances represent a broad range of the vertical flight planning problem.

D.3 Performance variability of MIP solver

The MIP solvers were believed by many researchers and practitioners for a long time to be deterministic and perfect for benchmarking. The issue of *performance variability* in MIP solvers was first introduced to the MIP community by [56], where she reported a seemingly neutral change in the computing environment can result in a drastic change in solver performance. An experiment in [79] also reported a drastic performance variability of up to 25% by adding redundant

constraints. A performance variability of up to a factor of 915 for the MIPLIB instances is also observed in [134] by randomly permuting rows or columns of a MIP model. The roots of such performance variability were first explained in [134] to be *imperfect tie-breaking*. There are many decisions to make in a branch-and-cut process, e.g., the cut separation, cut filtering, and the order of the variables to branch on, etc. Such decisions are made based on ordering the candidates by a score. However, it is impossible to have a perfect score that uniquely distinguishes all candidates at each step. When a tie in the score occurs, a deterministic choice was always made, e.g., by taking always the first candidate. Therefore, changing the order of the variables or constraints will lead to a change of path in the tree search, thus very different behavior and performance in MIP solver. It was further argued in [149] that even if a perfect score for tie-breaking exists, it may be too expensive to compute. Therefore, randomness is intrinsic in MIP solvers, and one can even exploit the randomness to improve performance as in [81], where a *bet-and-go* approach was proposed that tried out a number of different neutrally perturbed settings for a short runtime, and then pick the best setting and continue for a long run. However, performance variability must be taken into account in scalability study and benchmarking different formulations or new algorithmic ideas to avoid misinterpretation. A more detailed discussion on performance variability can be found in [149].

In response to the issue of performance variability, MIP solvers start to break ties randomly, and allow users to specify a random seed to initialize the random number generator, e.g., CPLEX since 12.5. Ready or not, it is time for us to accept the fact that MIP solvers are randomized algorithms. An experimental study of the MIP solvers should follow a proper experimental setup for randomized algorithms, cf. [163, 122]. An empirical study of a (randomized) MIP solver based on a fixed random seed (as done in e.g. [157, 224, 90, 86, 239]) will limit the conclusion to only a specific implementation of the MIP solver with a peculiar sequence of random numbers. A proper experimental setup for studying a specific problem instance should be performed by running MIP solver with multiple random seeds and collecting proper statistics; empirical study for a problem class should include a wide range of instances from the problem class, and assign to each instance a different random seed. Each instance can be paired with a unique random seed, such that algorithmic candidates are evaluated on each instance with the common random seed, so as to reduce evaluation variance [162].

All experiments ran on a computing node with 12-core Intel Xeon X5675 CPU at 3.07 GHz and 48 GB RAM. We use CPLEX 12.6 with

Table D.1: The performance variability of CPLEX with default setting under two random seeds for each testing instance: a fixed default seed and a randomly generated seed.

Method	1 thread					12 threads				
	#solved	var.coef.		$\frac{max}{min}$		#solved	var.coef.		$\frac{max}{min}$	
	def./ran.seed	avg.	max.	avg.	max.	def/ran.seed	avg.	max.	avg.	max.
Lambda	233/236	0.28	1.84	1.91	23.79	240/240	0.15	0.82	1.18	2.39
SOS	48/49	0.33	1.06	1.50	3.23	64/64	0.37	1.72	1.82	13.49
Delta	227/230	0.34	1.48	1.59	6.70	220/220	0.22	1.37	1.31	5.31

both single thread and 12 threads. We fixed the memory parameters such as working memory (`WorkMem`) to 40 GB, and the node storage file switch (`NodeFileInd`) to 3. We first ran CPLEX with the default setting on the 240 testing instances. Two random seed settings were run, one with CPLEX default fixed seed, and the other assigned a different random number to each instance as random seed. The goal is to evaluate the performance variability of CPLEX due to different random seed settings. Two types of variability measure are used: (i) *variation coefficient* (termed variability score in [134]), which is a relative variability measure defined as the standard deviation divided by the mean, i.e., $\frac{2\cdot|t_1-t_2|}{t_1+t_2}$ where t_1 and t_2 are the computation time of the two random seed settings; (ii) the *ratio* of the maximum and minimum computation time between the two random seeds for each instance, which is also mentioned in [134]. Both the average and the maximum of both variability measures across all instances are presented in Table D.1. The variability measure is calculated with only the instances that are solved within 300 seconds. The variability of CPLEX due to random seed on vertical flight planning problem is certainly not negligible. The average variation coefficient is between 0.15 to 0.37, while average ratio is between 1.18 to 1.91. In general, the variability is higher in single thread than parallel computation, especially in the Lambda method: both its average measures are 60% to 85% higher in single thread, and it has the highest max-min-ratio of 23.79, which is on an instance that is solved by a randomly generated seed in 12.6 seconds, but takes the CPLEX with default seed 300 seconds and still leaves a 0.02% gap. The variability appears to increase for models that are harder to solve, i.e., the variability of the SOS is higher than the Delta and then higher than the Lambda method. Note that the variability measure for especially the SOS method is probably an underestimate, since it is only calculated on a small set of solved instances. Hence although the parallel SOS seems more variable than the sequential, the

high variability scores are mainly contributed by the 15 additional instances that are solved in the parallel SOS but not in the sequential SOS. Although using the randomly generated seed solves a few more instances to optimality than using the default seed in 1-thread case, no statistical significant difference is found by Wilcoxon's signed rank test or binomial test.

D.4 Automatic configuration of MIP solver

D.4.1 Automatic solver configuration

The MIP solvers are highly parameterized matheuristic algorithms. CPLEX 12.6 has a total of 159 parameters, where around 70 to 80 parameters can influence algorithmic behaviors. Although CPLEX claimed that "A great deal of algorithmic development effort has been devoted to establishing default ILOG CPLEX parameter settings that achieve good performance on a wide variety of MIP models." [120, p. 222], this configuration needs not be a good choice for a specific problem class of interest. Experimental study using MIP solver with only the default configuration will limit the study to only a specific implementation of the MIP solver with a particular parameter setting rather than a general algorithm. Especially when empirically comparing algorithmic ideas, there will be high risk of misinterpretation due to the interaction of the algorithmic idea with certain algorithmic parameters of CPLEX. Each algorithmic idea to be compared should be given a fair amount of configuration effort by the same procedure.

The automatic configuration experiments are performed on the 240 training instances. 11 copies of the training set of the instances are generated, each using a different random instance order and a different random seed for each instance. 10 copies are used for 10 independent *training* trials, respectively, and one copy is used for *validation* where the best configurations found in the 10 training trials are compared. The best configuration identified by the validation phase is applied to the *testing* set of instances to assess the quality of the automatically trained configuration. The automatic configuration of CPLEX on MIPLIB instances done by Hutter et al. [115] has sped up over the use of default setting by a factor of 1.3 to 52. We followed the algorithmic parameter file for solving MILP by CPLEX listed at [114], which has a total of 74 parameters. Two parameters are removed from the list: `lpmethod` since no pure LP exists in our problem (LP relaxation in MIP is controlled by `MIP startalgorithm` or `MIP subalgorithm`); `NodeFileInd` is fixed to 3, such that it allows CPLEX to write the node

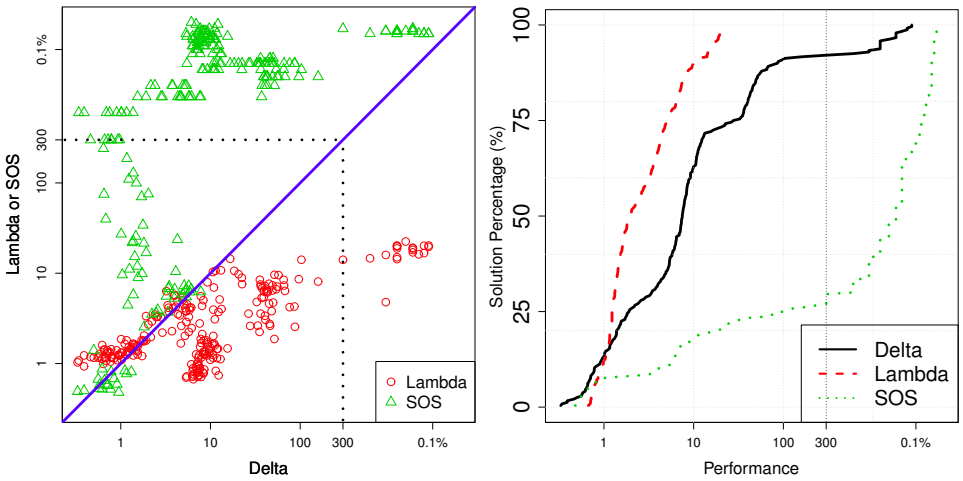


Figure D.1: The comparison of three piecewise linear formulations: Delta, Lambda, and SOS methods, solved by CPLEX with default setting.

files to hard disk rather than using a swap when the working memory (`WorkMem`) is exceeded.

There exists software for such automatic algorithm configuration tasks, e.g. ParamILS [118] is used in [115]. We have used JRace, which is a Java software for racing based configurators such as iterated racing [37] and post-selection [233, 242]. Each of the three piecewise linear formulations, Lambda, Delta, and SOS method, is trained separately. The cutoff time for the validation and testing phase is set to 300 seconds, and the cutoff time for the training phase is set to a much smaller value of 10 seconds. For the instances that are unsolved with a gap of $\delta\%$ after the cutoff time $\kappa = 10$ seconds, we modified the penalized average runtime (`PAR-10`) of $10 \cdot \kappa$ in [115] to `mPAR-10`: $10 \cdot \kappa \cdot (1 + (\delta - 0.01)/10)$, such that the unsolved runs during training can still compare with each other using their optimality gaps. A configuration budget of 3 000 evaluations is allowed for each training trial, which amounts to maximum around 8 hours per trial.

D.4.2 Formulation comparison with default setting

Figure D.1 compares the three piecewise linear formulations, Delta, Lambda, and SOS methods, using the CPLEX default setting with 12 threads and randomly generated seed on the testing set. The performance distribution plot on the left shows the runtime or the optimality gap if unsolved after 300 seconds. The Delta method is clearly better performing than the SOS, while it is also clearly outperformed by the Lambda method except for a few smallest instances. The runtime de-

Table D.2: The comparison of the default configuration and the automatically tuned configuration of CPLEX and the performance variability induced by using the two configurations. Number of solved instances, average runtime for solved instances, and the number of wins (out of 240) over the other configuration are shown in the comparison of the two configurations. The average speedup factor is also presented.

Method	default			tuned			avg.	var.coef.		$\frac{max}{min}$	
	#solved	avg.time	#win	#solved	avg.time	#win	spdup.	avg.	max.	avg.	max.
Lambda	240	4.24	84	240	3.49	151	1.21	0.20	0.83	1.25	2.42
SOS	64	38.72	39	67	18.87	170	2.05	0.61	1.81	2.83	20.62
Delta	220	38.43	14	240	2.49	226	15.37	1.19	1.92	9.24	51.99

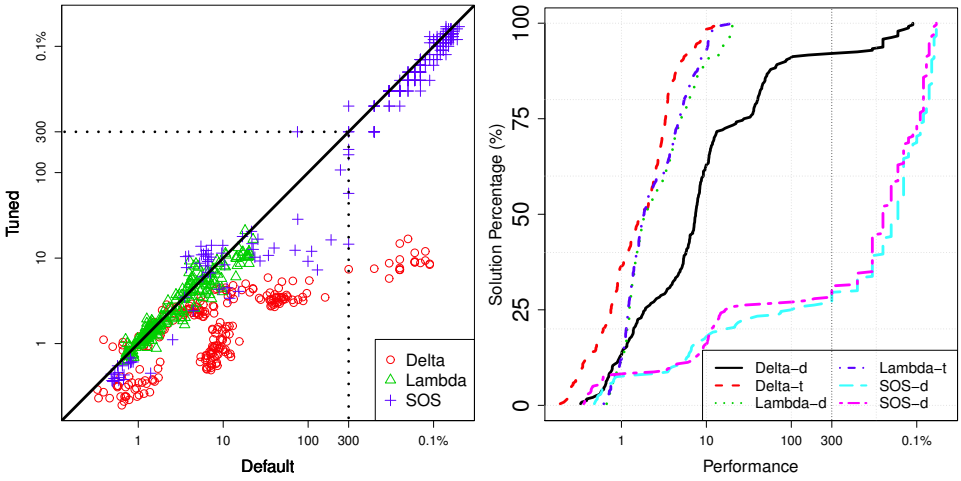


Figure D.2: The comparison of default and tuned parameter setting of CPLEX on the three piecewise linear formulations: Delta, Lambda, and SOS methods.

velopment plot on the right shows the percentage of instances that are solved within a runtime or reach a gap after cutoff. It agrees with the clear trend that the Lambda method is the overall best performing formulation of the three. In fact, the Lambda method is the only one that solves all the instances within 300 seconds. The Delta method solves 220 out of 240, and the SOS method solves only 64. The difference between the three is also statistically significant by the Wilcoxon's signed rank test or binomial test with $\alpha = 0.01$.

D.4.3 Automatically tuned setting versus default setting

The comparison of the default configuration and the automatically tuned configuration is listed in Table D.2. The tuned configuration

statistically significantly outperforms the CPLEX default setting for each of the three formulations. The tuned setting of the Lambda method speeds up in average 21% over the default setting, and performs better in 151 instances out of 240 while worse in 84. The tuned setting of the SOS method solves 3 more instances to optimality, and is more than twice as fast as the default setting in the solved instances, It also wins in 170 instances while loses in 39. The tuned setting of the Delta method solves all the 20 remaining unsolved problems, improves the default CPLEX setting by an average speedup factor of over 15, and performs better in 226 out of 240 instances. Different formulations benefit from automatic configuration differently. This is best illustrated in Figure D.2: the tuned configuration improves the Delta method much more than the SOS method which benefits more than the Lambda method. The performance variability measures due to the two different CPLEX configurations shown in the last four columns of Table D.2 is also drastically higher than the ones shown in Table D.1. The highest ratio is ca. 52 times, with an instance solved in 5.8 seconds by the tuned configuration while taking CPLEX with default setting 300 seconds with a gap of 0.03%.

D.4.4 Formulation comparison with tuned setting

With the automatically tuned configuration, the ranking of the three piecewise linear formulations shown in Figure D.3 looks quite different from Figure D.1 with default setting. The Delta method clearly and statistically significantly outperforms Lambda method, and the average speedup is 40%. This shows that the comparison of different MIP formulations heavily depends on the setting of the MIP solver. Comparing them using only the default setting may limit the conclusion to only a particular implementation of the MIP solver. Generalizing such conclusion may lead to misinterpretation. As a more reliable experimental setup to benchmark different formulations, an automatic configuration of the MIP solver should always be performed.

D.4.5 Further analysis on the tuned configuration

Since the tuned configuration drastically improves the Delta method over the default one for CPLEX, a further analysis of the tuned Delta setting is conducted. 38 out of the 72 parameters have been varied by the automatic configuration. We tried to vary from the default (tuned) configuration one of these 38 parameters to its tuned (default) value, respectively, evaluated them on the testing instances to assess their influence. The list of changed parameters and their de-

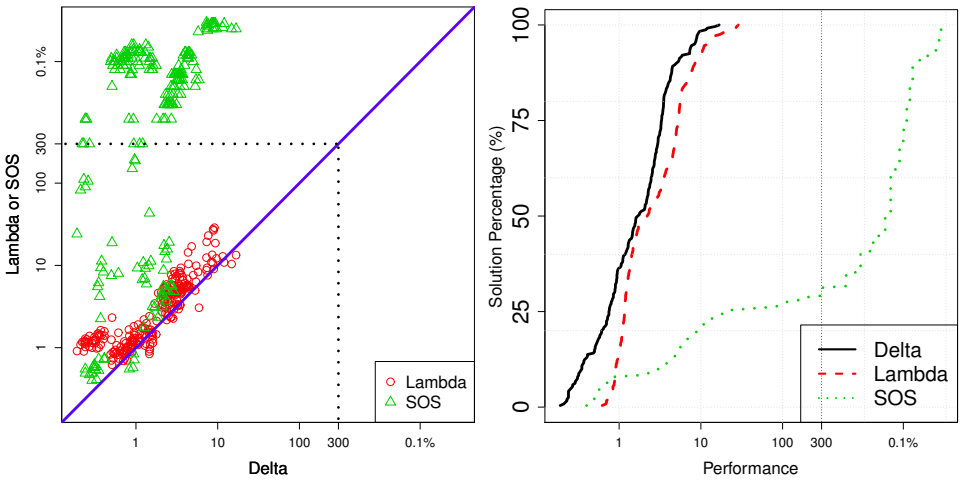


Figure D.3: The comparison of three piecewise linear formulations: Delta, Lambda, and SOS methods, solved by CPLEX with tuned configuration.

fault and tuned value can be found in `http://iridia.ulb.ac.be/~zyuan/downloads/deltaParam.xls`. There are 15 parameter changes from the default setting that can lead to a significant performance improvement (by Wilcoxon's signed rank test with $\alpha = 0.01$). The most influential one is to turn off the `preprocessing aggregator` (`on` by default). Varying only this parameter already solves the 20 by default unsolved instances, and has an average runtime of 11.4 seconds, which speeds up the default setting by a factor of 3.4. However, 3 single parameter variations from the default setting leads to significant performance deterioration. The most worsening one is changing the `MIP subalgorithm` from default value of `auto` (i.e., always select `dual simplex`) to `primal simplex`, which takes 2.2 times longer runtime than the default setting. On the other hand, there are 7 parameters, changing which from the tuned configuration towards default leads to statistically significantly worse performance. Again, the most influential is to turn the `preprocessing aggregator on` from the tuned configuration, which performs even worse than the default configuration for the hardest instances, as shown in the left of Figure D.4. An interesting observation is that the parameter `MIP subalgorithm` also belongs to one of the 7 most influential parameters, varying which from `primal simplex` to default `auto` statistically significantly worsens the performance (p-value 10^{-8}), and takes in average 5% more runtime, as shown in the right of Figure D.4. This shows that it can be misleading to analyze the influence of a single parameter to a given problem, or to set parameters in a one-factor-at-a-time fashion, since it also

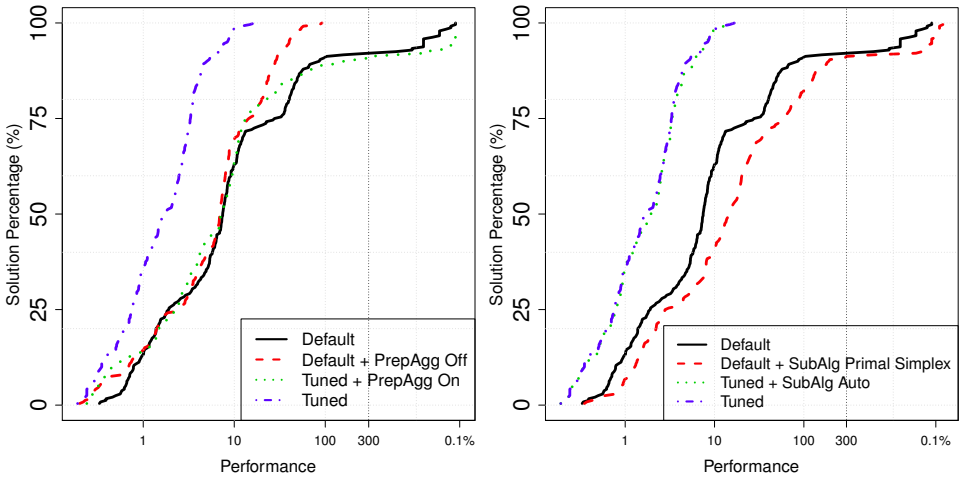


Figure D.4: Solver performance when changing one parameter `preprocessing aggregator` (left) or `MIP subalgorithm` (right) from either the default or the tuned configuration.

depends on other parameters. Such parameter correlation should be taken into account, and a sophisticated automatic configuration tool such as JRace should be a good choice.

D.5 Conclusions

MIP solvers are highly parameterized and randomized matheuristic algorithms. In this article, we analyze the performance variability of a MIP solver CPLEX, and apply an automatic configuration to CPLEX for comparing three classic piecewise linear formulations in the vertical flight planning problem. The performance variability in CPLEX due to random seed setting is certainly not negligible, the average variation coefficient ranges from 0.15 to 0.37. The performance variability due to different CPLEX parameter settings is even higher, and the formulation comparison depends heavily on the CPLEX setting. The experiments are conducted using different random seed and automatic configuration tool JRace. The automatically tuned configuration significantly improves the CPLEX default setting in all the three formulations by a factor of up to 15. Besides, the automatic configuration makes a theoretically superior but by default computationally inferior formulation (the Delta method) stand out as the best performing formulation.

Bibliography

[1] M. A. Abramson, C. Audet, J. W. Chrissis, and J. G. Walston. Mesh adaptive direct search algorithms for mixed variable optimization. *Optimization Letters*, 3(1):35–47, 2009.

[2] B. Adenso-Díaz and M. Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Operations Research*, 54(1):99–114, 2006.

[3] R. K. Ahuja, Ö. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1):75–102, 2002.

[4] P. J. Angeline. Adaptive and self-adaptive evolutionary computations. In M. Palaniswami et al., editors, *Computational intelligence: a dynamic systems perspective*. IEEE Press, 1995.

[5] C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of solvers. In I. P. Gent, editor, *Principles and Practice of Constraint Programming – CP 2009*, volume 5732 of *LNCS*, pages 142–157. Springer, Heidelberg, Germany, 2009.

[6] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton university press, 2006.

[7] C. Audet and J. Dennis. Analysis of generalized pattern searches. *SIAM Journal on Optimization*, 13:889–903, 2000.

[8] C. Audet and J. Dennis. Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on optimization*, 17(1):188–217, 2007.

[9] C. Audet and D. Orban. Finding optimal algorithmic parameters using derivative-free optimization. *SIAM Journal on Optimization*, 17:642, 2006.

[10] C. Audet and J. J. E. Dennis. Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on Optimization*, 17(1):188–217, 2006.

[11] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[12] A. Auger, N. Hansen, J. M. P. Zerpa, R. Ros, and M. Schoenauer. Experimental comparisons of derivative free optimization algorithms. In J. Vahrenhold, editor, *Experimental Algorithms, 8th International Symposium, SEA 2009*, volume 5526 of *LNCS*, pages 3–15. Springer, Heidelberg, Germany, 2009.

[13] A. Auger and N. Hansen. A restart cma evolution strategy with increasing population size. In *Proc. of IEEE Congress on Evolutionary Computation*, pages 1769–1776. IEEE, 2005.

[14] P. Balaprakash, M. Birattari, and T. Stützle. Improvement strategies for the F-Race algorithm: Sampling design and iterative refinement. In T. Bartz-Beielstein et al., editors, *HM'2007*, volume 4771 of *LNCS*, pages 108–122. Springer, Heidelberg, Germany, 2007.

[15] P. Balaprakash, M. Birattari, T. Stützle, and M. Dorigo. Adaptive sample size and importance sampling in estimation-based local search for the probabilistic traveling salesman problem. *European Journal of Operational Research*, 199(1):98–110, 2009.

[16] P. Balaprakash, M. Birattari, T. Stützle, Z. Yuan, and M. Dorigo. Ant colony optimization and estimation-based local search for the probabilistic traveling salesman problem. *Swarm Intelligence*, 3(3):223–242, 2009.

[17] T. Bartz-Beielstein, C. Lasarczyk, and M. Preuss. Sequential parameter optimization. In *IEEE Congress on Evolutionary Computation 2005*, pages 773–780. IEEE Press, 2005.

[18] T. Bartz-Beielstein, C. Lasarczyk, and M. Preuss. Sequential parameter optimization toolbox, manual version 0.5, september 2008, 2008.

[19] T. Bartz-Beielstein. *Experimental Research in Evolutionary Computation–The New Experimentalism*. Springer, Berlin, Germany, 2006.

[20] T. Bartz-Beielstein and S. Markon. Tuning search algorithms for real-world applications: A regression tree based approach. In

Proceedings of Congress on Evolutionary Computation (CEC), 2004, pages 1111–1118. IEEE, 2004.

[21] T. Bartz-Beielstein and M. Preuss. Considerations of budget allocation for sequential parameter optimization (spo). In *Workshop on Empirical Methods for the Analysis of Algorithms, Proceedings*, pages 35–40, 2006.

[22] R. Battiti, M. Brunato, and F. Mascia. *Reactive search and intelligent optimization*. Springer, New York, 2008.

[23] R. Battiti and G. Tecchiolli. The reactive tabu search. *ORSA journal on computing*, 6(2):126–140, 1994.

[24] E. L. M. Beale and J. A. Tomlin. Global Optimization Using Special Ordered Sets. *Mathematical Programming*, 10:52–69, 1976.

[25] S. Becker. Racing-Verfahren für Tourenplanungsprobleme. Diplomarbeit, Technische Universität Darmstadt, Darmstadt, Germany, 2004.

[26] S. Becker, J. Gottlieb, and T. Stützle. Applications of racing algorithms: An industrial perspective. In E.-G. Talbi et al., editors, *Artificial Evolution: 7th International Conference, Evolution Artificielle, EA 2005*, volume 3871 of *LNCS*, pages 271–283, Lille, France, 2005. Springer, Heidelberg, Germany.

[27] J. J. Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA Journal on computing*, 4(4):387–411, 1992.

[28] H.-G. Beyer and H.-P. Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1):3–52, 2002.

[29] P. Billingsley. *Probability and Measure*. John Wiley & Sons, New York, NY, USA, 1986.

[30] M. S. bin Hussin, T. Stützle, and M. Birattari. A study of stochastic local search algorithms for the quadratic assignment problems. In E. Ridge et al., editors, *Proceedings of SLS-DS 2007, Doctoral Symposium on Engineering Stochastic Local Search Algorithms*, pages 11–15, Brussels, Belgium, September 2007.

[31] M. Birattari. On the estimation of the expected performance of a metaheuristic on a class of instances. How many instances, how many runs? Technical Report TR/IRIDIA/2004-001, IRIDIA, Université Libre de Bruxelles, Belgium, IRIDIA, Université Libre de Bruxelles, Belgium, 2004.

[32] M. Birattari. *The Problem of Tuning Metaheuristics as seen from a Machine Learning Perspective*. PhD thesis, Université Libre de Bruxelles, 2004.

[33] M. Birattari. *Tuning Metaheuristics: A machine learning perspective*. Springer, Berlin, Germany, 2009.

[34] M. Birattari, P. Balaprakash, and M. Dorigo. The ACO/F-Race algorithm for combinatorial optimization under uncertainty. In K. F. Doerner et al., editors, *Metaheuristics - Progress in Complex Systems Optimization*, Operations Research/Computer Science Interfaces Series, pages 189–203. Springer Verlag, Berlin, Germany, 2007.

[35] M. Birattari, M. Gagliolo, S. bin Hussin, T. Stützle, and Z. Yuan. Discussion in IRIDIA coffee room, October, 2008.

[36] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In W. B. Langdon et al., editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18. Morgan Kaufmann Publishers, San Francisco, CA, 2002.

[37] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. F-Race and iterated F-Race: An overview. In T. Bartz-Beielstein et al., editors, *Experimental Methods for the Analysis of Optimization Algorithms*, pages 311–336. Springer, Berlin, Germany, 2010.

[38] C. Blum and K. Socha. Training feed-forward neural networks with ant colony optimization: An application to pattern classification. In N. Nedjah et al., editors, *Proceedings of Fifth International Conference on Hybrid Intelligent Systems (HIS'05)*, pages 233–238, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[39] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm intelligence: from natural to artificial systems*. Oxford university press, 1999.

[40] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[41] R. E. Burkard. Quadratic assignment problems. In P. Pardalos et al., editors, *Handbook of Combinatorial Optimization, 2nd edition*, pages 2741–2814. Springer, New York, USA, 2013.

[42] R. E. Burkard, S. E. Karisch, and F. Rendl. QAPLIB -— a quadratic assignment problem library. *Journal of Global Optimization*, 10(4):391–403, 1997.

[43] K. Burnham and D. Anderson. *Model selection and multimodel inference: a practical information-theoretic approach*. Springer, 2002.

[44] O. Caelen and G. Bontempi. How to allocate a restricted budget of leave-one-out assessments for effective model selection in machine learning: a comparison of state-of-the-art techniques. In K. Verbeeck et al., editors, *Proceedings of the 17th Belgian-Dutch Conference on Artificial Intelligence (BNAIC'05)*, pages 51–58, Brussels, Belgium, 2005.

[45] V. Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[46] I. Charon and O. Hudry. The noising method: a new method for combinatorial optimization. *Operations Research Letters*, 14(3):133–137, 1993.

[47] C.-H. Chen and L. H. Lee. *Stochastic simulation optimization: an optimal computing budget allocation*. World scientific, 2011.

[48] M. Chiarandini. *Stochastic local search methods for highly constrained combinatorial optimisation problems*. PhD thesis, Technische Universität Darmstadt, Darmstadt, Germany, 2005.

[49] M. Chiarandini, M. Birattari, K. Socha, and O. Rossi-Doria. An effective hybrid algorithm for university course timetabling. *Journal of Scheduling*, 9(5):403–432, 2006.

[50] M. Chiarandini and T. Stützle. Experimental evaluation of course timetabling algorithms. Technical Report AIDA-02-05, FG Intellektik, FB Informatik, Technische Universität Darmstadt, Darmstadt, Germany, 2002.

[51] M. Chiarandini and T. Stützle. Stochastic local search algorithms for graph set t-colouring and frequency assignment. *Constraints*, 12(3):371–403, 2007.

[52] M. Clerc and J. Kennedy. The particle swarm–explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[53] A. R. Conn, K. Scheinberg, and L. N. Vicente. *Introduction to derivative-free optimization*. SIAM, 2009.

[54] W. J. Conover. *Practical Nonparametric Statistics*. John Wiley & Sons, New York, NY, USA, 1999.

[55] D. W. Corne, M. J. Oates, and D. B. Kell. On fitness distributions and expected fitness gain of mutation rates in parallel evolutionary algorithms. In *Proc. of PPSN VII*, pages 132–141. Springer, 2002.

[56] E. Danna. Performance variability in mixed integer programming. In *Presentation at Workshop on Mixed Integer Programming*, 2008.

[57] G. B. Dantzig. On the significance of solving linear programming problems with some integer variables. *Econometrica*, 28(1):30 – 44, 1960.

[58] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

[59] L. Davis. Applying adaptive algorithms to epistatic domains. In *Proc. of IJCAI*, pages 162–164, 1985.

[60] L. Davis. Adapting operator probabilities in genetic algorithms. In *Proceedings of 3rd International Conference on Genetic Algorithms and their*, pages 61–69. Morgan Kaufmann, 1989.

[61] A. Dean and D. Voss. *Design and Analysis of Experiments*. Springer, New York, NY, 1999.

[62] M. L. den Besten. *Simple Metaheuristics for Scheduling. An empirical investigation into the application of iterated local search to deterministic scheduling problems with tardiness penalities*. PhD thesis, Technische Universität Darmstadt, 2004.

[63] G. Di Caro and M. Dorigo. AntNet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research*, 9:317–365, 1998.

[64] L. Di Gaspero and A. Roli. Stochastic local search for large-scale instances of the haplotype inference problem by pure parsimony. *Journal of Algorithms*, 63(1-3):55–69, 2008.

[65] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[66] M. Dorigo and L. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[67] M. Dorigo and L. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[68] M. Dorigo, V. Maniezzo, and A. Colorni. Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics – Part B*, 26(1):29–41, 1996.

[69] M. Dorigo. *Optimization, learning and natural algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[70] M. Dorigo, M. A. Montes de Oca, and A. P. Engelbrecht. Particle swarm optimization. *Scholarpedia*, 3(11):1486, 2008.

[71] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, Cambridge, MA, 2004.

[72] M. Dorigo and T. Stützle. Ant colony optimization: overview and recent advances. In *Handbook of metaheuristics*, pages 227–263. Springer, 2010.

[73] A. E. Eiben, Z. Michalewicz, M. Schoenauer, and J. E. Smith. Parameter Control in Evolutionary Algorithms. In F. G. Lobo, C. F. Lima, and Z. Michalewicz, editors, *Parameter Setting in Evolutionary Algorithms*, Studies in Computational Intelligence Series, pages 19–46. Springer, Berlin, Germany, 2007.

[74] T. A. Feo and M. G. Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133, 1995.

[75] A. Fialho. Adaptive operator selection for optimization. *PhD thesis, Ecole Doctorale d'Informatique, Université Paris-Sud, Paris*, 2010.

[76] A. Fialho, L. Costa, M. Schoenauer, and M. Sebag. Dynamic multi-armed bandits and extreme value-based rewards for adaptive operator selection in evolutionary algorithms. In T. Stützle et al., editors, *Proc. of LION*, pages 176–190. Springer-Verlag, Berlin, Germany, 2009.

[77] Á. Fialho, L. Da Costa, M. Schoenauer, and M. Sebag. Extreme value based adaptive operator selection. In *Proceedings of PPSN X*, pages 175–184. Springer, 2008.

[78] Á. Fialho, M. Schoenauer, and M. Sebag. Toward comparison-based adaptive operator selection. In *Proceedings of GECCO*, pages 767–774. ACM, 2010.

[79] M. Fischetti and M. Monaci. On the role of randomness in exact tree search methods. In *Presentation at Matheuristics*, 2012.

[80] M. Fischetti and A. Lodi. Heuristics in mixed integer programming. *Wiley Encyclopedia of Operations Research and Management Science*, 2011.

[81] M. Fischetti and M. Monaci. Exploiting erraticism in search. *Operations Research*, 62(1):114–122, 2014.

[82] G. Francesca, P. Pellegrini, T. Stützle, and M. Birattari. Off-line and on-line tuning: A study on operator selection for a memetic algorithm applied to the QAP. In P. Merz and J.-K. Hao, editors, *Proc. of EvoCOP*, volume 6622 of *Lecture Notes in Computer Science*, pages 203–214. Springer, 2011.

[83] M. Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32(200):675–701, 1937.

[84] M. Fu, editor. *Handbook of simulation optimization*. "International Series in Operations Research & Management Science".

[85] A. Fuegenschuh, H. Homfeld, A. Huck, A. Martin, and Z. Yuan. Scheduling locomotives and car transfers in freight transport. *Transportation Science*, 42(4):478–491, 2008.

[86] A. Fügenschuh, C. Hayn, and D. Michaels. Mixed-Integer Linear Methods for Layout-Optimization of Screening Systems in Recovered Paper Production. *Optimization and Engineering*, 15(2):533–573, 2014.

[87] A. Fügenschuh. Parametrized greedy heuristics in theory and practice. In M. Blesa et al., editors, *Proceedings of Second International Workshop on Hybrid Metaheuristics*, volume 3636 of *Lecture Notes in Computer Science*, pages 21–31. Springer Verlag, Berlin, Germany, 2005.

[88] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman New York, 1979.

[89] L. D. Gaspero, G. di Tollo, A. Roli, and A. Schaerf. Hybrid local search for constrained financial portfolio selection problems. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 4510 of *LNCS*, pages 44–58. Springer, Heidelberg, Germany, 2007.

[90] B. Geißler, A. Martin, A. Morsi, and L. Schewe. Using piecewise linear functions for solving MINLPs. In *Mixed Integer Nonlinear Programming*, pages 287–314. Springer, 2012.

[91] M. Gendreau and J.-Y. Potvin. *Handbook of metaheuristics*, volume 2. Springer, 2010.

[92] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

[93] F. Glover. Tabu search – part I. *ORSA Journal on computing*, 1(3):190–206, 1989.

[94] F. Glover. Tabu search —- part II. *ORSA Journal on computing*, 2(1):4–32, 1990.

[95] D. E. Goldberg. Genetic algorithms and rule learning in dynamic system control. In *International Conference on Genetic Algorithms and Their Applications*, pages 8–15. Morgan Kaufmann Publishers Inc., 1985.

[96] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, Boston, MA, USA, 1989.

[97] Y. Hamadi, E. Monfroy, and F. Saubion, editors. *Autonomous Search*. Springer, Berlin, Germany, 2007.

[98] N. Hansen. The CMA evolution strategy: a comparing review. In J. Lozano et al., editors, *Towards a new evolutionary computation*, volume 192 of *Studies in Fuzziness and Soft Computing*, pages 75–102. Springer, Berlin, Germany, 2006.

[99] N. Hansen, S. Finck, R. Ros, and A. Auger. Real-parameter black-box optimization benchmarking 2009: Noisy functions definitions. Technical Report RR-6869, INRIA, 2009.

[100] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[101] N. Hansen. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

[102] N. Hansen, S. Finck, R. Ros, and A. Auger. Real-parameter black-box optimization benchmarking 2009: Noiseless functions definitions. Technical Report RR-6829, INRIA, 2009.

[103] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 312–317. IEEE, 1996.

[104] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[105] P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44(4):279–303, 1990.

[106] P. Hansen and N. Mladenović. An introduction to variable neighborhood search. In *Meta-heuristics*, pages 433–458. Springer, 1999.

[107] H. H. Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012.

[108] H. H. Hoos and T. Stützle. *Stochastic local search: Foundations & applications*. Elsevier, 2004.

[109] D. Huang, T. T. Allen, W. I. Notz, and N. Zeng. Global optimization of stochastic black-box systems via sequential kriging meta-models. *Journal of global optimization*, 34(3):441–466, 2006.

[110] F. Hutter. Software ParamILS. `http://www.cs.ubc.ca/labs/beta/Projects/ParamILS`, 2010.

[111] F. Hutter, T. Bartz-Beielstein, H. H. Hoos, K. Leyton-Brown, and K. Murphy. Sequential model-based parameter optimisation: an experimental investigation of automated and interactive approaches. In T. Bartz-Beielstein et al., editors, *Empirical Methods for the Analysis of Optimization Algorithms*, pages 363–414. Springer, Berlin, Germany, 2010.

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

[113] F. Hutter, H. Hoos, and K. Leyton-Brown. Tradeoffs in the empirical evaluation of competing algorithm designs. *Annals of Mathematics and Artificial Intelligence*, 60(1):65–89, 2010.

[114] F. Hutter. Automated Configuration of MIP solvers. `http://www.cs.ubc.ca/labs/beta/Projects/MIP-Config`. Last accessed: 2016-05-04.

[115] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In *Proc. of CPAIOR*, volume 9335 of *LNCS*, pages 186–202. Springer, 2010.

[116] F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. Murphy. Time-bounded sequential parameter optimization. In *Proc. of Learning and intelligent optimization*, pages 281–298. Springer, 2010.

[117] F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. P. Murphy. An experimental investigation of model-based parameter optimisation: SPO and beyond. In F. Rothlauf, editor, *Genetic and Evolu-*

tionary Computation Conference, GECCO 2009, pages 271–278. ACM press, New York, 2009.

[118] 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, 2009.

[119] F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In R. C. Holte and A. Howe, editors, *Proceedings of the 22nd conference on artificial intelligence (AAAI)*, pages 1152–1157. AAAI Press, 2007.

[120] IBM ILOG CPLEX. 12.6 user's manual. 2014.

[121] D. S. Johnson, L. A. McGeoch, C. Rego, and F. Glover. 8th DIMACS implementation challenge. `http://www.research.att.com/~dsj/chtsp/`, 2001.

[122] D. S. Johnson. A theoretician's guide to the experimental analysis of algorithms. *Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS implementation challenges*, pages 215–250, 2002.

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

[124] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In L. J. Eshelman, editor, *Proc. of the 6th International Conference on Genetic Algorithms*, pages 184–192. Morgan Kaufmann, San Francisco, CA, 1995.

[125] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. Isac – instance-specific algorithm configuration. In *Proceedings of 19th European Conference on Artificial Intelligence (ECAI)*, pages 751–756, 2010.

[126] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: An introduction to cluster analysis*. Wiley, New York, 1990.

[127] J. Kennedy and R. Mendes. Neighborhood topologies in fully informed and best-of-neighborhood particle swarms. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 36(4):515–519, 2006.

[128] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proc. of IEEE International Conference on Neural Networks*, pages 1942–1948. IEEE Press, Piscataway, NJ, 1995.

[129] J. Kennedy, R. Eberhart, and Y. Shi. *Swarm Intelligence*. Morgan Kaufmann, San Francisco, CA, 2001.

[130] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

[131] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[132] J. Kleijnen. Analyzing simulation experiments with common random numbers. *Management science*, 34(1):65–74, 1988.

[133] J. Knowles and D. Corne. Instance generators and test suites for the multiobjective quadratic assignment problem. In *Evolutionary Multi-criterion Optimization*, pages 295–310. Springer, 2003.

[134] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, et al. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011.

[135] T. G. Kolda, R. M. Lewis, and V. Torczon. Optimization by direct search: New perspectives on some classical and modern methods. *SIAM review*, 45(3):385–482, 2003.

[136] T. C. Koopmans and M. Beckmann. Assignment problems and the location of economic activities. *Econometrica: journal of the Econometric Society*, pages 53–76, 1957.

[137] L. Kotthoff. Algorithm selection for combinatorial search problems: A survey. *AI Magazine*, 2014.

[138] E. Krempser, Á. Fialho, and H. J. C. Barbosa. Adaptive operator selection at the hyper-level. In C. Coello et al., editors, *Proc. of PPSN*, volume 7492 of *Lecture Notes in Computer Science*, pages 378–387. Springer, 2012.

[139] A. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw Hill, NewYork, NJ, USA, 1991.

[140] E. Lawler, J. Lenstra, A. Rinnooy Kan, and D. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, Chichester, 1985.

[141] R. Lenne, C. Solnon, T. Stützle, E. Tannier, and M. Birattari. Effective stochastic local search algorithms for the genomic median problem. In E. Ridge et al., editors, *Proceedings of SLS-DS 2007, Doctoral Symposium on Engineering Stochastic Local Search Algorithms*, pages 1–5, Brussels, Belgium, September 2007.

[142] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4):22, 2009.

[143] T. Liao, K. Socha, M. A. M. de Oca, T. Stützle, and M. Dorigo. Ant colony optimization for mixed-variable optimization problems. *IEEE Transactions on Evolutionary Computation*, 18(4):503–518, 2014.

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

[145] Lindawati, H. Lau, and D. Lo. Clustering of search trajectory and its application to parameter tuning. *Journal of the Operational Research Society*, 64(12):1742–1752, 2013.

[146] Lindawati, Z. Yuan, H. C. Lau, and F. Zhu. Automated parameter tuning framework for heterogeneous and large instances: Case study in quadratic assignment problem. In G. Nicosia and P. Pardalos, editors, *Proc. of LION7*, volume 7997 of *Lecture Notes in Computer Science*, pages 423—-437. Springer-Verlag, Berlin, Germany, 2013.

[147] P. Lindroth and M. Patriksson. *Pure Categorical Optimization: A Global Descent Approach*. Department of Mathematical Sciences, Division of Mathematics, Chalmers University of Technology, University of Gothenburg, 2011.

[148] F. Lobo, C. F. Lima, and Z. Michalewicz, editors. *Parameter Setting in Evolutionary Algorithms*. Springer, Berlin, Germany, 2007.

[149] A. Lodi and A. Tramontani. Performance variability in mixed-integer programming. *TutORials in Operations Research*, pages 1–12, 2013.

[150] M. López-Ibánez, T. Stützle, and M. Dorigo. Ant colony optimization: A component-wise overview. (TR/IRIDIA/2015-006), 2015.

[151] M. López-Ibánez, J. Dubois-Lacoste, T. Stützle, and M. Birattari. The **irace** package: Iterated racing for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, January 2011.

[152] S. N. Lophaven, H. B. Nielsen, and J. Søndergaard. DACE – A Matlab Kriging toolbox, version 2.0. Technical Report IMM-REP-2002-12, Informatics and Mathematical Modelling (IMM), Technical University of Denmark, Copenhagen, Denmark, 2002.

[153] H. Lourenço, O. Martin, and T. Stützle. Iterated local search. *Handbook of metaheuristics*, pages 320–353, 2003.

[154] H. R. Lourenço, O. C. Martin, and T. Stützle. Iterated local search: Framework and applications. In *Handbook of Metaheuristics*, pages 363–397. Springer, 2010.

[155] H. M. Markowitz and A. S. Manne. On the solution of discrete programming problems. *Econometrica*, 25(1):84 – 110, 1957.

[156] O. Maron and A. W. Moore. Hoeffding races: Accelerating model selection search for classification and function approximation. In J. D. Cowan et al., editors, *Advances in Neural Information Processing Systems*, volume 6, pages 59–66. Morgan Kaufmann Publishers, San Francisco, USA, 1994.

[157] A. Martin, M. Möller, and S. Moritz. Mixed integer models for the stationary case of gas network optimization. *Mathematical programming*, 105(2-3):563–582, 2006.

[158] O. Martin, S. W. Otto, and E. W. Felten. Large-step markov chains for the traveling salesman problem. *Complex Systems*, 5:299–326, 1991.

[159] J. R. Martins and A. B. Lambe. Multidisciplinary design optimization: a survey of architectures. *AIAA journal*, 51(9):2049–2075, 2013.

[160] F. Mascia, P. Pellegrini, M. Birattari, and T. Stützle. An analysis of parameter adaptation in reactive tabu search. *International Transactions in Operational Research*, 21(1):127–152, 2013.

[161] D. C. Matthews, A. M. Sutton, D. Hains, and L. D. Whitley. Improved robustness through population variance in ant colony optimization. In T. Stützle et al., editors, *Proceedings of Engineering Stochastic Local Search Algorithms (SLS)*, volume 5752 of *Lecture Notes in Computer Science*, pages 145–149. Springer, 2009.

[162] C. McGeoch. Analyzing algorithms by simulation: variance reduction techniques and simulation speedups. *ACM Computing Surveys*, 24(2):195–212, 1992.

[163] C. C. McGeoch. Feature article-toward an experimental method for algorithm simulation. *INFORMS Journal on Computing*, 8(1):1–15, 1996.

[164] O. Mengshoel. Understanding the role of noise in stochastic local search: Analysis and experiments. *Artificial Intelligence*, 172(8-9):955–990, 2008.

[165] P. Merz and B. Freisleben. Fitness landscape analysis and memetic algorithms for the quadratic assignment problem. *IEEE Transaction on Evolutionary Computation*, 4(4):337–352, 2000.

[166] P. Merz and B. Freisleben. A genetic local search approach to the quadratic assignment problem. In *Proceedings of the 7th International Conference on Genetic Algorithms*, pages 465–472. Morgan Kaufmann, 1997.

[167] P. Merz and B. Freisleben. A comparison of memetic algorithms, tabu search, and ant colonies for the quadratic assignment problem. In *Proceedings of Congress on Evolutionary Computation*, pages 2063–2070. IEEE Press, 1999.

[168] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[169] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

[170] M. A. Montes de Oca, K. Van den Enden, and T. Stützle. Incremental particle swarm-guided local search for continuous optimization. In M. J. Blesa et al., editors, *LNCS 5296. Proceedings of the International Workshop on Hybrid Metaheuristics. HM 2008*, pages 72–86. Springer, Berlin,Germany, 2008.

[171] D. C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, New York, NY, USA, fifth edition, 2000.

[172] J. More and S. Wild. Benchmarking derivative-free optimization algorithms. *SIAM Journal on Optimization*, 20(1):172–191, 2009.

[173] P. Moscato. Memetic algorithms: A short introduction. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 219–234. McGraw Hill, London, 1999.

[174] V. Nannen and A. E. Eiben. Relevance estimation and value calibration of evolutionary algorithm parameters. In *Proc. of IJCAI 2007*, pages 975–980. AAAI Press/IJCAI, Menlo Park, CA, 2007.

[175] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[176] C. Neumüller, A. Scheibenpflug, S. Wagner, A. Beham, and M. Affenzeller. Large scale parameter meta-optimization of metaheuristic optimization algorithms with heuristiclab hive. In *Proceedings of the 8th spanish congress on Metaheuristics, Evolutionary and Bioinspired Algorithms (MAEB)*, page 8 pages. 2012.

[177] K. Ng, A. Gunawan, and K. Poh. A hybrid algorithm for the quadratic assignment problem. In *International Conference on Scientific Computing*, pages 14–17, 2008.

[178] S. Nouyan. *Teamwork in a Swarm of Robots – An Experiment in Search and Retrieval*. PhD thesis, Université Libre de Bruxelles, Brussels, Belgium, Brussels, Belgium, 2008.

[179] S. Nouyan, A. Campo, and M. Dorigo. Path formation in a robot swarm. *Swarm Intelligence*, 2(1):1–23, 2008.

[180] B. M. Ombuki, M. Nakamura, and K. Onaga. An evolutionary scheduling scheme based on gkga approach to the job shop scheduling problem. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 81(6):1063–1071, 1998.

[181] M. Padberg. Approximating separable nonlinear functions via mixed zero-one programs. *Operations Research Letters*, 27(1):1–5, 2000.

[182] A. Papoulis. *Probability, Random Variables, and Stochastic Processes*. McGraw Hill, NewYork, NJ, USA, third edition, 1991.

[183] P. M. Pardalos and H. Wolkowicz, editors. *Quadratic Assignment and Related Problems*. DIMACS Series. American Mathematical Society, 1994.

[184] P. Pellegrini. Application of two nearest neighbor approaches to a rich vehicle routing problem. Technical Report TR/IRIDIA/2005-15, IRIDIA, Université Libre de Bruxelles, Belgium, 2005.

[185] P. Pellegrini, T. Stützle, and M. Birattari. A critical analysis of parameter adaptation in ant colony optimization. *Swarm Intelligence*, 6(1):23–48, 2012.

[186] C. Philemotte and H. Bersini. The gestalt heuristic: learning the right level of abstraction to better search the optima. Technical Report TR/IRIDIA/2008-021, IRIDIA, Université Libre de Bruxelles, Belgium, 2008.

[187] R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization. An overview. *Swarm Intelligence*, 1(1):33–57, 2007.

[188] M. J. D. Powell. The NEWUOA software for unconstrained optimization. In *Large-Scale Nonlinear Optimization*, volume 83 of *Nonconvex Optimization and Its Applications*, pages 255–297. Springer, Berlin, Germany, 2006.

[189] M. J. D. Powell. The BOBYQA algorithm for bound constrained optimization without derivatives. Technical Report NA2009/06, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, Cambridge, UK, 2009.

[190] M. J. Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The computer journal*, 7(2):155–162, 1964.

[191] R. C. Prim. Shortest connection networks and some generalizations. *Bell Labs Technical Journal*, 36(6):1389–1401, 1957.

[192] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologishen Evolution*. PhD thesis, Technical University of Berlin, Germany, 1971.

[193] G. Reinelt. *The traveling salesman: computational solutions for TSP applications*. Springer-Verlag, 1994.

[194] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[195] M. Risler, M. Chiarandini, L. Paquete, T. Schiavinotto, and T. Stützle. An algorithm for the car sequencing problem of the ROADEF 2005 challenge. Technical Report AIDA–04–06, FG Intellektik, TU Darmstadt, Darmstadt, Germany, 2004.

[196] O. Rossi-Doria, M. Sampels, M. Birattari, M. Chiarandini, M. Dorigo, L. M. Gambardella, J. Knowles, M. Manfrin, M. Mastrolilli, B. Paechter, L. Paquete, and T. Stützle. A comparison of the performance of different metaheuristics on the timetabling problem. In E. Burke et al., editors, *Practice and Theory of Automated Timetabling IV*, volume 2740 of *LNCS*, pages 329–351. Springer, Heidelberg, Germany, 2003.

[197] R. Ruiz and T. Stützle. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033–2049, 2007.

[198] J. Sacks, W. J. Welch, T. J. Mitchell, and H. P. Wynn. Design and analysis of computer experiments. *Statistical science*, pages 409–423, 1989.

[199] S. Sahni and T. Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23(3):555–565, 1976.

[200] T. Schiavinotto and T. Stützle. The linear ordering problem: Instances, search space analysis and algorithms. *Journal of Mathematical Modelling and Algorithms*, 3(4):367–402, 2004.

[201] M. Schneider and H. H. Hoos. Quantifying homogeneity of instance sets for algorithm configuration. In *Learning and Intelligent Optimization*, pages 190–204. Springer, 2012.

[202] H.-P. Schwefel. *Evolutionsstrategie und Numerische Optimierung*. PhD thesis, Technical University of Berlin, Germany, 1975.

[203] D. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, Boca Raton, FL, USA, second edition, 2000.

[204] S. Siegel and N. J. Castellan, Jr. *Non Parametric Statistics for the Behavioral Sciences*. McGraw Hill, NewYork, NJ, USA, second edition, 1988.

[205] K. Smith-Miles and L. Lopes. Measuring instance difficulty for combinatorial optimization problems. *Computers & Operations Research*, 39(5):875–889, 2012.

[206] K. Socha and C. Blum. An ant colony optimization algorithm for continuous optimization: application to feed-forward neural network training. *Neural Computing and Applications*, 16(3):235–247, 2007.

[207] K. Socha and M. Dorigo. Ant colony optimization for continuous domains. *European journal of operational research*, 185(3):1155–1173, 2008.

[208] T. A. Sriver, J. W. Chrissis, and M. A. Abramson. Pattern search ranking and selection algorithms for mixed variable simulation-based optimization. *European journal of operational research*, 198(3):878–890, 2009.

[209] O. Steinmann, A. Strohmaier, and T. Stützle. Tabu search vs. random walk. In G. Brewka, C. Habel, and B. Nebel, editors, *KI-97: Advances in Artificial Intelligence*, volume 1303 of *LNAI*, pages 337–348. Springer, Heidelberg, Germany, 1997.

[210] J. Stoer and R. Bulirsch. *Introduction to numerical analysis*. Springer Science & Business Media, 2013.

[211] R. Storn and K. Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

[212] T. Stützle. Software ACOTSP. `http://iridia.ulb.ac.be/~mdorigo/ACO/aco-code/public-software.html`, 2002.

[213] T. Stützle and H. H. Hoos. $\mathcal{MAX}$–$\mathcal{MIN}$ ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[214] T. Stützle. *Local Search Algorithms for Combinatorial Problems: Analysis, Improvements, and New Applications*, volume 220 of *DISKI*. Infix, Sankt Augustin, Germany, 1999.

[215] T. Stützle. Iterated local search for the quadratic assignment problem. *European Journal of Operational Research*, 174(3):1519–1539, 2006.

[216] T. Stützle and S. Fernandes. New benchmark instances for the QAP and the experimental analysis of algorithms. In *Proc. of EvoCOP*, volume 3004 of *Lecture Notes in Computer Science*, pages 199–209. Springer, 2004.

[217] T. Stützle and H. H. Hoos. MAX-MIN ant system and local search for combinatorial optimization problems: Towards adaptive tools for combinatorial global optimization. In S. Voss et al., editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 313–329. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999.

[218] T. Stützle, M. López-Ibánez, P. Pellegrini, M. Maur, M. M. de Oca, M. Birattari, and M. Dorigo. Parameter adaptation in ant colony optimization. In Y. Hamadi et al., editors, *Autonomous Search*, pages 191–215. Springer, 2012.

[219] J. Styles, H. H. Hoos, and M. Müller. Automatically configuring algorithms for scaling performance. In *Proceedings of Learning and Intelligent Optimization Conference (LION6)*, pages 205–219. Springer, 2012.

[220] É. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel computing*, 17(4):443–455, 1991.

[221] D. Thierens. An adaptive pursuit strategy for allocating operator probabilities. In *Proceedings of conference on genetic and evolutionary computation (GECCO)*, pages 1539–1546. ACM, 2005.

[222] V. Torczon. On the convergence of pattern search algorithms. *SIAM Journal on Optimization*, 7(1):1–25, 1997.

[223] M. Ventresca, B. Ombuki-Berman, and A. Runka. Predicting genetic algorithm performance on the vehicle routing problem using information theoretic landscape measures. In M. Middendorf and C. Blum, editors, *Proceedings of Evolutionary Computation in Combinatorial Optimization (EvoCOP)*, volume 7832 of *Lecture Notes in Computer Science*, pages 214–225. Springer Berlin Heidelberg, 2013.

[224] J. P. Vielma, S. Ahmed, and G. Nemhauser. Mixed-integer models for nonseparable piecewise-linear optimization: unifying framework and extensions. *Operations research*, 58(2):303–315, 2010.

[225] W. Wiesemann and T. Stützle. Iterated ants: An experimental study for the quadratic assignment problem. In M. Dorigo et al., editors, *Proceedings of 5th International Workshop on Ant Colony Optimization and Swarm Intelligence*, volume 4150 of *Lecture Notes in Computer Science*, pages 179–190. Springer, 2006.

[226] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, pages 80–83, 1945.

[227] J. Wilson. Variance reduction techiniques for digital simulation. *American Journal of Mathematical and Management Sciences*, 4(3):227–312, 1984.

[228] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *J. Artif. Intell. Res.(JAIR)*, 32:565–606, 2008.

[229] L. Xu, H. Hoos, and K. Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*, pages 210–216, 2010.

[230] B. Yuan and M. Gallagher. Statistical racing techniques for improved empirical evaluation of evolutionary algorithms. In X. Yao et al., editors, *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *LNCS*, pages 172–181. Springer, Heidelberg, Germany, 2004.

[231] B. Yuan and M. Gallagher. A hybrid approach to parameter tuning in genetic algorithms. In *Proceedings of the IEEE Congress in Evolutionary Computation (CEC'05)*, volume 2, pages 1096–1103. IEEE Press, Piscataway, NJ, 2005.

[232] B. Yuan and M. Gallagher. Combining Meta-EAs and racing for difficult ea parameter tuning tasks. In *Parameter Setting in Evolutionary Algorithms*, volume 54 of *Studies in Computational Intelligence*, pages 121–142. Springer, Heidelberg, Germany, 2007.

[233] Z. Yuan, M. M. de Oca, M. Birattari, and T. Stützle. Continuous optimization algorithms for tuning real and integer parameters of swarm intelligence algorithms. *Swarm Intelligence*, 6(1):49–75, 2012.

[234] Z. Yuan, A. Fügenschuh, H. Homfeld, P. Balaprakash, T. Stützle, and M. Schoch. Iterated greedy algorithms for a real-world cyclic train scheduling problem. In M. J. Blesa et al., editors, *Hybrid Metaheuristics, 5th International Workshop, HM 2008*, volume 5296 of *LNCS*, pages 102–116. Springer, Heidelberg, Germany, 2008.

[235] Z. Yuan, M. Montes de Oca, M. Birattari, and T. Stützle. Modern continuous optimization algorithms for tuning real and integer algorithm parameters. In M. Dorigo et al., editors, *Proceedings of ANTS 2010, the Seventh International Conference on Swarm Intelligence*, volume 6234 of *LNCS*, pages 204–215. Springer, Heidelberg, Germany, 2010.

[236] Z. Yuan, T. Stützle, and M. Birattari. MADS/F-Race: mesh adaptive direct search meets F-race. In M. Ali et al., editors, *Proceedings of IEA-AIE 2010*, volume 6096 of *LNAI*, pages 41–50. Springer Verlag, Heidelberg, Germany, 2010.

[237] Z. Yuan. Automatic configuration of mip solver: Case study in vertical flight planning. *Proceedings of Matheuristics*, pages 48–57, 2016.

[238] Z. Yuan, L. Amaya Moreno, A. Fügenschuh, A. Kaier, and S. Schlobach. Discrete speed in vertical flight planning. In F. Corman et al., editors, *Proceedings of International Conference on Computational Logistics*, volume 9335 of *Lecture Notes in Computer Science*, pages 734–749. Springer, 2015.

[239] Z. Yuan, L. Amaya Moreno, A. Maolaaisha, A. Fügenschuh, A. Kaier, and S. Schlobach. Mixed integer second-order cone programming for the horizontal and vertical free-flight planning problem. Technical report, AMOS#21, Applied Mathematical Optimization Series, Helmut Schmidt University, Hamburg, Germany, 2015.

[240] Z. Yuan, A. Fügenschuh, A. Kaier, and S. Schlobach. Variable speed in vertical flight planning. In *Operations Research Proceedings*, pages 635–641. Springer, 2014.

[241] Z. Yuan, S. D. Handoko, D. T. Nguyen, and H. C. Lau. An empirical study of off-line configuration and on-line adaptation in operator selection. In P. M. Pardalos et al., editors, *Proceeding of Learning and Intelligent OptimizatioN (LION8)*, volume 8426 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2014.

[242] Z. Yuan, T. Stützle, M. A. Montes de Oca, H. C. Lau, and M. Birattari. An analysis of post-selection in automatic configuration. In *Proceeding of GECCO*, pages 1557–1564. ACM, 2013.

[243] M. Zlochin, M. Birattari, N. Meuleau, and M. Dorigo. Model-based search for combinatorial optimization: A critical survey. *Annals of Operations Research*, 131(1–4):373–395, 2004.