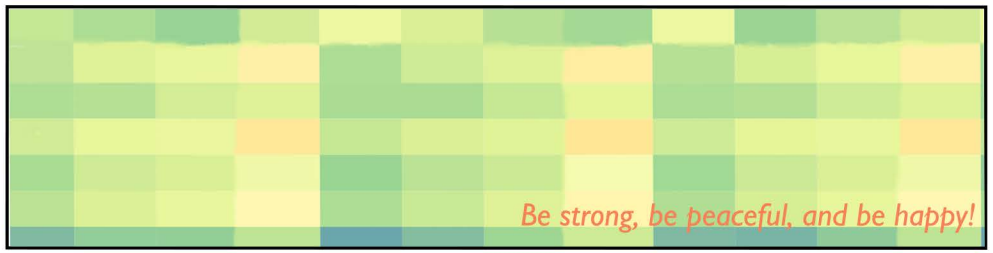
Benchmarking discrete optimization heuristics: from building a sound experimental environment to algorithm configuration

Ye, F.



Benchmarking Discrete Optimization Heuristics: From Building a Sound Experimental Environment to Algorithm Configuration Furong Ye 叶馥榕

Benchmarking Discrete Optimization Heuristics

From Building a Sound Experimental Environment to Algorithm Configuration

Furong Ye 叶馥榕

Benchmarking Discrete Optimization Heuristics: From Building a Sound Experimental Environment to Algorithm Configuration

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van rector magnificus prof.dr.ir. H. Bijl,
volgens besluit van het college voor promoties
te verdedigen op woensdag 1 juni 2022
klokke 13:45 uur

door

Furong Ye
geboren te Guizhou, China
in 1993

Promotor:
Prof.dr. Thomas H.W. Bäck

Co-promotores:
dr.-ing. HDR Carola Doerr (CNRS and Sorbonne Université, France)
dr. Hao Wang

Promotiecomissie:
Prof.dr. Aske Plaat
Prof.dr. Marcello M. Bonsangue
dr. Anna V. Kononova
Prof.dr.-ing. Frank Neumann (University of Adelaide, Australia)
Prof.dr. Pascal Kerschke (Technische Universität Dresden, Germany)

Het onderzoek beschreven in dit proefschrift is uitgevoerd aan het Leiden Institute of Advanced Computer Science (LIACS).

This work is financially supported by the Chinese Scholarship Council (CSC No. 201706310143).

To my parents.

"Nothing can be accomplished without norms or standards."

Mencius, 372-289 BC

Contents

Chapter 1

Introduction

A fundamental research field of optimization is discrete optimization, which is prevalent in real-world applications, ranging from operation research (e.g., scheduling [33] and logistics [9, 66]) to other fields of computer science (e.g., neural architecture search [64]). It is also of vital importance in theoretical studies, e.g., NP-hardness of optimization problems [6, 66, 70] and the analysis of algorithmic complexity of optimization algorithms [37, 40, 49, 84, 90, 91, 110].

Many discrete optimization problems are hard-to-solve, and various algorithms are proposed in the literature, e.g., [9, 10, 33, 70, 72, 99, 102, 152], to solve different problems. In this thesis, we focus only on the study of *evolutionary computation (EC)* methods, which have been widely and successfully applied for discrete optimization [9, 33, 70, 72, 99, 152]. An increasing number of algorithms such as genetic algorithms (GAs) [161], estimation of distribution algorithms (EDAs) [103], genetic programming (GP) [101], etc., are being proposed to solve different complex problems. Meanwhile, we lack a clear conclusion about the performance of these algorithms across different types of problems. Therefore, one key challenge of our research community is *to develop guidelines on which algorithms to favor for which kinds of problems.* Another important challenge is to *enhance our understanding of performance limits of EC methods,* so as to gain insight into the potential of further improving their design. Different approaches have been conducted to address these two key challenges, ranging from theoretical analysis of the algorithms [51, 86, 124] to a practical comparison of the algorithms on real-world optimization problems [134, 164].

1.1 Benchmarking in Optimization

The theory-oriented approach and the approach of real-world optimization are conducted to understand algorithms' performance. Meanwhile, benchmarking studies can offer an intermediate approach associating these two approaches. The two approaches complement each other, as they can help us gain insight into algorithms' behaviour via different approaches, and there is much untapped potential in bounding these approaches together. For example, an algorithm proposed using one approach can help solve problems in another approach by generating hypotheses tested on the approach where the algorithm was originally proposed, which can be easily achieved between the benchmarking study and real-world optimization by applying algorithms assessed on benchmarks to real-world optimization problems. Benchmarking may also benefit theoreticians by enhancing mathematically-derived ideas into techniques being broadly applicable in practical optimization. It also constitutes a catalyst for formulating new research questions for theoretical research and the real-world optimization domain.

The recent discussion [12] in the EC community has summarized that benchmarking studies can aim for: (1) *Assessment of Problems and Algorithms.* Benchmarking can provide empirical analyses of algorithms' performance across different optimization problems, addressing the questions of how the increasing number of algorithms compare across different optimization problems. Consequently, we can select the "winner" algorithms for competitions. On the other hand, it can also help us assess the optimization problems and illustrate the algorithms' behaviour during the optimization process; (2) *Sensitivity of Performance.* Benchmarking can assess the sensitivity of algorithms with response to different problem instances. It will also benefit parameter tuning by helping us learn the impact of the parameters for an algorithm on certain problem instances. Moreover, it supports characterizing algorithms' performance by problem instance features and vice versa; (3) *Performance Exploration.* Benchmarking can produce data for machine learning tasks to explore promising algorithms for a given problem. Benchmarking data can also be used for automatic algorithm design, selection, and configuration; (4) *Complementing and Inspiring Theoretical Study.* The experimental results of benchmarking studies can help valid theoretical results. Moreover, the benchmarking results may inspire theoretical works; (5) *Reproducibility and Algorithm Development.* Benchmarking projects can support us in verifying the reproducibility of a given program. Also, understanding algorithms' behavior addresses the question of how the underlying design of algorithms can be used to solve different types of optimization problems. Furthermore, benchmarking may inspire novel designs

of algorithms.

We will demonstrate the potential of benchmarking by showing in this thesis how it inspires new theoretical results and the design of new algorithms. Also, we will present how we can gain insights into the behaviour of the automatic algorithm configuration tools based on experiments on classic theory-oriented problems.

1.2 Scope of the Thesis

We will apply the benchmarking approach in this thesis to investigate the performance of EC methods on different problems. With respect to the algorithms to be investigated, we focus on *iterative optimization heuristics* (IOHs). The IOHs aim to find the optimal solution for a problem minimizing or maximizing $f : S \to \mathbb{R}, x \mapsto f(x)$ by searching for new solution candidates iteratively. For each iteration, one or a set of solution candidates $\{s_1, s_2, \ldots, s_\lambda\} \in S$ are generated. The algorithm terminates when a specific criterion is met, e.g., the time budget is reached, or a solution with the required quality is found. The class of IOHs subsumes evolutionary algorithms (EAs), (Quasi-)Monte Carlo algorithms, swarm intelligence, differential evolution, EDAs, efficient global optimization, Bayesian optimization, local search variants (for example, first/steepest ascent and variable neighbourhood search), etc. As for the optimization problems, we focus on *pseudo-Boolean optimization problems*, a subset of discrete optimization.

1.3 The Demand for a New Benchmarking Environment

In the context of discrete optimization, several attempts to construct widely accepted benchmarking software have been undertaken [81, 132], but these are typically restricted to certain problem classes (often classical NP-hard problems such as Satisfiability (SAT) [81], Traveling Salesperson Problem (TSP) [132], etc.) without attempting to generate a set of scalable or generalizable optimization problems. In addition, many frameworks strongly focus on constructive heuristics, which are assumed to have access to the problem instance data (in contrast to black-box optimization heuristics, which implicitly learn about the problem instance only through the evaluation of potential solutions). The few attempts to create a sound benchmarking platform for discrete black-box optimization heuristics, e.g., Weise's optimization benchmarking

platform [158], did not receive significant attention from the scientific community.

In December 2018, Facebook announced its benchmarking environment for black-box optimization [131]. Though their Nevergrad platform comprises a few discrete problems such as TSP, shifted sphere function [145], etc., it mainly focuses on noisy continuous optimization. Another famous benchmarking project, the COmparing Continuous Optimizers (COCO) [79] software constitutes well-established and widely recognized software for benchmarking derivative-free black-box optimization heuristics. The COCO software is under constant development. Apart from its well-known BBOB benchmark set [80], it also offers noisy, multi-objective [149], and mixed-integer [148] problem collections. While COCO has been designed to analyze IOHs on different types of problems, its designers have chosen to pre-select these problems that users can test their algorithms on. Benchmarking new problems with COCO requires substantial knowledge of its software design, and is therefore quite time-consuming. Also, it requires additional work using COCO to extend the performance statistics and visualizations.

To break down barriers between different tools and unify research ideas from different domains, we wish for a sound benchmarking environment that allows algorithms to be tested on a wide range of problems and visualize algorithm behavior with statistical analysis. Moreover, implementing problems or algorithms of existing benchmark projects shall not require much effort to be integrated into this new benchmarking software. The design of such benchmarking environment shall take in account the components of optimization algorithms to be compared, the types of optimization problems, the performance measures used to evaluate algorithms, and other possible techniques to present algorithms' behaviour. Considering these components, we have developed the **benchmarking software IOHprofiler** with a modular structure, which provides the environment to perform the study of this thesis.

1.4 Research Questions

This thesis concentrates on an overarching research question:

How can benchmarking studies benefit theoretical analysis on the optimization problems and the practical study of algorithm design?

From considering *how to build benchmarking software* to presenting *what we gain from benchmarking studies*, we provide comprehensive study cases to illustrate valid answers

for this question.

As building the IOHprofiler benchmarking software, we discuss the questions:

1. *Which components shall an applicable benchmarking pipeline take into account?* To answer this question, we have addressed the sub-questions of what problems and algorithms we are interested in, how to collect the raw data of experiments, what performance measures we can apply for analyzing algorithms' performance, and how to visualize these results. In addition, technique issues of software development such as the choices of programming language, the efficiency of implementation, and version control are also involved in developing our IOHprofiler benchmarking software.

Benefiting from IOHprofiler, we can compare different algorithms on a set of benchmarking problems for the following open question:

2. *How do various evolutionary algorithms perform on different problems, and what is the impact of the parameters and the operators?* In this thesis, we investigate how the population size affects the performance of $(1+\lambda)$ EAs on OneMax and LeadingOnes, how the optimal mutation rates adjust during the optimization process for OneMax and LeadingOnes, if/how the uniform crossover is helpful on LeadingOnes, and what the promising configurations of a $(\mu+\lambda)$ GA are for different types of problems.

The benchmarking results motivate future research on the topic of parameter tuning, answering the following questions:

3. *How to interpolate local and global search during optimization?* This question is motivated by observing how randomized local search (RLS) and different EAs perform at different stages of the optimization process for OneMax and LeadingOnes

4. *What is the impact of the cost metric (i.e., the expected running time (ERT) and the area under the empirical cumulative distribution function curve (AUC)) for the performance of algorithm configuration methods?* We investigate the performance of the promising configurations obtained using algorithm configuration methods. Moreover, the results help us understand the behaviour of algorithm configuration methods.

5. *What are the next steps for algorithm configuration?* We leverage our benchmarking data of various GAs for the topic of dynamic algorithm selection. Our

investigation on the performance of different dynamic algorithm policies reveals the potential topics for the future work on dynamic algorithm selection.

1.5 Our Contributions

Overall, this thesis involves three topics: benchmarking discrete optimization algorithms, empirical analyses of evolutionary computation, and automatic algorithm configuration. The objective is benchmarking EAs on discrete optimization for the selection and the design of better optimizers.

In practice, we start with **building the IOHprofiler benchmark software.** IOHprofiler consists of two main parts: IOHexperimenter and IOHanalyzer. IOHexperimenter provides a platform for programming experiments, and IOHanalyzer performs post-processing of algorithm performance data. IOHprofiler is the cornerstone of the study of this thesis. It supports us in testing algorithms on a wide range of problems and allows us to perform and visualize the statistical analysis on algorithms' performance. Benefiting from the functionalities of IOHprofiler, we can systematically work on the benchmark study in the following chapters.

Our initial benchmarking work focuses on the two classic problems OneMax and LeadingOnes. We **study the impact of mutation rate and population size** on the $(1 + \lambda)$ EA. We find that the $(1 + \lambda)$ EA benefits from small λ for OneMax. However, the value of λ does not significantly affect the ERT values for LeadingOnes. In addition, **we observe that crossover can help for LeadingOnes** by testing a $(\mu + \lambda)$ genetic algorithm with different crossover probabilities. We find that as μ increases, the value of the optimal crossover probability increases, while for fixed μ, its value decreases with increasing problem dimension.

The work of analyzing EAs and local search algorithms on OneMax and LeadingOnes clearly shows that local search is preferable at some (in our use-case, late) stages of the optimization process, whereas larger *mutation strengths* of the EAs are beneficial at other stages. Adaptive choice allows us to leverage complementarity. Therefore, we analyze in this thesis a smooth way of interpolating between local and non-local search by **proposing a new *normalized bit mutation***. Experimental results show improvement in using the normalized bit mutation compared to using either local search or EAs.

Our benchmarking study is also performed on more benchmarking problems provided by IOHprofiler. Twelve heuristics and various configurations of the $(\mu+\lambda)$ GA are tested. **We investigate how crossover and mutation interplay with each**

other and the impact of population size. The obtained result provides us, for different benchmark problems, the promising settings of mutation rate, crossover probability, and population size. The benchmark data also inspires us towards the work of automatic algorithm configuration and dynamic algorithm selection.

We apply three AC techniques: iterated racing (Irace) [111], mixed-integer parallel efficient global optimization (MIP-EGO) [155], and mixed-integer evolutionary strategies (MIES) [109], to configure the $(\mu + \lambda)$ GA for two different objectives, i.e., ERT and AUC, respectively. The AC methods aim at automatically finding the best configuration (i.e., the optimal parameter setting and operator choice) of an algorithm for the given problem. However, first, we need to decide on a *cost metric* as the objective of the configuration task. We investigate the impact of minimizing ERT and maximizing AUC as the cost metric, respectively. **Our results suggest that even when interested in expected running time performance (i.e., ERT), it might be preferable to use *anytime performance* measures (i.e., AUC) for the configuration task. We also observe that tuning for ERT is much more sensitive with respect to the budget that is allocated to the target algorithms.**

Apart from applying algorithm configuration for static settings of evolutionary algorithms, we are also interested in the dynamic settings. **We leverage our benchmark data of static algorithms for the study of *dynamic algorithm selection*.** The study inspires us to focus on the automatic detection of the timing of switching algorithms and efficient *warm-start* strategies of the switching.

1.6 Structure of the Thesis

Chapter 2 provides relevant background for this thesis. In particular, we provide here an introduction to optimization, EAs, the performance measures, and the benchmark problems that are used in this thesis. Chapter 3 then introduces the IOHprofiler benchmarking software. Chapter 4 presents our first use-case of benchmarking on the two classic problems, OneMax and LeadingOnes. The proposed standard normalized bit mutation is also introduced in Chapter 4. Chapter 5 continues to benchmark twelve heuristics and various configurations of the $(\mu + \lambda)$ GA on more benchmark problems provided by IOHprofiler. The topic of AC is discussed in Chapter 6, where we apply the three techniques, Irace, MIP-EGO, and MIES, to configure the $(\mu + \lambda)$ GA. In Chapter 7, we leverage the benchmark data of Chapter 5 for the study of dynamic algorithm selection. The thesis is summarized in Chapter 8, which briefly

recalls the main contributions and discusses future research topics inspired by our work.

1.7 Software and Publications

This thesis is based on the following works.

1.7.1 Software and Documentation

A key contribution is the IOHprofiler environment that is available on the following websites.

- IOHprofiler wiki page: https://iohprofiler.github.io.
- IOHprofiler Github https://github.com/IOHprofiler.
- IOHanalyzer web-based interface: https://iohanalyzer.liacs.nl.

1.7.2 Journal Publications

1. Carola Doerr, Furong Ye, Naama Horesh, Hao Wang, Ofer M Shir, and Thomas Bäck. Benchmarking discrete optimization heuristics with IOHprofiler. *Applied Soft Computing*, 106027. Elsevier, 2020.

   This paper contributes to Section 5.1. It describes our benchmarking results of twelve heuristics on the twenty three problems provided by IOHprofiler.

2. Furong Ye, Carola Doerr, Hao Wang, and Thomas Bäck. Automated Configuration of Genetic Algorithms by Tuning for Anytime Performance. *IEEE Transactions on Evolutionary Computation*. IEEE, 2022.

   This paper contributes to Chapter 6. It presents our work of tuning the $(\mu + \lambda)$ GA for minimizing ERT or maximizing AUC on the twenty five problems provided by IOHprofiler, using the three AC techniques, Irace, MIP-EGO, and MIES.

3. Hao Wang, Diederick Vermetten, Furong Ye, Carola Doerr, Thomas Bäck. IOHanalyzer: Detailed Performance Analyses for Iterative Optimization Heuristic. *ACM Transactions on Evolutionary Learning and Optimization*, in press. ACM, 2022.

This paper contributes to Section 3.3. It introduces the IOHanalyzer module of IOHprofiler in detail.

4. Jacob de Nobel*, Furong Ye*, Diederick Vermetten, Hao Wang, Carola Doerr, and Thomas Thomas Bäck. IOHexperimenter: Benchmarking Platform for Iterative Optimization Heuristics. *arXiv preprint arXiv:2111.04077*. 2021. (Under revision of *Evolutionary Computation Journal*)(*These authors contributed equally to this work.)

   This paper contributes to Section 3.2, which introduces the IOHexperimenter module of IOHprofiler in details.

1.7.3 Peer-reviewed Conference Publications

1. Carola Doerr, Furong Ye, Sander van Rijn, Hao Wang, Thomas Bäck. Towards a theory-guided benchmarking suite for discrete black-box optimization heuristics: profiling $(1 + \lambda)$ EA variants on OneMax and LeadingOnes. *In Proc. of Genetic and Evolutionary Computation Conference (GECCO'18)*, 951–958. ACM, 2018.

   This paper contributes to Section 4.1. It publishes our benchmarking results of the $(1 + \lambda)$ EAs on OneMax and LeadingOnes. The results motivated a refined analysis for the optimization time of the $(1 + \lambda)$ EA on LeadingOnes.

2. Furong Ye, Carola Doerr, and Thomas Bäck. Interpolating Local and Global Search by Controlling the Variance of Standard Bit Mutation. *In Proc. of IEEE Congress on Evolutionary Computation (CEC'19)*, 2292–2299. IEEE, 2019.

   This paper contributes to Section 4.2. It investigates how the mutation rate affects the performance of the $(1+\lambda)$ EAs on OneMax and LeadingOnes. The standard normalized bit mutation is proposed in this work.

3. Furong Ye, Hao Wang, Carola Doerr, and Thomas Bäck. Benchmarking a $(\mu+\lambda)$ Genetic Algorithm with Configurable Crossover Probability. *In Proc. of Parallel Problem Solving from Nature (PPSN'20)*, 699-713. Springer, 2020.

   This paper contributes to Sections 4.3 and 5.2. It investigates the impact of the crossover probability for the $(\mu+\lambda)$ GA. This work shows that crossover

can be helpful for LeadingOnes and inspires us with the study of dynamic crossover probability.

4. Furong Ye, Carola Doerr, and Thomas Bäck. Leveraging Benchmarking Data for Informed One-Shot Dynamic Algorithm Selection. *In Proc. of Genetic and Evolutionary Computation Conference (GECCO'21), Companion Material*, 245–246. ACM, 2021.

   This paper contributes to Chapter 7. It publishes our result of leveraging benchmark data in Section 5.2 for dynamic algorithm selection.

1.7.4 Other Documentation

1. Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, Thomas Bäck. IOHprofiler: A benchmarking and profiling tool for iterative optimization heuristics. *arXiv preprint arXiv:1810.05281*. 2018.

   This article contributes to Chapter 3. It presents the general overview of the IOHprofiler software.

Chapter 2

Preliminaries

This chapter briefly introduces three topics of this thesis, covering *Optimization*, *Evolutionary Algorithms*, and *Algorithm Configuration*. Detailed descriptions of performance measures and benchmark problems used in the empirical study are also provided.

2.1 Optimization

Optimization aims at finding the *best* solution for a given problem. Optimization problems arise in many disciplines, e.g., biology [75], engineering [33], logistics [9, 66], physics [113], etc. Meanwhile, benchmarks have been built to unify ideas and methods for different domains such as numerical analysis [79], software engineering [81], traveling salesperson problems [132], etc.

Optimization problems consist of three elements, i.e., objective functions, decision variables, and constraints.

The **objective function** is a mapping that assesses the quality of a candidate solution. The assessment returns a value (i.e., single-objective optimization) or a set of values (i.e., multi-objective optimization). We only consider the single-objective optimization that is subject to maximization in this thesis. In short, we aim at maximizing a function:

$$f : S \to \mathbb{R}, x \mapsto f(x), \tag{2.1}$$

where we refer to S as the search space and its element $x \in S$ as a search point (or solution candidate). The decision variables of x can have different types: real

(i.e., continuous), integer (e.g., ordinal), and nominal (i.e., categorical). Continuous optimization considers only real-valued variables, discrete optimization considers integer and nominal variables, and mixed-integer problems consist of multiple types of variables.

Constraints restrict the values of variables that can be taken for solutions, which can be either hard constraints or soft constraints. Solutions must satisfy the requirements of hard constraints, and penalties will be assigned if soft constraints are violated [28]. Also, constraints can be distinguished by *equality constraints* $h(x) = 0$ and *inequality constraints* $g(x) \leq 0$.

2.1.1 Pseudo-Boolean Optimization

In this thesis, we study a subset of discrete optimization, whose variables consist of binary variables, namely *pseudo-Boolean optimization* [19]:

$$f : \{0,1\}^n \to \mathbb{R}, x \mapsto f(x). \tag{2.2}$$

A variety of problems are related to pseudo-Boolean optimization, including spin glass [11], maximum satisfiability [70], fault location [114], clustering [135], project selection [130], etc. Many techniques have been applied to solve these problems specifically. In this thesis, we study the performance of EAs and other IOHs on pseudo-Boolean optimization problems.

2.2 Evolutionary Algorithms

Though exact algorithms, such as dynamic programming, have been developed for solving optimization problems, these techniques usually require additional effort for specific problems and can not solve hard problems. However, EAs have achieved success in approximating solutions to problems in many fields [6, 9, 33, 64, 66, 70].

EAs were originally inspired by biological evolution. The general procedure of EAs is producing *offspring* using variation operators after initializing a *parent* population of solution candidates (i.e., individuals), then updating the population by selecting from offspring (and parent) solution candidates. This procedure is iterated until the termination criterion is reached.

Different design of operators will result in variations of EAs such as GAs, evolution strategies (ESs), etc. For instance, mutation and crossover are two common operators of EAs. Mutation allows exploiting promising search areas using small mutation rates,

and crossover creates offspring by recombining information of two or more parents. GAs use both mutation and crossover as variation operators. However, the EAs for pseudo-Boolean optimization usually concentrate on mutation only. Also, there are different strategies to form new populations. For example, a plus-strategy will select solution candidates from both parent and offspring populations, and a comma-strategy will consider only the offspring population.

Apart from the combinations of different operators, we have a variety of algorithms because most operators are parameterized, raising the question of how to configure them properly for the given optimization task. For example, EAs require proper settings of the population size, the mutation rate, and the selective pressure; GAs' performance can be affected by the values of the population size and the crossover probability. Note that we consider that the EAs for pseudo-Boolean optimization in this thesis are mutation-only algorithms.

2.3 Parameter Tuning Techniques

It is well known that the choice of parameters and operators influences the performance of EAs significantly [2, 93]. There are two classes of approaches to determine algorithm settings, namely static parameter setting and dynamic parameter control [42].

Static parameter setting identifies parameter values and operator choices of the algorithms for a given problem. The settings are predefined for the optimization process. Usually, the settings are based on empirical studies or theoretical works. For the empirical study methods, the design of experiments (DOE) [26] methods can help us understand the relationship between input parameters and decide a promising algorithm setting. Nowadays, automatic tuning tools have also been applied to identify algorithms' parameters, and the obtained results have shown significant advantages against manual settings. In the theory research domain, researchers prove bounds for the running time of algorithms with respect to specific parameters values, such that we can find suitable parameter settings by minimizing these bounds [163].

Dynamic parameter control adjusts parameter values and even operator choices during the optimization process. The aim is to benefit from applying promising settings at different stages of the optimization process. Both empirical [2, 61, 92] and theoretical studies [42] have been conducted for dynamic algorithm control. Recent work has formulated dynamic algorithm configuration as a contextual Markov decision process, and the authors compared the reinforcement learning and the classic sequential model-based optimization for general algorithm configuration (SMAC) on

their test bed [17]. Also, a similar topic, dynamic algorithm selection, has attracted attention in recent research [153]. Theoreticians also investigate the optimal parameters for theory-oriented problems such as OneMax and proposed theory-inspired self-adaptation methods [40, 46].

In the following, we introduce *algorithm configuration (AC)*, also known as hyperparameter optimization, which belongs to the class of static parameter tuning approaches. AC is applied to explore promising operator combinations and parameter settings of the algorithms for a given problem. Most AC methods do not require preliminary knowledge of the algorithm and the problem. Consequently, we can not directly explain why (or if) the obtained configurations perform well. Benefiting from benchmarking, we can understand the behavior of both algorithms and the AC methods.

We provide here the definition of the algorithm configuration problem [60]:

Definition 2.1 (AC: Algorithm Configuration)**.** Given a set of problem instances P, a parametrized algorithm A with parameter space Θ, and a cost metric $c : \Theta \times P \to \mathbb{R}$ that is subject to minimization, the objective of the AC problem is to find a configuration $\theta^* \in \Theta$ such that the cost $c(\theta, P)$ is as small as possible.

The AC problem can be seen as a meta-optimization problem, asking to optimize performance of a specific solver on a given set of problem instances.

Many approaches, e.g., Bayesian optimization [155], local search [83], evolutionary algorithms [109], gradient-based optimization [16], etc., can be used for the AC problem. Among the best-known tools are paramILS [83], SMAC [82], and Irace [111]. These methods have been applied to boost the performance of algorithms in many domains such as TSP [111], software engineering [14], and machine learning [97].

2.4 Algorithm Performance Measures

We introduce here the performance measures used in this thesis to assess algorithms' behaviour, concerning three different objectives, namely *fixed-target performance*, *fixed-budget performance*, and *anytime performance*. The measures can also be used as the cost metric in Definition 2.1.

2.4.1 Fixed-target Performance

For the fixed-target results, we consider the cost needed by each algorithm to find a solution that is at least as good as a certain target. In this thesis, we consider the

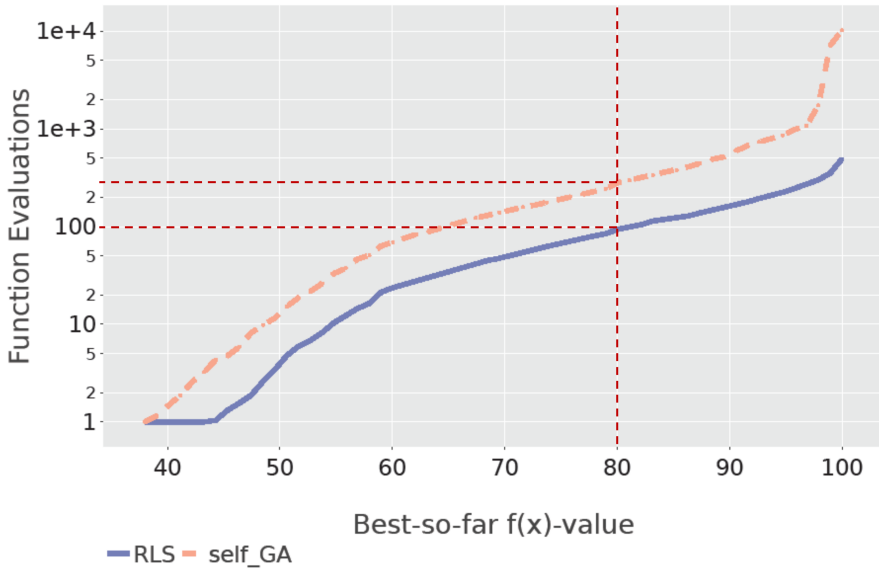


Figure 2.1: ERT values of the two algorithms RLS and self_GA for a maximization problem in a fixed-target perspective.

time cost measured by the expected running time (ERT) value, where *time* indicates the number of function evaluations.

Definition 2.2 (ERT: Expected Running Time)**.** Given a target ϕ for a problem P, the ERT of an algorithm A hitting ϕ is

$$\mathrm{ERT}(A, P, \phi) = \frac{\sum_{i=1}^{r} \min\{t_i(A, P, \phi), B\}}{\sum_{i=1}^{r} \mathbb{1}\{t_i(A, P, \phi) < \infty\}} , \tag{2.3}$$

where r is the number of independent runs of A, B is the given budget (i.e., the maximal number of function evaluations), $t_i(A, P, \phi) \in \mathbb{N} \cup \{\infty\}$ is the running time (for finite values, the running time is the number of function evaluations that the i-th run of A on the problem P uses to hit the target ϕ, and $t_i(A, P, \phi) = \infty$ is used if none of the solutions is better than ϕ), and $\mathbb{1}(\mathcal{E})$ is the indicator function returning 1 if event $\mathcal{E}$ happens and 0, otherwise. $t_i(A, P, \phi) < \infty$ indicates that the algorithm hits the target within the given budget B in the i-th run. If the algorithm hits the target ϕ in all r runs, the ERT is equal to the average hitting time (AHT).

Definition 2.3 (AHT: Average Hitting Time)**.** Given a target ϕ for a problem P, the

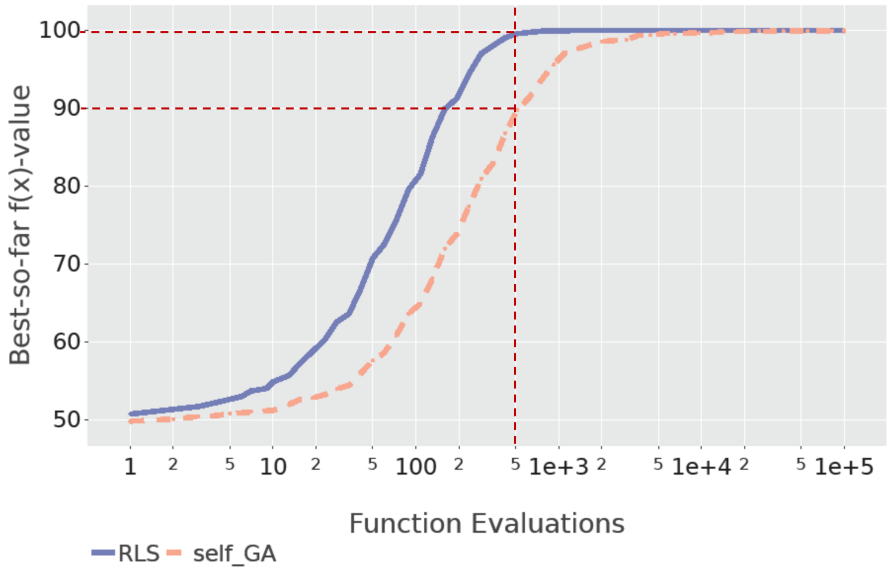


Figure 2.2: The mean of best-found fitness values of the two algorithms RLS and self_GA for a maximization problem in a fixed-budget perspective.

AHT of an algorithm A hitting ϕ is

$$\mathrm{AHT}(A, P, \phi) = \frac{\sum_{i=1}^{r} t_i(A, P, \phi)}{r}\,, \tag{2.4}$$

where r is the number of independent runs of A, $t_i(A, P, \phi)$ is the running time (i.e., the number of function evaluations) that the i-th run of A uses to hit the target ϕ of P.

Figure 2.1 is an example showing fixed-target curves, which plots the ERT values (y-axis) that the two algorithms RLS and self_GA need to find a solution satisfying $f(x) \geq \phi$, where the target value ϕ is the value on x-axis. We see that the ERT of the RLS for the target value 80 is around 100, while the ERT of the self_GA is around 300.

2.4.2 Fixed-budget Performance

For the fixed-budget results, we consider the quality of solutions found by each algorithm with a given budget. Figure 2.2 is an example showing fixed-budget curves, which plots average of the best-found fitness values (y-axis) after using specific budget

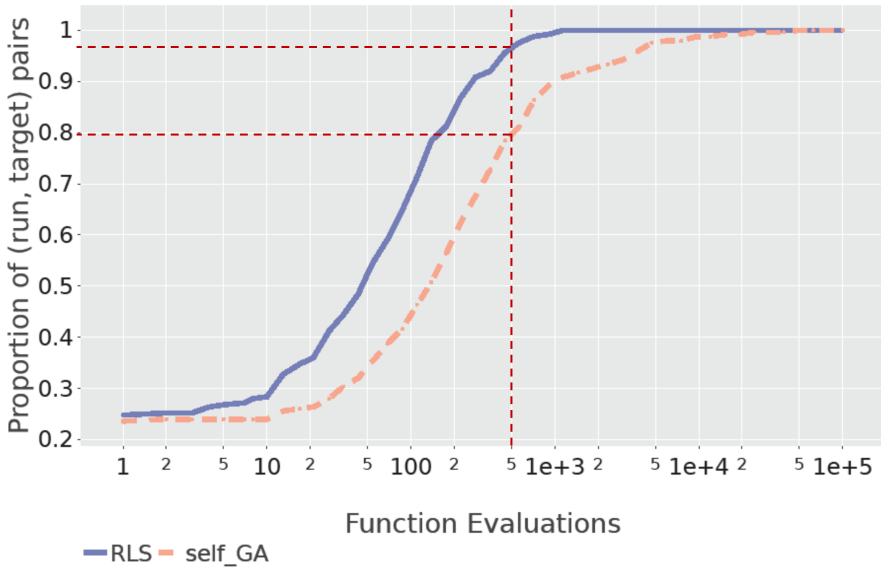


Figure 2.3: ECDF values of the two algorithms RLS and self_GA for different budgets.

B, where the budget value B is the value on x-axis. We see that, after using 500 function evaluations, the mean of best-found fitness value of the RLS is 100, while the mean of best-found fitness value of the self_GA is 90.

2.4.3 Anytime Performance

Another important concept in the analysis of IOHs are empirical cumulative distribution function (ECDF) curves, which allow to aggregate performance across different targets. The definition of ECDF is given below. Figure 2.3 plots ECDF values (y-axis) after using specific budget B, where the budget value B is the value on x-axis. We see that, around 96% runs of the (run,target) pairs hit the corresponding target within the given budget of 500 function evaluations for the RLS, while this number is 80% for the self_GA. Note that ECDF can also be applied for evaluating algorithm performance across different functions.

Definition 2.4 (ECDF: empirical cumulative distribution function of the running time)**.** Given a set of targets $\Phi = \{\phi_i \in \mathbb{R} \mid i \in \{1, 2, \ldots, m\}\}$ for a real-valued problem P and a set of budgets $T = \{t_j \in \mathbb{N} \mid j \in \{1, 2, \ldots, B\}\}$ for an algorithm A, the ECDF value of A at budget t_j is the fraction of (run, target)-pairs (r, ϕ_i) that

satisfy that the run r of the algorithm A finds a solution has fitness at least as good as ϕ_i within the budget t_j.

In this thesis, we evaluate anytime performance of algorithms using the area under the ECDF curve (AUC). The domain of AUC values is $[0, 1]$. The definition is given below. Note that the given definition is for a discretized version of AUC, of which values can be affected by the given targets Φ and budgets T.

Definition 2.5 (AUC: area under the ECDF curve)**.** Given a set of targets $\Phi = \{\phi_i \in \mathbb{R} \mid i \in \{1, 2, \ldots, m\}\}$ and a set of budgets $T = \{t_j \in \{1, 2, \ldots, B\} \mid j \in \{1, 2, \ldots, z\}\}$, the AUC $\in [0, 1]$ (normalized over B) of algorithm A on problem P is the area under the ECDF curve of the running time over multiple targets. For maximization, it reads

$$\mathrm{AUC}(A, P, \Phi, T) = \frac{\sum_{h=1}^{r} \sum_{i=1}^{m} \sum_{j=1}^{z} \mathbb{1}\{\phi_h(A, P, t_j) \geq \phi_i\}}{rmz},$$

where r is the number of independent runs of A and $\phi_h(A, P, t)$ denotes the value of the best solution that A evaluated within its first t evaluations of the run h.

2.5 Benchmark Problems

In this thesis, we focus on *pseudo-Boolean optimization problems*, i.e., all the suggested benchmark problems are expressed as functions $f : \{0, 1\}^n \to \mathbb{R}$. We also pay particular attention to the *scalability* of the problems, with the idea that the benchmark problems should allow to assess performances across different dimensions. All problems have been implemented and integrated within the IOHprofiler software.

Conventions Throughout this thesis, the variable n denotes the dimension of the problem that the algorithm operates upon. We assume that n is known to the algorithm; this is a natural assumption, since every algorithm needs to know the decision space that it is requested to search. Note though, that the *effective dimension* of a problem can be smaller than n, e.g., due to the usage of *dummy variables* that do not contribute to the function values, or due to other reductions of the search space dimensionality (see Section 2.5.6 for examples). In practice, we thus only require that n is an upper bound for the effective number of decision variables.

For *constrained problems*, such as the N-Queens problem (see Section 2.5.10), we follow common practice in the evolutionary computation community and use penalty

terms to discount infeasible solutions by the number and magnitude of constraint violations.

We formulate all problems as *maximization* problems.

Notation A search point $x \in \{0,1\}^n$ is written as $(x_1, \ldots, x_n)$. By $[k]$ we abbreviate the set $\{1, 2, \ldots, k\}$ and by $[0..k]$ the set $[k] \cup \{0\}$. All logarithms are to the base 10 and are denoted by log. An exception is the natural logarithm, which we denote by ln. Finally, we denote by id the identity function, regardless of the domain.

2.5.1 Problems vs. Instances

We define a *problem* in this thesis as a collection of *functions* sharing some common properties. For example, the NK landscape problem refers to problems with different gene interactions [94]. While we are interested in covering different types of fitness landscapes, we care much less about their actual embedding, and mainly seek to understand algorithms that are invariant under the problem representation. In the context of pseudo-Boolean optimization, a well-recognized approach to request representation invariance to demand that an algorithm shows the same or similar performance on any instance mapping each bit string $x \in \{0,1\}^n$ to the function value $f(\sigma(x \oplus z))$, where z is an arbitrary bit string of length n, $\oplus$ denotes the bit-wise XOR function, and $\sigma(y)$ is to be read as the string $(y_{\sigma(1)}, \ldots, y_{\sigma(n)})$ in which the entries are swapped according to the permutation $\sigma : [n] \to [n]$. Using these transformations, we obtain from one particular problem f a whole set of instances $\{f(\sigma(\cdot \oplus z)) \mid z \in \{0,1\}^n, \sigma \text{ permutation of } [n]\}$, all of which have fitness landscapes that are pairwise isomorphic. Further discussions of these *unbiasedness* transformations can be found in [54, 106].

Apart from unbiasedness, we also focus in this work on *ranking-based heuristics,* i.e., algorithms which only make use of *relative*, and not of *absolute* function values. To allow future comparisons with non-ranking-based algorithms, we test the algorithms on instances that are shifted by a multiplicative and an additive offset. That is, instead of receiving the values $f(\sigma(x \oplus z))$, only the transformed values $af(\sigma(x \oplus z)) + b$ are made available to the algorithms.

2.5.2 Overview of Selected Benchmark Problems

We summarize here the benchmark problems that we repeatedly use in this thesis to compare algorithms' performance, which are from the suite of IOHprofiler for

pseudo-Boolean optimization (PBO). The PBO suite originally consisted of twenty three problem [55], and two problems were added afterwards in [170].

- **F1 and F4-F10:** OneMax and its W-model extensions; details in Sections 2.5.3 and 2.5.6
- **F2 and F11-F17:** LeadingOnes and its W-model extensions; details in Sections 2.5.4 and 2.5.6
- **F3:** Harmonic; see Section 2.5.5
- **F18:** LABS: Low Autocorrelation Binary Sequences; see Section 2.5.7
- **F19-21:** Ising Models; see Section 2.5.8
- **F22:** MIVS: Maximum Independent Vertex Set; see Section 2.5.9
- **F23:** NQP: N-Queens; see Section 2.5.10
- **F24:** CT: Concatenated Trap; see Section 2.5.11
- **F25:** NKL: Random NK landscapes; see Section 2.5.12

2.5.3 F1: OneMax

The OneMax function is the best-studied benchmark problem in the context of discrete EC, often referred to as the "drosophila of EC". It asks to optimize the function

$$\text{F1} : \text{OM} : \{0,1\}^n \to [0..n], x \mapsto \sum_{i=1}^{n} x_i.$$

The problem has a very smooth and non-deceptive fitness landscape. Due to the well-known coupon collector effect (see, for example, [57] for a detailed explanation of this effect), it is relatively easy to make progress when the function values are small, and the probability to obtain an improving move decreases considerably with increasing function value.

With the '$\oplus z$' transformations introduced in Section 2.5.1, the OneMax problem becomes the problem of minimizing the Hamming distance to an unknown target string $z \in \{0,1\}^n$.

That OneMax is interesting beyond the study of theoretical aspects of evolutionary computation has been argued in [147]. We believe that OneMax plays a similar

role as the sphere function in continuous domains, and should be added to each benchmark set: it is not very time-consuming to evaluate, and can provide a first basic stress test for new algorithm designs.

2.5.4 F2: LeadingOnes

Among the non-separable functions, the LeadingOnes function is certainly the one receiving most attention in the theory of EC community. The LeadingOnes problem asks to maximize the function

$$\mathrm{F2} : \mathrm{LO} : \{0,1\}^n \to [0..n], x \mapsto \max\{i \in [0..n] \mid \forall j \leq i : x_j = 1\} = \sum_{i=1}^{n} \prod_{j=1}^{i} x_j, \quad (2.5)$$

which counts the number of initial ones.

Similar to OneMax, we argue that LeadingOnes should form a default benchmark problem: it is fast to evaluate and can point at fundamental issues of algorithmic designs, see also the discussions in Section 4.1.

2.5.5 F3: A Linear Function with Harmonic Weights

Two extreme linear functions are OneMax with its constant weights and binary value $\mathrm{BV}(x) = \sum_{i=1}^{n} 2^{n-i} x_i$ with its exponentially decreasing weights. An intermediate linear function is

$$\mathrm{F3} : \{0,1\}^n \to \mathbb{R}, x \mapsto \sum_i i x_i$$

with harmonic weights, which was suggested to be considered in [139].

2.5.6 F4-F17: The W-model

In [160], a collection of different ways to "perturb" existing benchmark problems in order to obtain new functions of scalable difficulties and landscape features has been suggested, the so-called W-model. These W-model transformations can be combined arbitrarily, resulting in a huge set of possible benchmark problems. In addition, these transformations can, in principle, be superposed to any base problems, giving yet another degree of freedom. Note here that the original work [160] and the existing empirical evaluations [159] only consider OneMax as underlying problem, but there is no reason to restrict the model to this function. We expect that in the longer term, the W-model, similarly to the well-known NK-landscapes [94] may constitute

an important building block for a scalable set of discrete benchmark problems. More research, however, is needed to understand how the different combinations influence the behavior of state-of-the-art heuristic solvers. In this thesis, we therefore restrict our attention to instances in which the different components of the W-model are used in an isolated way, see Section 2.5.6. The assessment of combined transformations clearly forms a promising line for future work.

The Basic Transformations

The W-model comprises four basic transformations, and each of these transformations is parametrized, hence offering a huge set of different problems already. We provide a brief overview of the W-model transformations that are relevant for the study in this thesis. A more detailed description can be found in the original work [160]. For some of the descriptions below we deviate from the exposition in [160], because in contrast to there, we consider *maximization* as objective, not minimization. Note also that we write $x = (x_1, \ldots, x_n)$, whereas in [160] the strings are denoted as $(x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$. Note also that the reduction of dummy variables is our own extension of the W-model, not originally proposed in [160].

1. **Reduction of dummy variables** $W(m, *, *, *)$**:** a reduction mapping each string $(x_1, \ldots, x_n)$ to a substring $(x_{i_1}, \ldots, x_{i_m})$ for randomly chosen, pairwise different $i_1, \ldots, i_m \in [n]$. This modification models a situation in which some decision variables do not have any or have only negligible impact on the fitness values. Thus, effectively, the strings $(x_1, \ldots, x_n)$ that the algorithm operates upon are reduced to substrings $(x_{i_1}, \ldots, x_{i_m})$ with $1 \leq i_1 < i_2 < \ldots < i_m \leq n$.

   We note that such scenarios have been analyzed theoretically, and different ways to deal with this *unknown solution length* have been proposed. Efficient EAs can obtain almost the same performance (in asymptotic terms) than EAs "knowing" the problem dimension [44, 62].

   Dummy variables are also among the characteristics of the benchmark functions contained in the Nevergrad platform [131], which might be seen as evidence for practical relevance.

   Example: With $n = 10$, $m = 5$, $i_1 = 1$, $i_2 = 2$, $i_3 = 4$, $i_4 = 7$, $i_5 = 10$, the bit string (1010101010) is reduced to (10010).

2. **Neutrality** $W(*, u, *, *)$**:** The bit string $(x_1, \ldots, x_n)$ is reduced to a string $(y_1, \ldots, y_m)$ with $m = n/u$, where u is a parameter of the transformation. For

each $i \in [m]$ the value of y_i is the majority of the bit values in a size-u substring of x. More precisely, $y_i = 1$ if and only if there are at least $u/2$ ones in the substring $(x_{(i-1)u+1}, x_{(i-1)u+2}, \ldots, x_{iu})$.[1] When $n/u \notin \mathbb{N}$, the last $n - u\lfloor n/u \rfloor$ remaining bits of x not fitting into any of the blocks are simply deleted; that is, we have $m = \lfloor n/u \rfloor$ and the entries x_i with $i > u\lfloor n/u \rfloor$ do not have any influence on y (and, thus, no influence on the function value).

Example: With $n = 10$ and $u = 3$ the bit string (1110101110) is reduced to (101).

3. **Epistasis** $W(*, *, \nu, *)$**:** The idea is to introduce local perturbations to the bit strings. To this end, a string $x = (x_1, \ldots, x_n)$ is divided into subsequent blocks of size ν. Using a permutation $e_\nu : \{0,1\}^\nu \to \{0,1\}^\nu$, each substring $(x_{(i-1)\nu+1}, \ldots, x_{i\nu})$ is mapped to another string $(y_{(i-1)\nu+1}, \ldots, y_{i\nu}) = e_\nu((x_{(i-1)\nu+1}, \ldots, x_{i\nu}))$. The permutation e_ν is chosen in a way that Hamming-1 neighbors $u, v \in \{0,1\}^\nu$ are mapped to strings of Hamming distance at least $\nu - 1$. Section 2.2 in [160] provides a construction for such permutations. For illustration purposes, we repeat below the map for $\nu = 4$, which is the parameter used in our experiments. This example can also be found, along with the general construction, in [160].

$e_4(0000) = 0000$	$e_4(0001) = 1101$	$e_4(0010) = 1011$	$e_4(0011) = 0110$
$e_4(0100) = 0111$	$e_4(0101) = 1010$	$e_4(0110) = 1100$	$e_4(0111) = 0001$
$e_4(1000) = 1111$	$e_4(1001) = 0010$	$e_4(1010) = 0100$	$e_4(1011) = 1001$
$e_4(1100) = 1000$	$e_4(1101) = 0101$	$e_4(1110) = 0011$	$e_4(1111) = 1110$

When $n/\nu \notin \mathbb{N}$, the last bits of x are treated by $e_{n-\nu\lfloor n/\nu \rfloor}$; that is, the substring $(x_{\nu\lfloor n/\nu \rfloor+1}, x_{\nu\lfloor n/\nu \rfloor+2}, \ldots, x_n)$ is mapped to a new string of the same length via the function $e_{n-\nu\lfloor n/\nu \rfloor}$.

Example: With $n = 10$, $\nu = 4$, and the permutation e_4 provided above, the bit string (1111011101) is mapped to (1110000110), because $e_4(1111) = 1110$ and $e_4(0111) = 0001$ and $e_2(01) = 10$.

4. **Fitness perturbation** $W(*, *, *, r)$**:** With these transformations we can de-

[1] Note that with this formulation there is a bias towards ones in case of a tie. We follow here the suggestion made in [160], but we note that this bias may have a somewhat complex impact on the fitness landscape. For our first benchmark set, we therefore suggest to use this transformation with odd values for u only.

termine the *ruggedness* and *deceptiveness* of a function. Unlike the previous transformations, this perturbation operates on the function values, not on the bit strings. To this end, a *ruggedness* function $r : \{f(x) \mid x \in \{0,1\}^n\} := V \to V$ is chosen. The new function value of a string x is then set to $r(f(x))$, so that effectively the problem to be solved by the algorithm becomes $r \circ f$.

To ease the analysis, it is required in [160] that the optimum $v_{\max} = \max\{f(x) \mid x \in \{0,1\}^n\}$ does not change, i.e., r must satisfy that $r(v_{\max}\}) = v_{\max}$ and $r(i) < v_{\max}$ for all $i < v_{\max}$. It is furthermore required in [160] that the ruggedness functions r are permutations (i.e., one-to-one maps). Both requirements are certainly not necessary, in the sense that additional interesting problems can be obtained by violating these constrains. We note in particular that in order to study *plateaus* of equal function values, one might want to choose functions that map several function values to the same value. We will include one such example in our testbed, see Section 2.5.6.

It should be noted that all functions of unitation (i.e., functions for which the function value depends only on the OneMax value of the search point, such as Trap or jump) can be obtained from a superposition of the fitness perturbation onto the OneMax problem.

Example: The well-known, highly deceptive Trap function can be obtained by superposing the permutation $r : [0..n] \to [0..n]$ with $r(i) = n - 1 - i$ for all $1 \leq i \leq n$ and $r(n) = n$.

Combining the Basic W-model Transformations

We note that any of the four W-model transformations can be applied independently of each other. The first three modifications can, in addition, be applied in an arbitrary order, with each order resulting in a different benchmark problem. In line with the presentation in [160], we consider in our implementation only those perturbations that follow the order given above. Each set of W-model transformations can be identified by a string $(\{i_1, \ldots, i_m\}, u, \nu, r)$ with $m \leq n$, $1 \leq i_1 < \ldots < i_m \leq n$, $u \in [n]$, $\nu \in [n]$, and $r : V \to V$, all to be interpreted as in the descriptions given in Section 2.5.6 above. Setting $\{i_1, \ldots, i_m\} = [n]$, $u = 1$, $\nu = 1$, and/or r as the identity function on V corresponds to not using the first, second, third, and/or forth transformation, respectively.

As mentioned, the W-model can in principle be superposed on any benchmark problem. The only complication is that the search space on which the algorithm

operates and the search space on which the benchmark problem is applied are not the same when $m < n$ or $u > 1$. More precisely, while the algorithm operates on $\{0,1\}^n$, the base problem has to be a function $f : \{0,1\}^s \to \mathbb{R}$ with $s = \lfloor m/u \rfloor$. We call s the *effective dimension* of the problem. When f is a scalable function defined for any problem dimension s—this is the case for most of our benchmark functions—we just reduce to the s-dimensional variant of the problem. When f is a problem that is only defined for a fixed dimension n, the algorithms should operate on the search space $\{0,1\}^\ell$ with $\ell \geq us$ and $\ell - us$ depending on the reduction that one wishes to achieve by the first transformation, the removal of dummy variables.

Selected W-Model Transformations

In contrast to existing works cited in [159, 160], we do not only study superpositions of W-model transformations to the OneMax problems (functions F4-F10), but we also consider LeadingOnes as a base problem (F11-17). This allows us to study the effects of the transformations on a well-understood separable and a well-understood non-separable problem. As mentioned, we only study individual transformations, and not yet combinations thereof.

We consider the reduction of $[n]$ to subsets of size $n/2$ and $0.9n$, i.e., only half and 90% of the bits, respectively, contribute to the overall fitness. We consider neutrality transformations of size $u = 3$, and we consider the epistasis perturbation of size $\nu = 4$. Finally, we consider the following ruggedness functions, where we denote by s the size of the effective dimension (see Section 2.5.6 for a discussion) and recall that both the s-dimensional OneMax and LeadingOnes functions take values in $[0..s]$. These functions are illustrated for $s = 10$ in Figure 2.4.

- $r_1 : [0..s] \to [0..\lceil s/2 \rceil + 1]$ with $r_1(s) = \lceil s/2 \rceil + 1$ and $r_1(i) = \lfloor i/2 \rfloor + 1$ for $i < s$ and even s, and $r_1(i) = \lceil i/2 \rceil + 1$ for $i < s$ and odd s.
- $r_2 : [0..s] \to [0..s]$ with $r_2(s) = s$, $r_2(i) = i + 1$ for $i \equiv s \pmod 2$ and $i < s$, and $r_2(i) = \max\{i - 1, 0\}$ otherwise.
- $r_3 : [0..s] \to [-5..s]$ with $r_3(s) = s$ and $r_3(s - 5j + k) = s - 5j + (4 - k)$ for all $j \in [s/5]$ and $k \in [0..4]$ and $r_3(k) = s - (5\lfloor s/5 \rfloor - 1) - k$ for $k \in [0..s - 5\lfloor s/5 \rfloor - 1]$.

We see that function r_1 keeps the order of the function values, but introduces small plateaus of the same function value. In contrast to r_1, function r_2 is a permutation of the possible function values. It divides the set of possible non-optimal function values $[0..s-1]$ into blocks of size two (starting at $s - 1$ and going in the inverse direction)

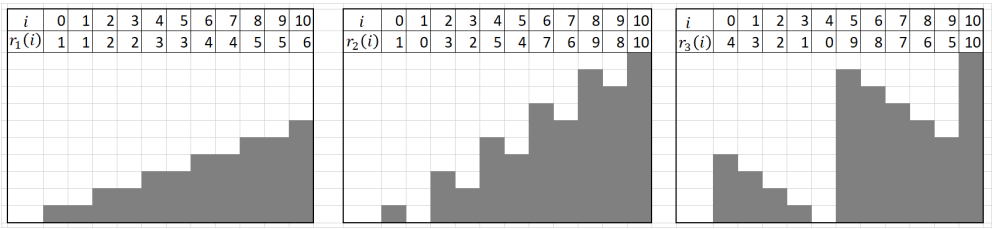


Figure 2.4: The ruggedness functions r_1, r_2, and r_3.

and interchanges the two values in each block. When s is odd, the value 0 forms its own block with $r_1(0) = 0$. Similarly, r_3 divides the set of possible function values in blocks of size 5 (starting at $s-1$ and going in inverse direction), and reverses the order of function values in each block.

Summarizing all these different setups, the functions F4-F17 are defined as follows:

F4: $\textsc{OneMax} + W(\lfloor n/2 \rfloor, 1, 1, \mathrm{id})$
F5: $\textsc{OneMax} + W(\lfloor 0.9n \rfloor, 1, 1, \mathrm{id})$
F6: $\textsc{OneMax} + W(n, u = 3, 1, \mathrm{id})$
F7: $\textsc{OneMax} + W(n, 1, \nu = 4, \mathrm{id})$
F8: $\textsc{OneMax} + W(n, 1, 1, r_1)$
F9: $\textsc{OneMax} + W(n, 1, 1, r_2)$
F10: $\textsc{OneMax} + W(n, 1, 1, r_3)$

F11: $\textsc{LeadingOnes} + W(\lfloor n/2 \rfloor, 1, 1, \mathrm{id})$
F12: $\textsc{LeadingOnes} + W(\lfloor 0.9n \rfloor, 1, 1, \mathrm{id})$
F13: $\textsc{LeadingOnes} + W(n, u = 3, 1, \mathrm{id})$
F14: $\textsc{LeadingOnes} + W(n, 1, \nu = 4, \mathrm{id})$
F15: $\textsc{LeadingOnes} + W(n, 1, 1, r_1)$
F16: $\textsc{LeadingOnes} + W(n, 1, 1, r_2)$
F17: $\textsc{LeadingOnes} + W(n, 1, 1, r_3)$

W-model vs. Unbiasedness Transformations and Fitness Scaling

To avoid confusion, we clarify the sequence of the transformations of the W-model and the unbiasedness and fitness value transformations discussed in Section 2.5.1. Both the re-ordering of the string by the permutation σ and the XOR with a fixed string $z \in \{0,1\}^n$ are executed *before* the transformations of the W-model are applied, while the multiplicative and additive scaling of the function values is applied to the result *after* the fitness perturbation of the W-model.

Example: Assume that the instance is generated from a base problem $f : \{0,1\}^n \to \mathbb{R}$, that the unbiasedness transformations are defined by a permutation $\sigma : [n] \to [n]$ and the string $z \in \{0,1\}^n$, the fitness scaling by a multiplicative scalar $b > 0$ and an additive term $a \in \mathbb{R}$. Assume further that the W-model transformations are defined by the vector $(i_1, \ldots, i_m, u, \nu, r)$. For each queried search point $x \in \{0,1\}^n$, the algorithm receives the function value $af(W(\sigma(x) \oplus z)) + b$, where $\sigma(x) = (x_{\sigma(1)}, \ldots, x_{\sigma(n)})$ and $W : \{0,1\}^n \to \mathbb{R}$ denotes the function that maps each

string to the fitness value defined via the W-transformations $(i_1, \ldots, i_m, u, \nu, r)$.

2.5.7 F18: Low Autocorrelation Binary Sequences

Obtaining binary sequences possessing a high merit factor, also known as the Low-Autocorrelation Binary Sequence (LABS) problem, constitutes a grand combinatorial challenge with practical applications in radar engineering and measurements [138, 126]. It also carries several open questions concerning its mathematical nature. Given a sequence of length n, $S = (s_1, \ldots, s_n)$ with $s_i \in \{-1, +1\}$, the merit factor is proportional to the reciprocal of the sequence's autocorrelation. The LABS optimization problem is defined as searching over the sequence space to yield the maximum merit factor: $\frac{n^2}{2E(S)}$ with $E(S) = \sum_{k=1}^{n-1} \left(\sum_{i=1}^{n-k} s_i \cdot s_{i+k} \right)^2$. This hard, non-linear problem has been studied over several decades (see, e.g., [116, 125]), where the only way to obtain exact solutions remains exhaustive search. As a pseudo-Boolean function over $\{0,1\}^n$, it can be rewritten as follows:

$$F_{\text{LABS}}(\vec{x}) = \frac{n^2}{2 \sum_{k=1}^{n-1} \left(\sum_{i=1}^{n-k} x'_i \cdot x'_{i+k} \right)^2} \quad \text{where } x'_i = 2x_i - 1. \tag{2.6}$$

2.5.8 F19-F21: The Ising Model

The Ising Spin Glass model [11] arose in solid-state physics and statistical mechanics, aiming to describe simple interactions within many-particle systems. The classical Ising model considers a set of spins placed on a regular lattice, where each edge $\langle i, j \rangle$ is associated with an interaction strength $J_{i,j}$. In essence, a problem-instance is defined upon setting up the coupling matrix $\{J_{i,j}\}$. Each spin directs *up* or *down*, associated with a value ± 1, and a set of n spin glasses is said to form a configuration, denoted as $S = (s_1, \ldots, s_n) \in \{-1, +1\}^n$. The configuration's energy function is described by the system's Hamiltonian, as a quadratic function of those n spin variables: $-\sum_{i<j} J_{i,j} s_i s_j - \sum_{i=1}^{n} h_i s_i$, where h_i is an external magnetic field. The optimization problem of interest is the study of the minimal energy configurations, which are termed *ground states*, on a final lattice. This is clearly a challenging combinatorial optimization problem, which is known to be NP-hard, and to hold connections with all other NP problems [113]. EAs have been investigated concerning the impact of their operators for the Ising model, yielding some theoretical results on certain graph instances (see, e.g., [22, 65, 141]).

We have selected and integrated three Ising model instances in IOHprofiler,

assuming zero external magnetic fields, and applying *periodic boundary conditions* (PBC). In order to formally define the Ising objective functions, we adopt a strict graph perspective, where $G = (V, E)$ is undirected and $V = [n]$. We apply an affine transformation $\{-1, +1\}^n \rightsquigarrow \{0, 1\}^n$, where the n spins become binary decision variables (this could be interpreted, e.g., as a coloring problem [141]). A generalized, compact form for the quadratic objective function is now obtained:

$$F_{\text{Ising}}(\vec{x}) = \sum_{\{u,v\} \in E} \left[x_u x_v + (1 - x_u)(1 - x_v) \right], \tag{2.7}$$

thus leaving the instance definition within G.

In what follows, we specify their underlying graphs, whose edges are equally weighted as unity, to obtain their objective functions using (2.7).

F19: The Ring (1D)

This basic Ising model is defined over a one-dimensional lattice. The objective function follows (2.7) using the following graph:

$$\begin{aligned} & G_{\text{Is1D}}: \\ & e_{ij} = 1 \quad \Leftrightarrow \quad j = i + 1 \quad \forall i \in \{1, \ldots, n-1\} \\ & \qquad\qquad \vee \quad j = n, i = 1 \end{aligned} \tag{2.8}$$

F20: The Torus (2D)

This instance is defined on a two-dimensional lattice of size N, using altogether $n = N^2$ vertices, denoted as (i, j), $0 \leq i, j \leq N-1$ [22]. Since PBC are applied, a *regular graph with each vertex having exactly four neighbors* is obtained. The objective function follows (2.7) using the following graph:

$$\begin{aligned} & G_{\text{Is2D}}: \\ e_{(i,j)(k,\ell)} = 1 \quad \Leftrightarrow \quad & [k = (i+1) \bmod N \quad \wedge \quad \ell = j \quad \forall i, j \in \{0, \ldots, N-1\}] \\ \vee \quad & [k = (i-1) \bmod N \quad \wedge \quad \ell = j \quad \forall i, j \in \{0, \ldots, N-1\}] \\ \vee \quad & [\ell = (j+1) \bmod N \quad \wedge \quad k = i \quad \forall j, i \in \{0, \ldots, N-1\}] \\ \vee \quad & [\ell = (j-1) \bmod N \quad \wedge \quad k = i \quad \forall j, i \in \{0, \ldots, N-1\}] \end{aligned} \tag{2.9}$$

F21: Triangular (Isometric 2D Grid)

This instance is also defined on a two-dimensional lattice, yet constructed on an isometric grid (also known as triangular grid), *whose unit vectors form an angle of* $\frac{2\pi}{3}$ [115]. The vertices are placed on integer-valued two-dimensional $n = N^2$ vertices, denoted as (i, j), $0 \leq i, j \leq N-1$, yielding altogether a regular graph whose vertices have exactly six neighbors each (due to PBC):

$$
\begin{aligned}
& G_{\mathrm{IsTR}}: \\
e_{(i,j)(k,\ell)} = 1 \quad \Leftrightarrow \quad & [k = (i+1) \bmod N \ \wedge \ \ell = j \quad \forall i, j \in \{0, \ldots, N-1\}] \\
\vee \quad & [k = (i-1) \bmod N \ \wedge \ \ell = j \quad \forall i, j \in \{0, \ldots, N-1\}] \\
\vee \quad & [\ell = (j+1) \bmod N \ \wedge \ k = i \quad \forall j, i \in \{0, \ldots, N-1\}] \\
\vee \quad & [\ell = (j-1) \bmod N \ \wedge \ k = i \quad \forall j, i \in \{0, \ldots, N-1\}] \\
\vee \quad & [\ell = (j+1) \bmod N \ \wedge \ k = (i+1) \bmod N \ \forall j, i \in \{0, \ldots, N-1\}] \\
\vee \quad & [\ell = (j-1) \bmod N \ \wedge \ k = (i-1) \bmod N \ \forall j, i \in \{0, \ldots, N-1\}]
\end{aligned}
\tag{2.10}
$$

2.5.9 F22: Maximum Independent Vertex Set

Given a graph $G = ([n], E)$, an independent vertex set is a subset of vertices where no two vertices are direct neighbors. A maximum independent vertex set (MIVS) (which generally is not equivalent to a *maximal* independent vertex set) is defined as an independent subset $V' \subset [n]$ having the largest possible size. Using the standard binary encoding $V' = \{i \in [n] \mid x_i = 1\}$, MIVS can be formulated as the maximization of the function

$$
F_{\mathrm{MIVS}}(x) = \sum_i x_i - n \cdot \sum_{i,j} x_i x_j e_{i,j}, \tag{2.11}
$$

where $e_{i,j} = 1$ if $\{i, j\} \in E$ and $e_{i,j} = 0$ otherwise.

In particular, following [6], we consider a specific, scalable problem instance, defining its Boolean graph as follows:

$$
\begin{aligned}
e_{ij} = 1 \quad \Leftrightarrow \quad & j = i + 1 \quad \forall i \in \{1, \ldots, n-1\} - \{n/2\} \\
\vee \quad & j = i + n/2 + 1 \quad \forall i \in \{1, \ldots, n/2 - 1\} \\
\vee \quad & j = i + n/2 - 1 \quad \forall i \in \{2, \ldots, n/2\}.
\end{aligned}
\tag{2.12}
$$

The resulting graph has a simple, standard structure as shown in Figure 2.5 for $n = 10$. The global optimizer has an objective function value of $|V'| = n/2+1$ for this standard

graph. Notably, $n \geq 4$ and n is required to be even; given an odd n, we identify the n-dimensional problem with the $n-1$-dimensional instance.

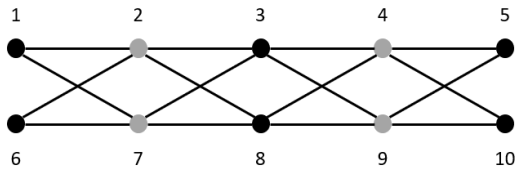


Figure 2.5: A scalable maximum independent set problem, with $n = 10$ vertices and the optimal solution of size 6 marked by the black vertices.

2.5.10 F23: N-Queens Problem

The N-queens problem (NQP) [15] is defined as the task to place N queens on an $N \times N$ chessboard in such a way that they cannot *attack* each other.[2] Figure 2.6 provides an illustration for the 8-queens problem. Notably, the *NQP is actually an instance of the MIVS problem* – when considering a graph on which all possible queen-attacks are defined as edges. NQP formally constitutes a *Constraints Satisfaction Problem*, but is posed here as a maximization problem using a binary representation:

$$
\begin{aligned}
&\text{maximize } \sum_{i,j} x_{ij} \\
&\text{subject to:} \\
&\quad \sum_{i} x_{ij} \leq 1 \quad \forall j \in \{1 \ldots, N\} \\
&\quad \sum_{j} x_{ij} \leq 1 \quad \forall i \in \{1 \ldots, N\} \\
&\quad \sum_{j-i=k} x_{ij} \leq 1 \quad \forall k \in \{-N+2, -N+3, \ldots, N-3, N-2\} \\
&\quad \sum_{i+j=\ell} x_{ij} \leq 1 \quad \forall \ell \in \{3, 4, \ldots, 2N-3, 2N-1\} \\
&\quad x_{ij} \in \{0, 1\} \ \forall i, j \in \{1, \ldots, N\}
\end{aligned}
$$

This formulation utilizes $n = N^2$ binary decision variables x_{ij}, which are associated with the chessboard's coordinates, having an origin $(1, 1)$ at the top-left corner. Setting a binary to 1 implies a single queen assignment in that cell. This formulation

[2]The NQP is traced back to the 1848 Bezzel article entitled "Proposal of the Eight Queens Problem"; for a comprehensive list of references we refer the reader to a documentation by W. Kosters at http://liacs.leidenuniv.nl/~kosterswa/nqueens/nqueens_feb2009.pdf.

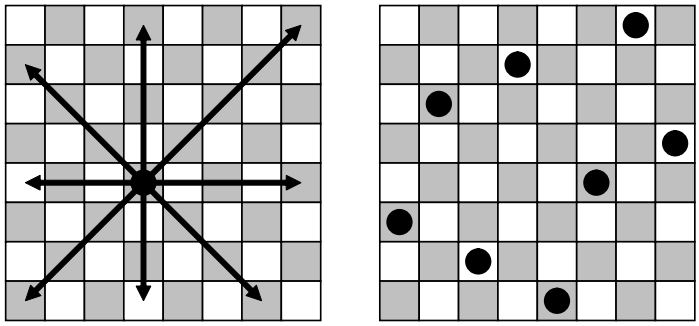


Figure 2.6: The 8-queens problem: [Left] all possible fields a queen can move to from position D4; [Right] a feasible solution.

promotes placement of as many queens as possible by means of the objective function, followed by four sets of constraints eliminating queens' *mutual threats*: the first two sets ensure a single queen on each row and each column, whereas the following two sets ensure a single queen at the increasing-diagonals (using the dummy indexing k) and decreasing-diagonals (using the dummy indexing ℓ). It should be noted that a *permutation formulation* also exists for this problem, and is sometimes attractive for RSHs. Due to chessboard symmetries, NQP possesses multiplicity of optimal solutions. Its attractiveness, however, lies in its hardness. In terms of a black-box objective function, we formulate NQP as the maximization of the following function:

$$F_{\text{NQP}}(\vec{x}) = \sum_{i=1}^{N}\sum_{j=1}^{N} x_{ij} - N \cdot \left(\sum_{i=1}^{N} \max\left\{0, -1 + \sum_{j=1}^{N} x_{ij}\right\} + \sum_{j=1}^{N} \max\left\{0, -1 + \sum_{i=1}^{N} x_{ij}\right\} + \sum_{k=-N+2}^{N-2} \max\left\{0, -1 + \sum_{\substack{j-i=k \\ i,j\in\{1,2,\ldots,N\}}} x_{ij}\right\} + \sum_{\ell=3}^{2N-1} \max\left\{0, -1 + \sum_{\substack{j+i=\ell \\ i,j\in\{1,2,\ldots,N\}}} x_{ij}\right\} \right) \tag{2.13}$$

2.5.11 F24: Concatenated Trap

Concatenated Trap (CT) is defined by partitioning a bit-string into segments of length k and concatenating $m = n/k$ trap functions that takes each segment as input. The trap function is defined as follows: $f_k^{\text{trap}}(u) = 1$ if the number u of ones satisfies $u = k$ and $f_k^{\text{trap}}(u) = \frac{k-1-u}{k}$ otherwise. We use $k = 5$ in our experiments.

2.5.12 F25: Random NK Landscapes

Random NK landscapes (NKL). The function values are defined as the average of n sub-functions $F_i : [0..2^{k+1} - 1] \to \mathbb{R}, i \in [1..n]$, where each component F_i only takes as input a set of $k \in [0..n - 1]$ bits that are specified by a neighborhood matrix. In this paper, k is set to 1 and entries of the neighbourhood matrix are drawn u.a.r. in $[1..n]$. The function values of F_i's are sampled independently from a uniform distribution on $(0, 1)$.

Chapter 3

The IOHprofiler Benchmarking Software

This chapter introduces our IOHprofiler benchmarking software. Following the motivations discussed in Chapter 1, we introduce in this chapter the functionalities and the accessibilities of the tool.

3.1 Overview

Recall that we plan to create a benchmarking software to perform robust testing of IOHs on a wide range of problems, while many tools have been created for specific sets of problems with different programming designs. An overarching benchmarking pipeline would be highly beneficial for this goal, as it allows for easy transition from the implementation of algorithms to the analysis and comparison of performance data. Therefore, we have developed IOHprofiler, which is a benchmarking software for detailed, highly modular performance analysis of iterative optimization heuristics.

IOHprofiler consists of two main components: **IOHexperimenter**, a module for processing the actual experiments and generating the performance data, and **IOHanalyzer** [156], a post-processing module for compiling detailed statistical evaluations. Figure 3.1 plots the workflow of IOHprofiler. With given benchmark problems (**IOHproblems**) and algorithms (**IOHalgorithms)**, IOHexperimenter generates the output data that can be used for IOHanalyzer. IOHanalyzer can perform performance analyses and visualize algorithms' behaviour. We maintain our

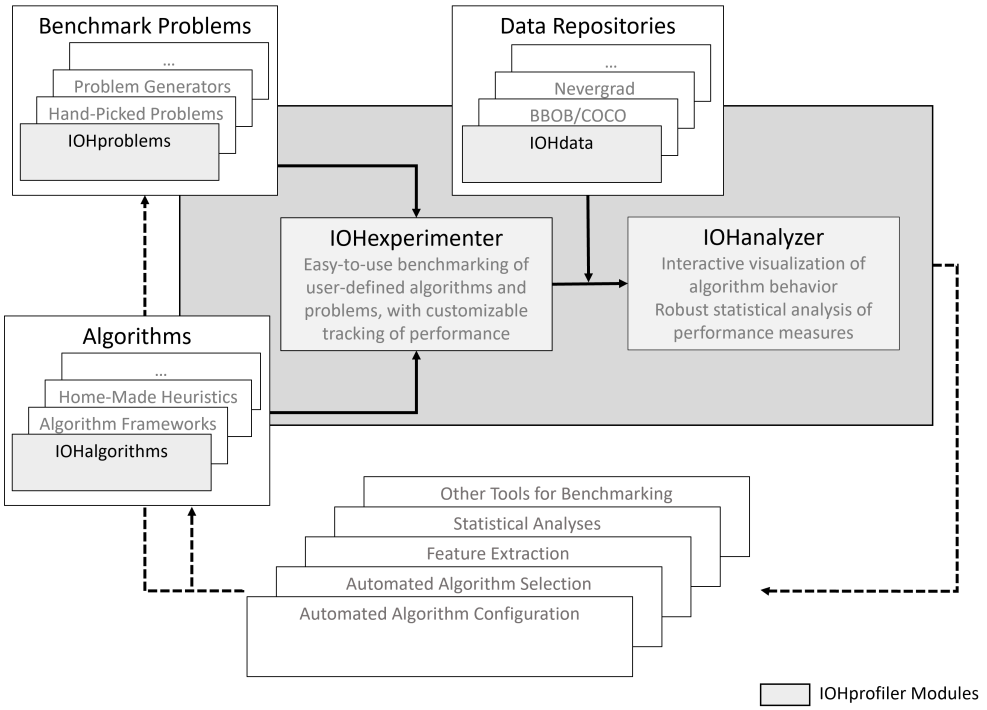


Figure 3.1: Workflow of IOHprofiler

data for the **IOHdata** module. The platform can be applied for the study of automatic algorithm configuration, algorithm selection, feature extraction, statistical analyses, and much more.

We briefly introduce the modules of IOHprofiler in the following:

- **IOHproblems:** a collection of benchmark problems. This component currently comprises (1) the PBO suite of pseudo-Boolean optimization problems suggested in [54], (2) the 24 numerical, noise-free BBOB functions from the COCO platform [78], and (3) the W-model problem generator proposed in [160].
- **IOHalgorithms:** a collection of IOHs. For the moment, the algorithms used for the benchmark studies presented in [3, 35, 54] are available. This subsumes textbook algorithms for pseudo-Boolean optimization, an integration to the object-oriented algorithm design framework ParadisEO [24], and the modular algorithm framework for CMA-ES variants originally suggested in [150] and extended in [35]. Further extensions for both combinatorial and numerical solvers are in progress.

- **IOHdata:** a data repository for benchmark data. This repository currently comprises the data from the experiments performed in [78], a sample data set used in this paper, and some selected data sets from the COCO repository [77]. **IOHdata** also contains performance data from the Nevergrad benchmarking environment [131], which can be fetched from their repository upon request.

- **IOHexperimenter:** the experimentation environment that executes IOHs on **IOHproblems** or external problems and automatically takes care of logging the experimental data. It allows for tracking the internal parameter of IOHs and supports various customizable logging options to specify when to register a data record.

- **IOHanalyzer:** the data analysis and visualization tool presented in this thesis.

3.2 The IOHexperimenter Module

IOHexperimenter is the module of IOHprofiler which can be considered as the interface between algorithms and problems, where it allows consistent data collection of both performance and algorithmic data such as the evolution of control parameters during the optimization process.

3.2.1 Functionalities

We consider here a benchmark process consisting of three components: *problems*, *loggers*, and *algorithms*. While these components interact to perform the benchmarking, they should be usable in a stand-alone manner, allowing any of these factors to be modified without impacting the behaviour of the others. Within IOHexperimenter, an interface is provided to ensure that any changes to the setup will be compatible with the other components of the benchmarking pipeline.

At its core, IOHexperimenter provides a standard interface towards expandable benchmark *problems* and several *loggers* to track the performance and the behaviour (internal parameters and states) of *algorithms* during the optimization process. The logger is integrated into a wide range of existing tools for benchmarking, and we have done such integration work with IOHproblems for discrete optimization and COCO's BBOB [79] for the continuous case. On the *algorithm* side, IOHexperimenter has been connected to several modular algorithm frameworks, such as modular GA (see

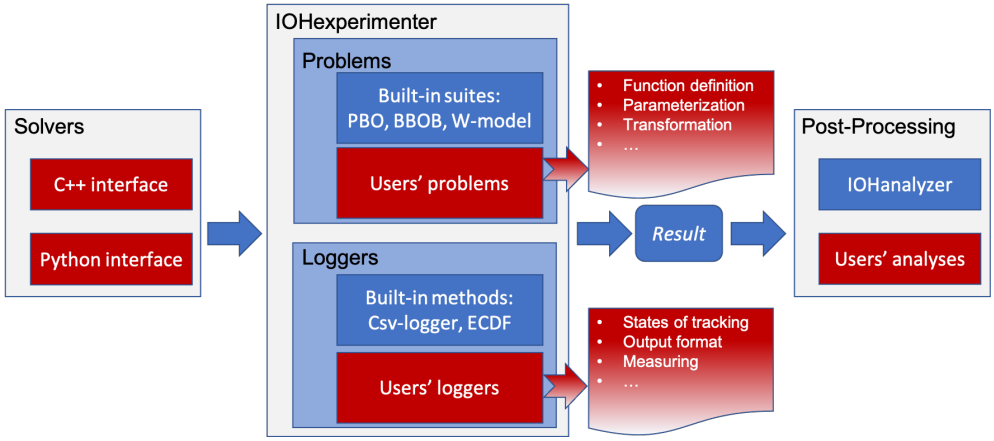


Figure 3.2: Workflow of IOHexperimenter

Chapter 6) and modular CMA-ES [35]. Additionally, output generated by the included *loggers* is compatible with the IOHanalyzer module for interactive performance analysis.

Figure 3.2 shows the way IOHexperimenter can be placed in a typical benchmarking workflow. The key factor here is the flexibility of design: IOHexperimenter can be used with any user-provided solvers and problems given a minimal overhead, and ensures output of experimental results which follow conventional standards. Because of this, the data produced by IOHexperimenter is compatible with post-processing frameworks like IOHanalyzer, enabling an efficient path from algorithm design to performance analysis. In addition to the built-in interfaces to existing software, IOHexperimenter aims to provide an user-accessible way to customize the benchmarking setup. We introduce in the following the typical usage of IOHexperimenter, as well as the ways in which it can be customized to fit different benchmarking scenarios.

3.2.2 Problems

In IOHexperimenter, a problem instance is defined as $P = T_y \circ f \circ T_x$, in which $f : x \to \mathbb{R}$ is a benchmark problem (e.g., for OneMax $x = \{0,1\}^n$ and for the sphere function $x = \mathbb{R}^n$) and T_x and T_y are automorphisms supported on x and $\mathbb{R}$, respectively, representing transformations in the problem's domain and range (e.g., translations and rotations for $x = \mathbb{R}^n$). To generate a problem instance, one needs to specify a tuple of a problem f, an instance identifier $i \in \mathbb{N}_{>0}$, and the dimension n of the

problem. Note that both transformations are applied to generalize the benchmark problem, where the instance id serves as the random seed for instantiating T_x and T_y.

Any problem instance that reconciles with this definition of P, can easily be integrated into IOHexperimenter, using the C++ core or the Python interface.[1]

The transformation methods are particularly important for robust benchmarking, as they allow for the creation of multiple problem instances from the same base-function, which enables checking of invariance properties of algorithms, such as scaling invariance. Built-in transformations for pseudo-Boolean functions are available, as well as transformation methods for continuous optimization used by [79].

When combining several problems together, a problem *suite* can be created. This suite can then be used for more convenient benchmarking by providing access to built-in iterators which allow a solver to easily run on all selected problem instances within the suite. Additionally, an interface to two classes of the W-model extensions (based on the OneMax and LeadingOnes respectively) [160] for generating problems is available.

3.2.3 Data Logging

IOHexperimenter provides *loggers* to track the performance of algorithms during the optimization process. The *loggers* determine which data is recorded and the format to record data. These *loggers* can be tightly coupled with the problems: when evaluating a problem, the attached loggers will be triggered with the relevant information to store. This information will be performance-oriented by default, with customizable levels of granularity, but can also include any algorithm parameters. This can be especially useful for tracking the evolution of self-adaptive parameters in iterative optimization algorithms.

The default logger makes use of a two-part data format: meta-information, such as function id, instance, dimension, etc., that gets written to `.info`-files, while the performance data itself gets written to space-separated `.dat`-files. A full specification of this format can be found in [156]. Data in this format can be used directly with the IOHanalyzer for interactive analysis of the recorded performance metrics.

In addition to the built-in loggers, customized logging functionality can be created within IOHexperimenter as well. This can be used to reduce the footprint of the data when doing massive experiments such as algorithm configuration, where only the final performance measure is relevant [3].

[1] Note that multi-objective problems do not follow this structure, and are not yet supported within IOHexperimenter. Integration of both noisy and mixed-variable type objective functions is in development.

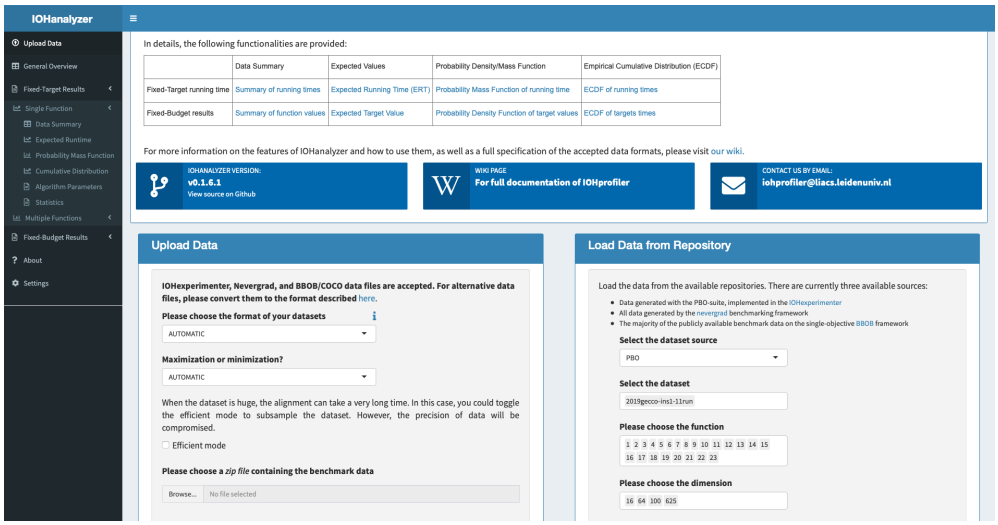


Figure 3.3: The screenshot of the first page of IOHanalyzer.

3.2.4 Accessibility

Note that IOHexperimenter is build in C++, with a direct interface to Python. A more low-level technical documentation of these procedures in both C++ and Python can be found on the IOHprofiler wiki at https://iohprofiler.github.io/. This wiki also provides access to getting-started information about installation and basic usage of IOHexperimenter and its place in the benchmarking pipeline.

3.3 The IOHanalyzer Module

As the post-processing module of iterative optimization heuristic, IOHanalyzer provides detailed statistics about fixed-target running times and fixed-budget performance of the benchmarked algorithms on real-valued, single-objective optimization tasks. Moreover, performance aggregation over several benchmark problems is possible, for example, in the form of ECDFs. Key advantages of IOHanalyzer over other performance analysis packages are its highly interactive design, which allows users to specify the performance measures, ranges, and granularity that are most useful for their experiments, and the possibility to analyze not only performance traces but also the evolution of dynamic state parameters.

Figure 3.3 shows the first page of IOHanalyzer, where presents the general information of IOHanalyzer. Users can upload data in the box frame on the left or/and

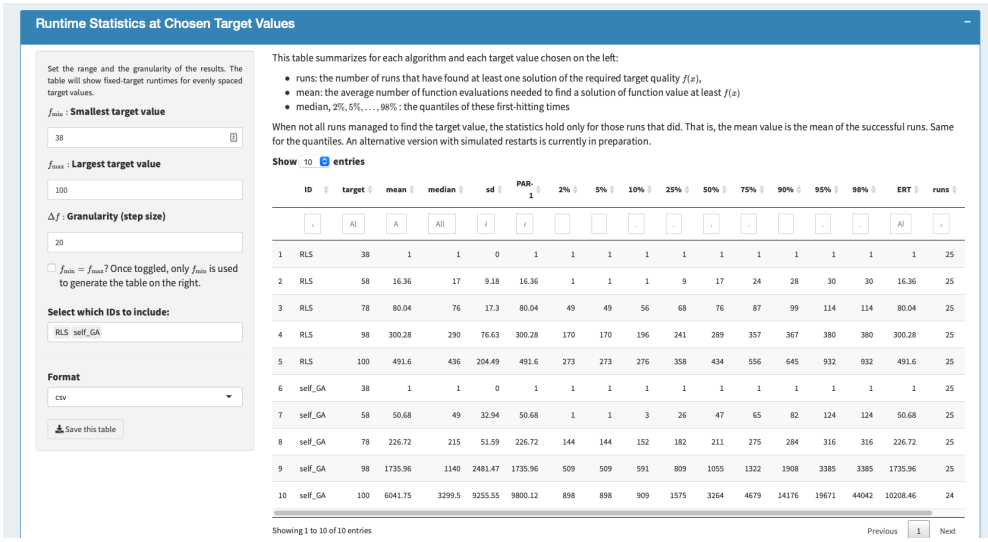


Figure 3.4: The screenshot of the data summary of fixed-target results for the RLS and the self_GA. The table in the figure lists the runtime of the algorithms for each target chosen on the left box.

load IOHdata from the box frame on the right for analyzing. Figure 2.1, Figure 2.2, and Figure 2.3 in Chapter 2 present the plots of fixed-targets results, fixed-budget results, and ECDF curves generated using IOHanalyzer. Apart from these plots, IOHanalyzer also provides detailed data in tables. For example, Figure 3.4 shows a screenshot of a table summarizing runtime for each algorithm and each chosen target.

Moreover, IOHanalyzer supports analyzing the values of parameters of algorithms for the chosen moments (e.g., when a required target is found or the predefined budget is used). An example is plotted in Figure 3.5.

Note that the data tables and the figures on the IOHanalyzer website can be downloaded.

3.3.1 Accessibility

In addition to the web-based application at https://iohanalyzer.liacs.nl, IOHanalyzer is available on GitHub https://github.com/IOHprofiler/IOHanalyzer and CRAN. It is implemented by R and C++. IOHanalyzer can directly process performance data from the main benchmarking platforms, including the COCO platform, Nevergrad, and our own IOHexperimenter. An R programming interface is provided for users preferring to have a finer control over the implemented functionalities. More

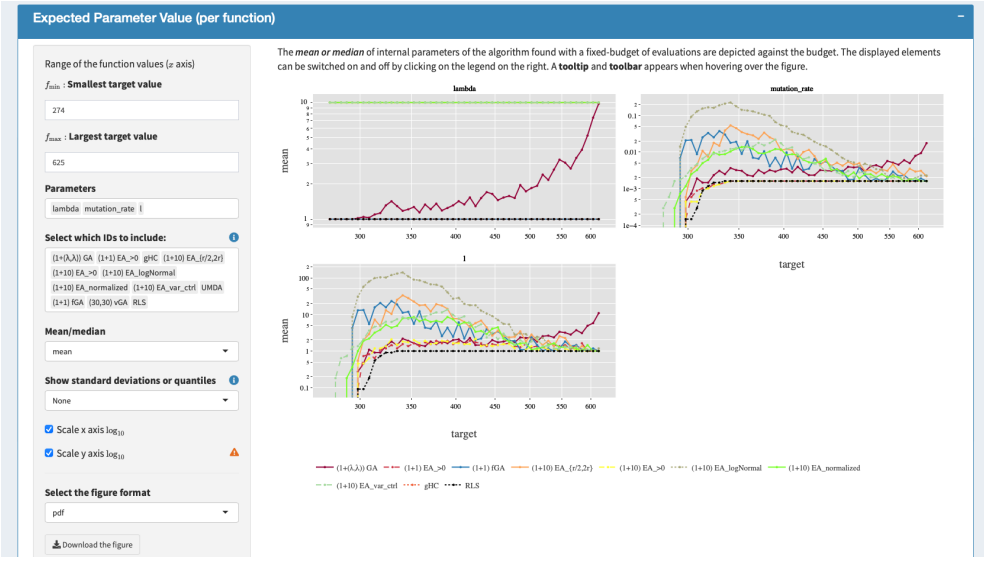


Figure 3.5: The screenshot of the plots of the mean of parameters values for the ten algorithms. The figures plot the mean of internal parameters 'lambda', 'mutation rate', and 'l' of the algorithms for each required target.

details of IOHanalyzer can be found in [156].

Chapter 4

Problem Specific Benchmarking: Study on OneMax and LeadingOnes

In this chapter, we focus on the two classic problems OneMax and LeadingOnes, to show how benchmarking can benefit theoreticians and practitioners for discussing new research directions. We start our investigation on the $(1+\lambda)$ EAs comparing the theoretical results and empirical performance. A *normalized standard bit mutation* is proposed based on benchmarking an existing self-adaption of mutation rate for the $(1+\lambda)$ EAs. In addition, we study the impact of the population size and the mutation rate for the $(1+\lambda)$ EAs and the crossover probability for the $(\mu+\lambda)$ GA.

4.1 Profiling $(1+\lambda)$ EA

4.1.1 Background

Two fundamental building blocks of evolutionary algorithms are global variation operators and populations. *Global variation operators* are sampling strategies that are characterized by the property that every possible solution candidate has a positive probability of being sampled within a short time window, regardless of the current state of the algorithm. *Standard bit mutation* is an example of a global mutation operator. From a given input string $x \in \{0,1\}^n$, standard bit mutation creates an

offspring by flipping each bit in x with some positive probability $0 < p < 1$, with independent decisions for each bit. For any x, y the probability to sample y from x is thus $p^{H(x,y)}(1-p)^{n-H(x,y)}$, where $H(x,y)$ is the number of bits in which x and y differ (*Hamming distance*). This probability is positive even for search points that are very far apart. The motivation to use global sampling strategies is to overcome local optima by eventually performing a sufficiently large jump.

Storing information about the optimization process, maintaining a diverse set of reasonably good solutions, and gathering a more complete picture about the structure of the problem at hand are among the most important reasons to employ *population-based EAs*. The first two objectives are served by the *parent population*, which is the subset of previously evaluated search points that are kept in the memory of the algorithm. The parent population is updated after each generation. New solution candidates are sampled from it through the use of variation operators. These points form the *offspring population* of the generation. Non-trivial offspring population sizes address the desire to gather more information about the fitness landscape before making any decision about which of the points from the parent and offspring population to keep in the memory for the next iteration.

It is very well understood that both the size of the parent population as well as the size of the offspring population can have a significant impact on the performance. Finding suitable parameter values for these two quantities remains to be a challenging problem in practical applications of EAs. From an analytical point of view, populations increase the complexity of the optimization process considerably, as they introduce a lot of dependencies that need to be taken care of in the mathematical analysis. It is therefore not surprising that only few theoretical works on population-based EAs exist, cf. [105] and references mentioned therein. Most existing theoretical works regard algorithms with non-trivial *offspring population* sizes, while the impact of the *parent population* size has received much less interest.

We present below empirical and theoretical results for the $(1+\lambda)$ EA, the arguably simplest EA that combines a global sampling technique with a non-trivial offspring population size.

4.1.2 Algorithms

As noted in [128] there exists an important discrepancy between the algorithms classically regarded in the theory of evolutionary computation literature and their common implementations in practice. For mutation-based algorithms like $(\mu + \lambda)$ and

Algorithm 1: flip_ℓ chooses ℓ different positions and flips the entries in these positions.

Input: $x = (x_1 \dots x_n) \in \{0,1\}^n$, $\ell \in \mathbb{N}$;
$y \leftarrow x$;
Select ℓ pairwise different positions $i_1, \dots, i_\ell \in [n]$ u.a.r.;
for $j = 1, \dots, \ell$ **do** $y_{i_j} \leftarrow 1 - x_{i_j}$;

(μ, λ) EAs, this discrepancy concerns the way new solution candidates are sampled from previously evaluated ones, and how the function evaluations are counted. Both algorithms use the above-described standard bit mutation as only variation operator. An often recommended value for the mutation rate p is $1/n$, which corresponds to flipping exactly one bit on average, an often desirable behavior when the search converges.

When implementing standard bit mutation, it would be rather inefficient to decide for each $i \in [n]$ whether or not the i-th bit of x should be flipped. Luckily, this is not needed, as we can simply observe that standard bit mutation can be equally expressed as drawing a random number ℓ from the binomial distribution $\text{Bin}(n, p)$ with n trials and success probability p and then flipping ℓ bits that are sampled from $[n]$ uniformly at random (u.a.r.) and without replacement. This latter operation is formalized by the flip_ℓ operator in Algorithm 1. We refer to ℓ as the *mutation strength* or the *step size*, while we call p the *mutation rate*.

Analyzing standard bit mutation, we easily observe that the probability to not flip any bit at all equals $(1-p)^n$, which for $p = 1/n$ converges to $1/\exp(1)$. That is, in about 36.8% of calls to this operator, a copy of the input is returned. For the $(1+\lambda)$ EA there is no benefit of evaluating such a copy (unless facing a dynamic or noisy optimization setting), since it applies *plus selection*, where both the parent as well as the offspring can be selected to "survive" for the next generation. It is therefore advisable to change the probability distribution from which the mutation strength ℓ is sampled. A straightforward (and commonly used) idea is to simply re-sample ℓ from $\text{Bin}(n, p)$ until a non-zero value is returned. This approach corresponds to distributing the probability mass $(1-p)^n$ of sampling a zero proportionally to all step sizes $\ell > 0$. This gives the conditional binomial distribution $\text{Bin}_{>0}(n, p)$, which assigns to each $\ell \in \mathbb{N}$ a probability of $\binom{n}{\ell} p^\ell (1-p)^{n-\ell} / (1-(1-p)^n)$. All our empirical results use this conditional sampling strategy. The results can therefore differ significantly from figures previously published in the theory of EA literature [91, 128].

Algorithm 2: The $(1+\lambda)$ $\text{EA}_{>0}$ with mutation rate $p \in (0,1)$ for the maximization of $f : \{0,1\}^n \to \mathbb{R}$

Initialization: Sample $x \in \{0,1\}^n$ u.a.r.;
Optimization: **for** $t = 1, 2, 3, \ldots$ **do**
  for $i = 1, \ldots, \lambda$ **do**
    Sample $\ell^{(i)}$ from $\text{Bin}_{>0}(n,p)$;
    $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$;
  Sample x from $\arg\max\{f(x), f(y^{(1)}), \ldots, f(y^{(\lambda)})\}$ u.a.r.;

The Basic $(1+\lambda)$ $\text{EA}_{>0}$ The $(1+\lambda)$ EA samples λ offspring in every iteration, from which only the best one survives (ties broken uniformly at random). Each offspring is created by standard bit mutation. Following our discussion above, we make use of the re-sampling strategy described above, and obtain the $(1+\lambda)$ $\text{EA}_{>0}$, which we summarize in Algorithm 2.

Adaptive $(1+\lambda)$ $\text{EA}_{>0}$ The $(1+\lambda)$ $\text{EA}_{>0}$ has two parameters: the offspring population size λ and the mutation rate p. Common implementations of the $(1+\lambda)$ $\text{EA}_{>0}$ use the same population size λ and the same mutation rate p throughout the whole optimization process (*static parameter choice*), while the use of *dynamic parameter values* is much less established. A few works exist, nevertheless, that propose to *control* the parameters of the $(1+\lambda)$ EA online [92]. We focus in our empirical comparison on algorithms that have a mathematical support. These are summarized in the following two subsections.

Adaptive Mutation Rates One of the few works that experiments with a *non-static mutation rate* for the $(1+\lambda)$ EA was presented in [46]. The there-suggested algorithm stores a parameter r that is adjusted online. In each iteration, the $(1+\lambda)$ $\text{EA}_{r/2,2r}$ creates $\lambda/2$ offspring by standard bit mutation with mutation rate $r/(2n)$, and it creates $\lambda/2$ offspring with mutation rate $2r/n$. The value of r is updated after each iteration. With probability $1/2$ it is set to the value that the best offspring individual of the last iteration has been created with (ties broken at random), and it is replaced by either $r/2$ or $2r$ otherwise (unbiased random decision). Finally, the value r is capped at 2 if smaller, and at $n/4$, if it exceeds this value. In our experiments, we use $r = 2$ as initial value.

In [46] it is shown analytically that the $(1+\lambda)$ $\text{EA}_{r/2,2r}$ yields an asymptotically optimal runtime on OneMax. This performance is strictly better than what any static

mutation rate can achieve, cf. Section 4.1.3. How well the adaptive scheme works for other benchmark problems is left as an open question in [46].

Adaptive Population Sizes Apart from the mutation rate, one can also consider to *adjust the offspring population size* λ. This is a much more prominent problem, because λ is an explicit parameter, while the mutation rate is often not specified (and thus by default assumed to be $1/n$).

In the theory of EC literature, the following three success-based update rules have been studied. In [87] the offspring population size λ is initialized as one. After each iteration, we count the number s of offspring that are at least as good as the parent. When $s = 0$, we double the population size, and we replace it by $\lfloor \lambda/s \rfloor$ otherwise. For brevity, we call this algorithm the $(1 + \{2\lambda, \lambda/s\})$ EA, and its resampling variant the $(1 + \{2\lambda, \lambda/s\})$ $\text{EA}_{>0}$.

Two similar schemes were studied in [104] where λ is doubled if no strictly better search point has been identified and either set to one or to $\max\{1, \lfloor \lambda/2 \rfloor\}$ otherwise. We regard here the resampling variants of these algorithms, which we call the $(1 + \{2\lambda, 1\})$ $\text{EA}_{>0}$ and the $(1 + \{2\lambda, \lambda/2\})$ $\text{EA}_{>0}$, respectively.

4.1.3 Profiling on OneMax

We recall that the class of OneMax functions is the generalization of the function OM that assigns to each bit string x the number $|\{i \in [n] \mid x_i = 1\}|$ of ones in it. For this generalization, OM is composed with all possible XOR operations on the hypercube. More precisely, for any bit string $z \in \{0,1\}^n$ we define the function $\text{OM}_z : \{0,1\}^n \to [0..n], x \mapsto |\{i \in [n] \mid x_i = z_i\}|$, the number of bits in which x and z agree. The OneMax problem is the collection of all functions OM_z, $z \in \{0,1\}^n$.

Theoretical Bounds OneMax is often referred to as the *drosophila of EC*. It is therefore not surprising that among all benchmark functions, OneMax is the problem for which most runtime results are available. We summarize in this section a few selected results.

Concerning the $(1 + \lambda)$ EA, the first question that one might ask is whether or not it can be beneficial to generate more than one offspring per iteration. When using the number of function evaluations (and not the number of generations) as performance measure, intuitively, it should always be better to create the offspring sequentially, to profit from intermediate fitness gains. This intuition has been formally proven in [87], where it is shown that for all $\lambda, k \in \mathbb{N}$ the expected optimization time (i.e., the number

of function evaluations until an optimal solution is queried for the first time) of the $(1+k\lambda)$ EA cannot be better than that of the $(1+\lambda)$ EA. This result implies that $k=1$ is an optimal choice. Note, however, that the number of *generations* needed to find an optimal solution can significantly decrease with increasing λ, so that it can be beneficial even for OneMax to run the $(1+\lambda)$ EA with $\lambda > 1$ when parallel function evaluations are possible.

For $\lambda = 1$ the runtime of the $(1+1)$ EA with **static mutation rate** $p > 0$ is quite well understood, cf. [163] for a detailed discussion. For general λ and static mutation rate $p = c/n$ (where here and henceforth $c > 0$ is assumed to be constant), the expected optimization time is $(1 \pm o(1))\big(\frac{n\lambda \ln\ln\lambda}{2\ln\lambda} + \frac{e^c}{c} n \ln n\big)$ [68]. An interesting observation made in [46] reveals that the parametrization $p = c/n$ is suboptimal: with $p = \ln(\lambda)/(2n)$ the $(1+\lambda)$ EA needs only an expected number of $O(n\lambda/\log(\lambda) + \sqrt{\lambda} n \log n)$ function evaluations to optimize OneMax [46] (the proof requires $\lambda \geq 45$ and $\lambda = n^{O(1)}$). We call this algorithm the $(1+\lambda)$ $\mathrm{EA}_{p=\ln(\lambda)/(2n)}$.

When using **non-static mutation rates**, the best expected optimization time that a $(1+\lambda)$ EA can achieve on OneMax is bounded from below by $\Omega(n\lambda/\log(\lambda) + n\log n)$ [8]. This bound is attained by a $(1+\lambda)$ EA variant with fitness-dependent mutation rates [8]. Interestingly, it is also achieved by the self-adjusting $(1+\lambda)$ $\mathrm{EA}_{r/2,2r}$ described in Section 5.1.2 (the proof requires again $\lambda \geq 45$ and $\lambda = n^{O(1)}$).

Concerning the algorithms using **non-static offspring population sizes** (see Section 5.1.2), we do not have an explicit theoretical analysis for the $(1+\{2\lambda, \lambda/s\})$ EA, but it is known that the expected optimization time of both the $(1+\{2\lambda, 1\})$ EA and the $(1+\{2\lambda, \lambda/2\})$ EA is $O(n \log n)$ [104].

Disclaimer. It is important to note that all the bounds reported above (and those mentioned in Section 4.1.4) hold, a priori, only for the classical $(1+\lambda)$ EA variants, not the resampling versions regarded here in this thesis. For most bounds, and in particular the ones with static parameter choices, it is, however, not difficult to prove that the modifications do not change the asymptotic order of the expected optimization times. What does change, however, is the leading constant. As a rule of thumb, runtime bounds for the $(1+\lambda)$ EA decrease by a multiplicative factor of about $1-(1-p)^n$ when the mutation strengths are sampled from the conditional binomial distribution $\mathrm{Bin}_{>0}(n,p)$.

Note also that in our summary we collect only statements about the total expected *optimization time*, i.e., AHT. Here again the $(1+1)$ EA and the $(1+\lambda)$ EA with static parameters form an exception as for these two algorithms a few theoretical fixed-budget results exist, cf. [45, 108, 122] and references therein.

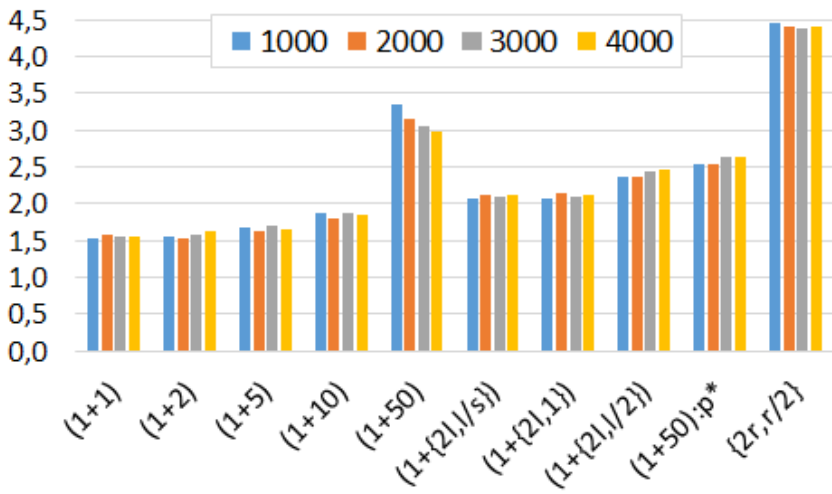


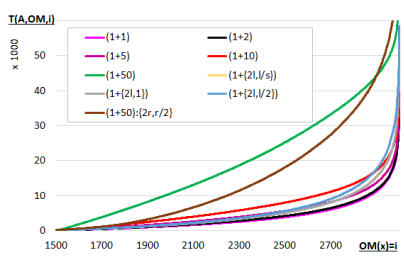
Figure 4.1: Average optimization times for 100 independent runs of the $(1+\lambda)$ $\text{EA}_{>0}$ variants for OneMax, normalized by $n \ln n$. (Initial) population size for the adaptive variants is 50. $p^* = \ln(\lambda)/(2n)$

Empirical Evaluation We now come to the results of our empirical investigation. All figures presented in this section are averages over 100 independent runs. This might look like a small number, but we recall that we track the whole optimization process, to present the fixed-target results below.

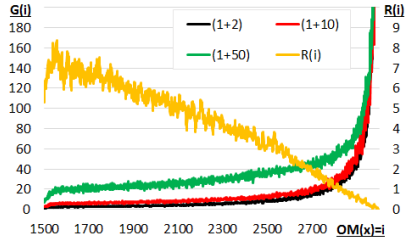
We have seen above that, for reasonable parameter settings, the average optimization times of the $(1+\lambda)$ EA variants are all of order $n \log n$. We therefore normalize the empirical averages in Figure 4.1 by this factor.

We observe that the normalized averages are quite stable across the dimensions, with an exception of the $(1+50)$ EA, whose relative performance improves with increasing problem dimension. We also see that the $(1+50)$ $\text{EA}_{>0}$ variant with $p^* = \ln(\lambda)/(2n)$ achieves a better optimization time than the $(1+50)$ $\text{EA}_{>0}$ with $p = 1/n$. The variants with adaptive offspring population size perform significantly worse than the $(1+1)$ $\text{EA}_{>0}$, which is not surprising given the performance hierarchy of the $(1+\lambda)$ EAs mentioned in the beginning of Section 4.1.3.

For the tested problem dimensions, the worst-performing algorithm in our comparison is the $(1+\lambda)$ $\text{EA}_{r/2,2r}$. This may come as a surprise since this algorithm is the one with the best theoretical support. The advantage of this algorithm seems to require much larger problem dimensions, different values of λ, and/or different settings of the hyper-parameters that determined its update mechanism.



(a) Average fixed-target runtimes $T(A, \mathrm{OM}, i)$.



(b) Gradients $G(A, \mathrm{OM}, i)$ and relative disadvantage $R(i)$ of the (1+50) $\mathrm{EA}_{>0}$ over the (1+2) $\mathrm{EA}_{>0}$.

Figure 4.2: Fixed target data for the 3,000-dimensional OneMax problem (averages over 100 runs)

A general question raised by the data in Figure 4.1 concerns the sensitivity of the adaptive $(1+\lambda)$ EA variants with respect to the initialization of their parameters and with respect to their hyper-parameters, which are the update strengths, but also the initial parameter values of λ and r, respectively.

It has been discussed that for OneMax the performance of the $(1+\lambda)$ EA can only be worse than that of the $(1+1)$ EA. In fact, the data in Figure 4.1 demonstrates a quite significant discrepancy between the performance of the (1+1) $\mathrm{EA}_{>0}$ and the (1+λ) $\mathrm{EA}_{>0}$ variants with $\lambda \geq 10$. The expected optimization time of the $(1+50)$ $\mathrm{EA}_{>0}$, for example, is about twice as large as that of the $(1+1)$ $\mathrm{EA}_{>0}$. Intuitively, this can be explained as follows. At the beginning of the OneMax optimization process the probability that a random offspring created by the $(1+\lambda)$ EA improves upon its parent is constant. In this phase the $(1+\lambda)$ EA variants with small λ have an advantage as they can (almost) instantly make use of this progress, while the $(1+\lambda)$ EAs with large λ first need to wait for all λ offspring to be evaluated. Since the expected fitness gain of the best of these λ offspring is not much larger than that of a random individual, large offspring population sizes are detrimental in this first phase of the optimization process. We note, however, that the relative disadvantage of large λ is much smaller towards the end of the optimization process. When, say, the parent individual x satisfies $\mathrm{OM}(x) = n - \Theta(1)$, the probability that a random offspring has a better function value is of order $\Theta(1/n)$ only. We therefore have to create $\Theta(n)$ offspring, in expectation, before we see any progress. The relative disadvantage of creating several offspring in one generation is therefore almost negligible in the later parts of the optimization process (provided that $\lambda = O(n)$). This informal explanation is confirmed by the plots in Figure 4.2, which display for $n = 3,000$

1. in Figure 4.2a: the empirical average fixed-target times $T(A, \mathrm{OM}, i)$; i.e., the average number of function evaluations that the $(1+\lambda)$ $\mathrm{EA}_{>0}$ variant A needs to identify a solution of OM-value at least i. We cap this plot at $60,000$ evaluations.

2. in Figure 4.2b: the gradients $G(A, i) := T(A, \mathrm{OM}, i) - T(A, \mathrm{OM}, i-1)$ (three lowermost curves), and the relative difference $R_i := (\mathrm{avg}_{j=i,i+1,\dots,i+5}(G((1+50)\mathrm{EA}_{>0}, j) - \mathrm{avg}_{j=i,i+1,\dots,i+5} G((1+2)\mathrm{EA}_{>0}, j)/G((1+2)\mathrm{EA}_{>0}, j))$ of the rolling average of the gradients of the $(1+50)$ and the $(1+2)$ $\mathrm{EA}_{>0}$.

Note that the gradient $G(A, i)$ measures the average time needed by algorithm A to make a progress of one when starting in a point x of OneMax-value $\mathrm{OM}(x) = i$. Small gradients are therefore desirable. We see that, for example, the $(1+50)$ $\mathrm{EA}_{>0}$ (green curve) needs, on average, about 20 fitness evaluations to generate a strictly better search point when starting in a solution of OM-value around $1,700$. The $(1+2)$ $\mathrm{EA}_{>0}$, in contrast, needs only about 2.6 function evaluations, on average. The relative disadvantage of the $(1+50)$ $\mathrm{EA}_{>0}$ over the $(1+2)$ $\mathrm{EA}_{>0}$ decreases with increasing function values from around 7 to zero, cf. the uppermost (yellow) curve in Figure 4.2b. We use the rolling average of 5 consecutive values here to obtain a smoother curve for $R(i)$.

Another important insight from Figure 4.2a is that the dominance of the $(1+1)$ $\mathrm{EA}_{>0}$ over all $(1+\lambda)$ $\mathrm{EA}_{>0}$ variants does not only apply to the total expected optimization time, but also to all intermediate target values. This can be shown with mathematical rigor by adjusting the proofs in [87, Section 3] to suboptimal target values.

4.1.4 Profiling on LeadingOnes

We recall that LeadingOnes is the generalization of the function LO, which counts the number of initial ones in the string, i.e., $\mathrm{LO}(x) = \max\{i \in [0..n] \mid \forall j \leq i : x_j = 1\}$. The generalization is by composing with an XOR-shift and a permutation of the positions. This way, we obtain for every $z \in \{0,1\}^n$ and for every permutation (one-to-one map) σ of the set $[n]$ the function $\mathrm{LO}_{z,\sigma} : \{0,1\}^n \to \mathbb{N}$, which assigns to each x the function value $\max\{i \in [0..n] \mid \forall j \in [i] : x_{\sigma(j)} = z_{\sigma(j)}\}$. The LeadingOnes problem is the collection of all these functions.

Theoretical Bounds Also for LeadingOnes it has been proven that the optimal value of the offspring population size λ in the $(1+\lambda)$ EA is one when using function evaluations and not generations as performance indicator [87]. In contrast to

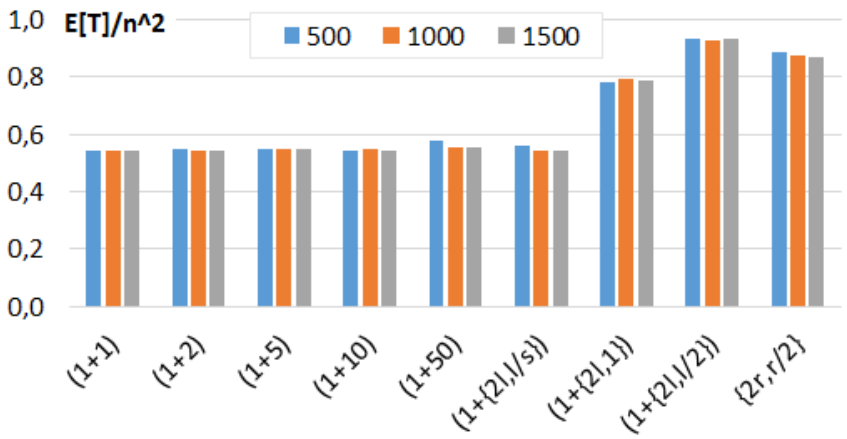


Figure 4.3: Average optimization times for 100 independent runs of the $(1+\lambda)$ $\text{EA}_{>0}$ variants for LeadingOnes, normalized by n^2. (Initial) population size for the adaptive variants is 50.

OneMax, however, we will observe, by empirical and mathematical means, that the disadvantage of non-trivial population sizes is much less pronounced for this problem.

The $(1+1)$ EA with fixed mutation probability p has an expected optimization time of $\frac{1}{2p^2}((1-p)^{-n+1}-(1-p))+1$ on LeadingOnes [21], which is minimized for $p \approx 1.59/n$. This choice gives an expected runtime of about $0.77n^2$. A fitness-dependent mutation rate can decrease this runtime further to around $0.68n^2$ [21]. For the $(1+1)$ $\text{EA}_{>0}$, it has been observed in [91] that its expected optimization time decreases with decreasing p. More precisely, it equals $\frac{1-(1-p)^n}{2p^2}((1-p)^{-n+1}-(1-p))+1$, which converges to $n^2/2+1$ for $p \to 0$.

For $\lambda = n^{O(1)}$, the expected optimization time of the $(1+\lambda)$ EA with mutation rate $p = 1/n$ is $O(n^2+n\lambda)$ [87]. With this mutation rate, the adaptive $(1+\{2\lambda, \lfloor\lambda/2\rfloor\})$ EA and the $(1+\{2\lambda, 1\})$ EA achieve an expected optimization time of $O(n^2)$ [104]. Any $(1+\lambda)$ EA variant with fixed offspring population size λ but possibly adaptive mutation rate p needs at least $\Omega(\frac{\lambda n}{\ln(\lambda/n)}+n^2)$ function evaluations, on average, to optimize LeadingOnes [8].

Recall that the bounds reported above are for the classic $(1+\lambda)$ EA variants, not the resampling versions.

This section presents a refining of the bound for the $(1+\lambda)$ EA, and we first present the empirical results that have motivated this analysis.

Empirical Results Similarly to the data plotted in Figure 4.1, we show in Figure 4.3 the normalized average optimization times of the different algorithms; the normalization factor is n^2.

For all $(1 + \lambda)$ EA variants, a fairly stable performance across the three tested dimensions $n = 500$, $n = 1,000$, and $n = 1,500$ can be observed. We also see that the adaptive algorithms seem to perform worse than the ones with static parameter values; we will address this point in more detail below.

Another interesting observation is that the value of λ does not seem to have a significant impact on the expected performance. This is in sharp contrast to the situation for OneMax, cf. our discussion in the second half of Section 4.1.3. Building on our discussion there, we can explain this phenomenon as follows. Unlike for OneMax, the situation for LeadingOnes is that the expected fitness gain of a random offspring created by a $(1 + \lambda)$ EA variant with static mutation rate $p = c/n$ is very small throughout the whole optimization process. More precisely, it decreases only mildly from around $2c/n$ when $\mathrm{LO}(x) = 0$ to $2c(1-c/n)^{n-1}/n \approx 2c/(e^c n)$ for $\mathrm{LO}(x) = n-1$. Thus, intuitively, the whole optimization process of LeadingOnes is very similar to the last steps of the OneMax optimization.

Figure 4.4 presents the average fixed-target runtimes for selected $(1 + \lambda)$ $\mathrm{EA}_{>0}$ variants on the 1,500-dimensional LeadingOnes problem. We add to this figure the fixed target runtime of Randomized Local Search (RLS), the greedy (1+1)-type hill climber that always flips one random bit per iteration. RLS has a constant expected fitness gain of $2/n$ on LeadingOnes and thus a total expected optimization time of $n^2/2 + 1$.

We observe that the performance of the (1+1) $\mathrm{EA}_{>0}$ and that of the (1+50) $\mathrm{EA}_{>0}$ are indeed very similar throughout the optimization process. The curves for the $(1 + \lambda)$ $\mathrm{EA}_{>0}$ with $\lambda = 2, 5, 10$ were indistinguishable in this plot and are therefore not shown in the figure. We also see that their fixed-target performance is better than that of RLS for all LO-values up to around $1,250 \approx 0.42n$ (exact empirical values are 1,227 for the $(1 + 50)$ $\mathrm{EA}_{>0}$ and 1,283 for the $(1 + 1)$ $\mathrm{EA}_{>0}$, but we recall that such numbers should be taken with care as they represent an average of 100 runs only. It should not be very difficult to compute the cutting point precisely, by mathematical means).

Since the only difference between the $(1 + 1)$ $\mathrm{EA}_{>0}$ and RLS is the distribution from which new offspring are sampled, we see that the $(1 + \lambda)$ EA variants profit from iterations in which more than one bit are flipped in the beginning of the optimization process, while they suffer from this same effect in the later parts. This situation is

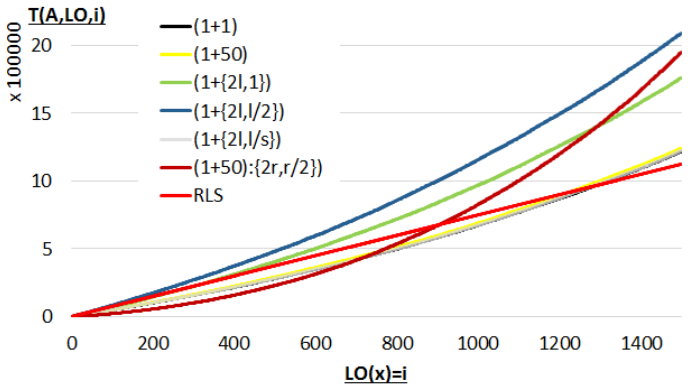


Figure 4.4: Fixed-target runtimes of the $(1 + \lambda)$ $\text{EA}_{>0}$ variants on the $1,500$-dimensional LeadingOnes problem

even more pronounced in the $(1 + \lambda)$ $\text{EA}_{r/2,2r}$, which outperforms all other tested algorithms for target values up to 709. Its performance then suffers from creating at least half of its offspring with a too large mutation rate. Recall that even if the algorithm has correctly identified the optimal mutation rate $p(\text{LO}(x))$, it still creates half of its offspring with mutation rate $2p(\text{LO}(x))$. This results in a mediocre overall performance. This observation certainly raises the question of how to adjust the structure of the $(1 + \lambda)$ $\text{EA}_{r/2,2r}$ to benefit from its good initial performance. From the viewpoint of hyper-heuristics, or algorithm selection, an adaptive selection between the $(1 + \lambda)$ $\text{EA}_{r/2,2r}$ and the $(1 + \lambda)$ $\text{EA}_{>0}$ would be desirable.

If we had looked only at the total optimization times, we would have classified the $(1 + \lambda)$ $\text{EA}_{r/2,2r}$ as being inefficient. The fixed-target results, however, nicely demonstrate that despite the poor overall performance, there is something to be learned from this algorithm. This emphasizes the need for a fine-grained benchmarking environment, similar to what is done in continuous black-box optimization (where fixed-target and fixed-budget considerations are a necessary standard since the algorithms cannot identify an optimal solution but only get arbitrarily close to it).

Precise Bounds for LeadingOnes We have observed that, for LeadingOnes, the runtimes of the $(1 + \lambda)$ $\text{EA}_{>0}$ variants with static parameter choices are very close to that of the $(1 + 1)$ $\text{EA}_{>0}$. Theorem 1 shows that for every constant λ, the expected optimization time of the $(1 + \lambda)$ $\text{EA}_{>0}$ converges from above against that of the $(1+1)$ $\text{EA}_{>0}$ (and the same holds for the $(1+\lambda)$ EA and $(1+1)$ EA, respectively). Theorem 1 can be proven by adjusting the proofs in [21] to the $(1+\lambda)$ EA. An important

ingredient in this analysis is the observation that the probability of making progress in one *generation* with parent individual x equals $1-(1-p(1-p)^{\mathrm{LO}(x)})^\lambda$, i.e., 1 minus the probability that none of the λ offspring is better. Recall that in order to create a better offspring, none of the first $\mathrm{LO}(x)$ bits should flip, while the $(\mathrm{LO}(x)+1)$-st bit *does* need to be flipped. The results for the $(1+\lambda)$ $\mathrm{EA}_{>0}$ can be obtained from that for the $(1+\lambda)$ EA by taking into account that the re-sampling strategy increases the expected fitness gain by a multiplicative factor of $1/(1-(1-p)^n)$. Therefore, we can derive the following:

Theorem 1. *For all $n, \lambda \in \mathbb{N}$ the expected optimization time of the $(1+\lambda)$ EA with static mutation rate $0 < p < 1$ on the n-dimensional* LeadingOnes *function is at most*

$$1 + \frac{\lambda}{2} \sum_{j=0}^{n-1} \frac{1}{1-(1-p(1-p)^j)^\lambda} \tag{4.1}$$

and the expected optimization time of the $(1+\lambda)$ $EA_{>0}$ is at most

$$1 + \frac{(1-(1-p)^n)\lambda}{2} \sum_{j=0}^{n-1} \frac{1}{1-(1-p(1-p)^j)^\lambda}. \tag{4.2}$$

To judge the precision of the bound stated in Theorem 1, we first note that for $\lambda = 1$ expression (4.1) is tight by the result presented in [21] (note though that the additive $+1$ term is suppressed there as they regard the number of iterations, not function evaluations). Similarly, for the $(1+1)$ $\mathrm{EA}_{>0}$ expression (4.2) is tight by the bound proven in [91].

Apart from this case with trivial offspring population size $\lambda = 1$, it might be tedious to compute the expected optimization time of the $(1+\lambda)$ EA on LeadingOnes exactly, since in our proof for Theorem 1 we would have to take into account that in one generation more than one search point that improves upon the current best search point can be generated. Since the $(1+\lambda)$ EA chooses the best one of these, the distribution of this offspring would have to be computed. Note, however, that this effect can only have a very mild impact on the bounds stated above, as it occurs relatively rarely and does, in general, not result in a much larger fitness gain. Put differently, the bounds in Theorem 1 are close to tight for reasonable (i.e., not too large) values of λ.

As the expressions in Theorem 1 are not easy to interpret, we provide in Table 4.1 a numerical evaluation of the upper bound (4.2) for different values of λ and n. We

add to this table (second row) the empirically observed averages, which show a good match to the theoretical bound.

	500	1,000	1,500	10,000	100,000	500,000
(1+1)	54.317%	54.313%	54.311%	54.309%	54.308%	54.308%
emp.	54.0%	54.1%	54.2%	-	-	-
(1+2)	54.349%	54.328%	54.322%	54.310%	54.308%	54.308%
emp.	54.8%	54.4%	54.2%	-	-	-
(1+5)	54.444%	54.376%	54.353%	54.315%	54.309%	54.308%
emp.	54.5%	54.8%	54.6%	-	-	-
(1+50)	55.883%	55.091%	54.829%	54.386%	54.316%	54.310%
emp.	57.6%	55.3%	55.2%	-	-	-

Table 4.1: Theoretical upper bounds from Theorem 1 and empirical optimization times for the $(1 + \lambda)$ $\text{EA}_{>0}$

4.1.5 Summary

The work profiling $(1 + \lambda)$ EA provids an example of how benchmarking studies associate theoretical analysis and empirical studies. In practice, the results shows that the value of λ significantly affects the performance of (1+1) EA for OneMax. However, we did not observe such effect for LeadingOnes. Moreover, the result inspired a refined analysis of the expected optimization time of the $(1 + \lambda)$ EA on LeadingOnes [56].

4.2 Interpolating Local and Global Search by Controlling the Variance of Standard Bit Mutation

Recall that the probability of using the standard bit mutation to sample a specific offspring y at distance $0 \leq d \leq n$ from x thus equals $p^{H(x,y)}(1-p)^{H(x,y)}$, where $H(x, y) = |\{1 \leq i \leq n \mid x_i \neq y_i\}|$ denotes the Hamming distance of x and y. This probability is strictly positive for all y, thus showing that the probability that an EA using standard bit mutation will have sampled a global optimum of f converges to one as the number of iterations increases. In contrast to pure random search, however, the distance at which the offspring y is sampled follows a binomial distribution, $\text{Bin}(n, p)$, and is thus concentrated around its mean np.

The ability to escape local optima comes at the price of frequent uses of non-optimal search radii even in those regimes in which the latter are stable for a long time. The incapability of standard bit mutation to adjust to such situations results in

important performance losses on almost all classical benchmark functions, which often exhibit large parts of the optimization process in which flipping a certain number of bits is required. A convenient way to control the degree of randomness in the choice of the search radius would therefore be highly desirable.

In this section we introduce such an interpolation. It allows to calibrate between deterministic and pure random search, while encompassing standard bit mutation as one specification. More precisely, we investigate *normalized standard bit mutation*, in which the mutation strength (i.e., the search radius) is sampled from a normal distribution $N(\mu, \sigma^2)$. By choosing $\sigma = 0$ one obtains a deterministic choice, and the "degree of randomness" increases with increasing σ. By the central limit theorem, we recover a distribution that is very similar to that of standard bit mutation by setting $\mu = np$ and $\sigma^2 = np(1-p)$.

Apart from conceptual advantages, normalized standard bit mutation offers the advantage of separating the variance from the mean, which makes it easy to control both parameters independently during the optimization process. While multi-dimensional parameter control for discrete EAs is still in its infancy, cf. comments in [92, 42], we demonstrate in this work a simple, yet efficient way to control mean and variance of normalized standard bit mutation.

4.2.1 Background

In Section 4.1, we observed that the EA with success-based self-adjusting mutation rate proposed in [46] outperforms the $(1+\lambda)$ EA for a large range of sub-optimal targets. It then drastically looses performance in the later parts of the optimization process, which results in an overall poor optimization time on OneMax and LeadingOnes functions of moderate problem dimensions $n \leq 10,000$. The proven optimal asymptotic behavior on OneMax in [46] can thus not be observed for these dimensions.

The algorithm from [46], which we named $(1+\lambda)$ $\text{EA}_{r/2,2r}$, has been mentioned in Section 4.1.2. The details are presented in Algorithm 3. It is a $(1+\lambda)$ EA which applies in each iteration two different mutation rates. Half of the offspring population is generated with mutation rate $r/(2n)$, the other half with mutation rate $2r/n$. The parameter r is the current best mutation strength, which is updated after each iteration, with a bias towards the rate by which the best of the λ offspring has been sampled.

Recall that we apply the standard bit mutation by first sampling a radius ℓ from the binomial distribution $\text{Bin}(n,p)$ and then applying the flip_ℓ operator, which flips

Algorithm 3: The 2-rate $(1 + \lambda)$ $\text{EA}_{r/2,2r}$ with adaptive mutation rates proposed in [46]

Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
Initialize $r \leftarrow r^{\text{init}}$; // Following [46] we use $r^{\text{init}} = 2$;
Optimization: **for** $t = 1, 2, 3, \dots$ **do**
  for $i = 1, \dots, \lambda/2$ **do**
    Sample $\ell^{(i)} \sim \text{Bin}_{>0}(n, r/(2n))$, create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$;
  for $i = \lambda/2 + 1, \dots, \lambda$ **do**
    Sample $\ell^{(i)} \sim \text{Bin}_{>0}(n, 2r/n)$, create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$;
  $x^* \leftarrow \arg\max\{f(y^{(1)}), \dots, f(y^{(\lambda)})\}$ (ties broken u.a.r.);
  if $f(x^*) \geq f(x)$ **then** $x \leftarrow x^*$;
  if x^* *has been created with mutation rate* $r/2$ **then** $s \leftarrow 3/4$ **else** $s \leftarrow 1/4$;
  Sample $q \in [0,1]$ u.a.r.;
  if $q \leq s$ **then** $r \leftarrow \max\{r/2, 2\}$ **else** $r \leftarrow \min\{2r, n/4\}$;

ℓ pairwise different bits that are chosen from the index set $[n]$ uniformly at random. Note that we still apply the *re-sampling strategy* enforcing that all offspring differ from their parents by at least one bit, which is achieved by sampling ℓ from the conditional binomial distribution $\text{Bin}_{>0}(n, p)$.

In Section 4.1.2, we compared the fixed-target performance of the $(1 + 50)$ $\text{EA}_{>0}$ (i.e., the $(1 + \lambda)$ EA using the conditional sampling rule introduced above) and the $(1 + 50)$ $\text{EA}_{r/2,2r}$ on OneMax and LeadingOnes. In Figure 4.5 we report similar empirical results for $n = 10,000$ (OneMax) and $n = 2,000$ (LeadingOnes) (the other results in the two figures will be addressed below). We also observed in Section 4.1 that for both functions the $(1+50)$ $\text{EA}_{r/2,2r}$ from [46] performs well for small target values, but drastically looses performance in the later stages of the optimization process.

Properties of OneMax and LeadingOnes

Both OneMax and LeadingOnes have a long period during the optimization run in which flipping one bit is optimal.

For OneMax flipping one bit is widely assumed to be optimal as soon as $f(x) \geq 2n/3$. Quite interestingly, however, this conjecture has not not been rigorously proven to date. It is only known that drift-maximizing (i.e., maximizing the expected fitness gain over the best-so-far individual) mutation strengths are *almost* optimal [45], in

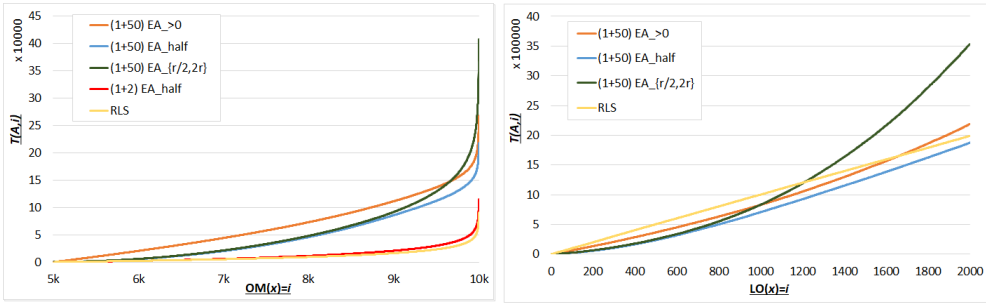


Figure 4.5: Average fixed-target running times for variants of the 2-rate $(1 + 50)$ EA for $10,000$-dimensional OneMax and $2,000$-dimensional LeadingOnes.

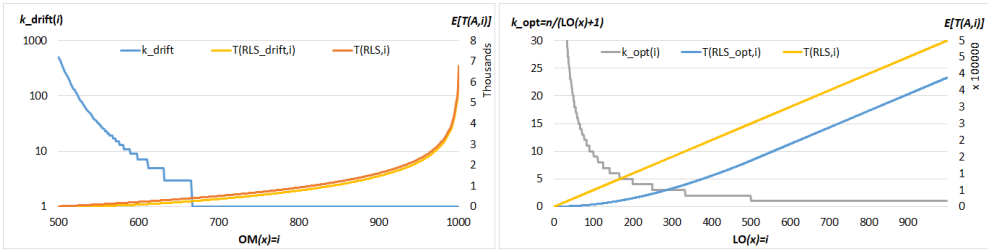


Figure 4.6: Drift maximizing and optimal mutation strength for $1,000$-dimensional OneMax and LeadingOnes functions, respectively. Note the logarithmic scale for k_{drift} for OneMax. For OneMax, RLS spends around 94% of the total optimization time in the regime in which $k_{\text{drift}} = 1$, for LeadingOnes this fraction is still 50%. For the drift-maximizing/optimal RLS-variants flipping in each iteration k_{drift} and k_{opt} bits, respectively, these fractions are around 96% for OneMax and 64% for LeadingOnes.

the sense that the overall expected optimization time of the elitist (1+1) algorithm using these rates in each step cannot be worse than the best-possible unary unbiased algorithm for OneMax by more than an additive $o(n)$ lower order term [45]. But even for the drift maximizer the statement that flipping one bit is optimal when $f(x) \geq 2n/3$ has only be shown for an approximation, not the actual drift maximizer. Numerical evaluations for problem dimensions up to $10,000$ nevertheless confirm that 1-bit flips are optimal when the OneMax-value exceeds $2n/3$.

For LeadingOnes, on the other hand, it is well known that flipping one bit is optimal as soon as $f(x) \geq n/2$ [38].

We display in Figure 4.6, which is adjusted from [53], the optimal and drift-maximizing mutation strength for LeadingOnes and OneMax, respectively. We

also display in the same figure the expected time needed by RLS_{opt} and $\text{RLS}_{\text{drift}}$, the elitist (1+1) algorithm using in each step these mutation rates. We see that these algorithms spend around 96% (for OneMax) and 64% (for LeadingOnes), respectively, of their time in the regime where flipping one bit is (almost) optimal. These numbers are based on an exact computation for LeadingOnes and on an empirical evaluation of 500 independent runs for OneMax.

Implications for the $(1 + 50)$ $\mathbf{EA}_{r/2,2r}$

Assume that in the regime of optimal one-bit flips the $(1 + 50)$ $\text{EA}_{r/2,2r}$ has correctly identified that flipping one bit is optimal. It will hence use the smallest possible value for r, which is 2. In this case, half the offspring are sampled with the (for this algorithm optimal) mutation rate $1/n$, while the other half of the offspring population is sampled with mutation rate $4/n$, thus flipping on average more than four times the optimal number of bits. It is therefore non-surprising that in this regime (and already before) the gradient of the average fixed-target running time curves in Figures 4.5 are much worse for the $(1 + 50)$ $\text{EA}_{r/2,2r}$ than for the $(1 + 50)$ $\text{EA}_{>0}$.

4.2.2 Creating Half the Offspring with Optimal Mutation Rate

The observations made in the last section inspire the design of the $(1+\lambda)$ $\text{EA}_{r,U(0,\sigma r/n)}$ defined in Algorithm 4. This algorithm samples half the offspring using as deterministic mutation strength the best mutation strength of the last iteration. The other offspring are sampled with a mutation rate that is sampled uniformly at random from the interval $(0, \sigma r/n)$.

As we can see in Figure 4.5 this algorithm significantly improves the performance in those later parts of the optimization process. Normalized total optimization times for various problem dimensions are provided in Figures 4.7 and 4.8, respectively. We display data for $\sigma = 2$ only, and call this $(1 + \lambda)$ $\text{EA}_{r,U(0,\sigma r/n)}$ variant $(1 + \lambda)$ EA_{half}. We note that smaller values of σ, e.g., $\sigma = 1.5$ would give better results. The same effect would be observable when replacing the factor two in the $(1 + \lambda)$ $\text{EA}_{r/(2n),2r}$, i.e., when using a $(1 + \lambda)$ $\text{EA}_{r/(\sigma n),\sigma r}$ rule with $\sigma \neq 2$.

It is remarkable that on LeadingOnes the $(1 + \lambda)$ EA_{half} performs better than RLS, the elitist (1+1) algorithm flipping in each iteration exactly one uniformly chosen bit. The slightly worse gradients for target values $v > n/2$ (which are a consequence of randomly sampling the mutation rate instead of using mutation strength one deterministically) are compensated for by the gains made in the initial phase of the

Algorithm 4: The $(1+\lambda)$ $\text{EA}_{r,U(0,\sigma r/n)}$. In line 6 we denote by $U(a,b)$ the uniform distribution in the interval (a,b). For $\sigma = 2$ we call this algorithm the $(1+\lambda)$ EA_{half}.

1 **Initialization:** Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
2 Initialize $r \leftarrow r^{\text{init}}$; // we use $r^{\text{init}} = 2$;
3 **Optimization:** **for** $t = 1, 2, 3, \ldots$ **do**
4 **for** $i = 1, \ldots, \lambda/2$ **do**
5 Set $\ell^{(i)} \leftarrow r$, create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$;
6 **for** $i = \lambda/2 + 1, \ldots, \lambda$ **do**
7 Sample $p^{(i)} \sim \min\{U(0, \sigma r/n), 1\}$, $\ell^{(i)} \sim \text{Bin}_{>0}(n, p^{(i)})$, create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$;
8 $i \leftarrow \min\left\{j \mid f(y^{(j)}) = \max\{f(y^{(k)}) \mid k \in [n]\}\right\}$;
9 $r \leftarrow \ell^{(i)}$;
10 **if** $f(y^{(i)}) \geq f(x)$ **then** $x \leftarrow y^{(i)}$;

optimization process, where the EA variants benefit from larger mutation rates.

On OneMax the performance of the $(1+\lambda)$ EA_{half} is better than that of the plain $(1+\lambda)$ $\text{EA}_{>0}$ for both tested values $\lambda = 50$ and $\lambda = 2$.

We recall that it is well known that, both for OneMax and LeadingOnes, the optimal offspring population size in the regular $(1+\lambda)$ EA is $\lambda = 1$ [87]. A monotonic dependence of the average optimization time on λ is conjectured (and empirically observed) but not formally proven. While for OneMax the impact of λ is significant, the dependency on λ is much less pronounced for LeadingOnes. Empirical results for both functions and a theoretical running time analysis for LeadingOnes can be found in [56]. For OneMax [68] offers a precise running time analysis of the $(1+\lambda)$ EA for broad ranges of offspring population sizes λ and mutation rates $p = c/n$. In light of the fact that the theoretical considerations in [46] required $\lambda = \omega(1)$, it is worthwhile to note that for all tested problem dimensions the $(1+2)$ $\text{EA}_{r/2,2r}$ performs better on OneMax than the $(1+50)$ $\text{EA}_{r/2,2r}$.

4.2.3 Normalized Standard Bit Mutation

In light of the results presented in the previous section, one may wonder if splitting the population into two halves is needed after all. We investigate this question by introducing the $(1+\lambda)$ $\text{EA}_{\text{norm.}}$ which in each iteration and for each $i \in [\lambda]$ samples the mutation strength $\ell^{(i)}$ from the normal distribution $N(r, r(1-r/n))$ around the best mutation strength r of the previous iteration and rounding the sampled value to the closest in-

teger. The reasons to replace the uniform distribution $U(r/n-\sigma, r/n+\sigma)$ will be addressed below. As before we enforce $\ell^{(i)} \geq 1$ by re-sampling if needed, thus effectively sampling the mutation strength from the conditional distribution $N_{>0}(r, r(1-r/n))$. Algorithm 5 summarizes this algorithm.

Algorithm 5: The $(1+\lambda)$ $\mathrm{EA}_{\mathrm{norm.}}$ with normalized standard bit mutation

1 **Initialization:** Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
2 Initialize $r \leftarrow r^{\mathrm{init}}$; // we use $r^{\mathrm{init}} = 2$;
3 **Optimization:** **for** $t = 1, 2, 3, \ldots$ **do**
4 **for** $i = 1, \ldots, \lambda$ **do**
5 Sample $\ell^{(i)} \sim \min\{N_{>0}(r, r(1-r/n)), n\}$, create $y^{(i)} \leftarrow \mathrm{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$;
6 $i \leftarrow \min\left\{j \mid f(y^{(j)}) = \max\{f(y^{(k)}) \mid k \in [n]\}\right\}$;
7 $r \leftarrow \ell^{(i)}$;
8 **if** $f(y^{(i)}) \geq f(x)$ **then** $x \leftarrow y^{(i)}$;

Note that the variance $r(1-r/n)$ of the unconditional normal distribution $N(r, r(1-r/n))$ is identical to that of the unconditional binomial distribution $\mathrm{Bin}(n, r/n)$. We use the normal distribution here for reasons that will be explained in the next section. Note, however, that very similar results would be obtained when replacing in line 4 of Algorithm 5 the normal distribution $N_{>0}(r, r(1-r/n))$ by the binomial one $\mathrm{Bin}_{>0}(n, r/n)$. We briefly recall that, by the central limit theorem, the (unconditional) binomial distribution converges to the (unconditional) normal distribution.

The empirical performance of the $(1+50)$ $\mathrm{EA}_{\mathrm{norm.}}$ is comparable to that of the $(1+50)$ $\mathrm{EA}_{\mathrm{half}}$ for both problems and all tested problem dimensions, cf. Figures 4.7 and 4.8. Note, however, that for $\lambda = 2$ the $(1+2)$ $\mathrm{EA}_{\mathrm{norm.}}$ performs worse than the $(1+2)$ $\mathrm{EA}_{\mathrm{half}}$.

4.2.4 Interpolating Local and Global Search

As discussed above, all EA variants mentioned so far suffer from the variance of the random selection of the mutation rate, in particular in the long final part of the optimization process in which the optimal mutation strength is one. We therefore analyze a simple way to reduce this variance on the fly. To this end, we build upon the $(1+\lambda)$ $\mathrm{EA}_{\mathrm{norm.}}$ and introduce a counter c, which is initialized at zero. In each iteration, we check if the value of r changes. If so, the counter is re-set to zero. It

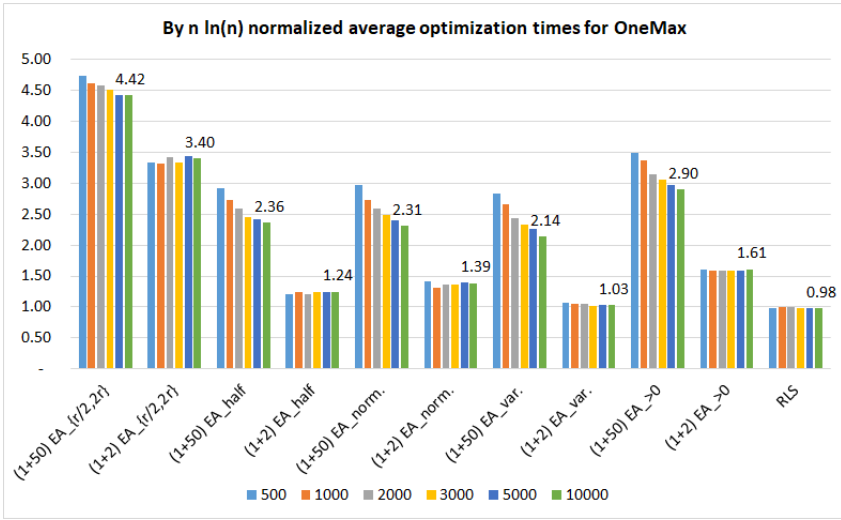


Figure 4.7: By $n \ln(n)$ normalized average optimization times for OneMax, for n between 500 and $10,000$. Displayed numbers are for $n = 10,000$.

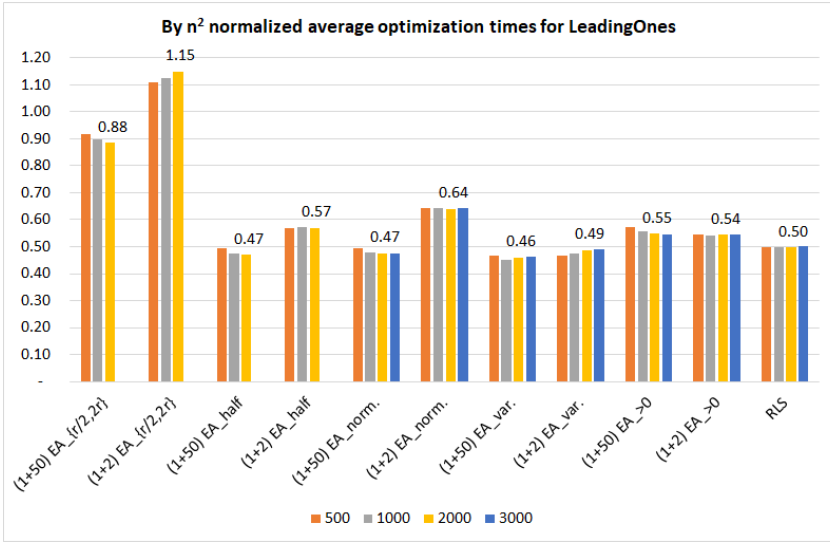


Figure 4.8: By n^2 normalized average optimization times for LeadingOnes, for n between 500 and $3,000$. Displayed numbers are for $n = 2,000$.

is increased by one otherwise, i.e., if the value of r remains the same. We use this counter to self-adjust the variance of the normal distribution. To this end, we replace in line 4 of Algorithm 5 the conditional normal distribution $N_{>0}(r, r(1-r/n))$ by

the conditional normal distribution $N_{>0}(r, F^c r(1-r/n))$, where $F < 1$ is a constant discount factor. Algorithm 6 summarizes this $(1+\lambda)$ EA variant with normalized standard bit mutation and a self-adjusting choice of mean and variance.

Algorithm 6: The $(1+\lambda)$ $\text{EA}_{\text{var.}}$ with normalized standard bit mutation and a self-adjusting choice of mean and variance

1 **Initialization:** Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
2 Initialize $r \leftarrow r^{\text{init}}$; // we use $r^{\text{init}} = 2$;
3 Initialize $c \leftarrow 0$;
4 **Optimization:** **for** $t = 1, 2, 3, \ldots$ **do**
5 **for** $i = 1, \ldots, \lambda$ **do**
6 Sample $\ell^{(i)} \sim \min\{N_{>0}(r, F^c r(1-r/n)), n\}$, create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$;
7 $i \leftarrow \min\left\{j \mid f(y^{(j)}) = \max\{f(y^{(k)}) \mid k \in [n]\}\right\}$;
8 **if** $r = \ell^{(i)}$ **then** $c \leftarrow c+1$; **else** $c \leftarrow 0$;
9 $r \leftarrow \ell^{(i)}$;
10 **if** $f(y^{(i)}) \geq f(x)$ **then** $x \leftarrow y^{(i)}$;

Choice of F: We use $F = 0.98$ in all reported experiments. Preliminary tests suggest that values $F < 0.95$ are not advisable, since the algorithm may get stuck with sub-optimal mutation rates. This could be avoided by introducing a lower bound for the variance and/or by mechanisms taking into account whether or not an iteration has been *successful*, i.e., whether it has produced a strictly better offspring.

The empirical comparison suggests that the self-adjusting choice of the variance in the $(1+\lambda)$ $\text{EA}_{\text{var.}}$ improves the performance on OneMax further, cf. also Figure 4.7 for average fixed-target results for $n = 10,000$. For $\lambda = 2$ the average performance is comparable to, but slightly worse than that of RLS. For LeadingOnes, the $(1+50)$ $\text{EA}_{\text{var.}}$ is comparable in performance to the $(1+50)$ $\text{EA}_{\text{norm.}}$, but we observe that for $\lambda = 2$ the $(1+\lambda)$ $\text{EA}_{\text{var.}}$ performs better. It is the only one among all tested EAs for which decreasing λ from 50 to 2 does not result in a significantly increased running time.

4.2.5 A Meta-Algorithm with Normalized Standard Bit Mutation

In the $(1+\lambda)$ $\text{EA}_{\text{var.}}$ we make use of the fact that a small variance in line 6 of Algorithm 6 results in a more concentrated distribution. The variance adjustment is thus an efficient way to steer the *degree of randomness* in the selection of the mutation

Algorithm 7: The $(1+\lambda)$ Meta-Algorithm with (static) normalized standard bit mutation. The RLS variant with deterministic search radius r and $(1+\lambda)$ EA using standard bit mutation with mutation rate r/n are identical to this algorithm with $\sigma^2 = 0$ and $\sigma^2 = r(1-r/n)$, respectively.

1 **Initialization:** Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
2 **Optimization:** **for** $t = 1, 2, 3, \ldots$ **do**
3 **for** $i = 1, \ldots, \lambda$ **do**
4 Sample $\ell^{(i)} \sim \min\{N_{>0}(r, \sigma^2), n\}$, create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$; // r and σ are two parameters;
5 $y \leftarrow \arg\max\{f(y^{(k)}) \mid k \in [n]\}$;
6 **if** $f(y) \geq f(x)$ **then** $x \leftarrow y$;

rate. It allows to interpolate between deterministic and random mutation rates. In our experimentation we do not go beyond the variance of the binomial distribution, but in principle there is no reason to not regard larger variance as well. The question of how to best determine the degree of randomness in the choice of the mutation rate has, to the best of our knowledge, not previously been addressed in the EC literature. We believe that this idea carries good potential, since it demonstrates that local search with its deterministic search radius and evolutionary algorithms with their global search radii are merely two different configurations of the same meta-algorithm, and not two different algorithms as the general perception might indicate. To make this point very explicit, we introduce with Algorithm 7 a general meta-algorithm, of which local search with deterministic mutation strengths and EAs are special instantiations.

Note that in this meta-model we use static parameter values, variants with adaptive mutation rates can be obtained by applying the usual parameter control techniques, as demonstrated above. Of course, the same normalization can be done for similar EAs, the technique is not restricted to elitist $(1+\lambda)$-type algorithms. Likewise, the condition to flip at least one bit can be omitted, i.e., one can replace the conditional normal distribution $N_{>0}(r, \sigma^2)$ in line 3 by the unconditional $N(r, \sigma^2)$.

4.2.6 Summary

In this section, we introduced the *normalized standard bit mutation*, which replaces the binomial choice of the mutation strength in standard bit mutation by a normal distribution [165]. This normalization allows a straightforward way to control the variance of the distribution, which can now be adjusted independently of the mean. We have demonstrated that such an approach can be beneficial when optimizing classic

benchmark problems such as OneMax and LeadingOnes.

We also note that the parameter control technique which we applied to adjust the mean of the sampling distribution for the mutation strength has an extremely short learning period, since we simply use the best mutation strength of the last iteration as mean for the sampling distribution of the next iteration. For more rugged fitness landscapes a proper learning, which takes into account several iterations, should be preferable. We recall that multi-dimensional parameter control has not received much attention inthe EC literature for discrete optimization problems [42, 92]. Our work falls into this category, and we have demonstrated a simple way to separate the control of the mean from that of the variance of the mutation strength distribution. In Chapter 6, we will introduce more work on *hyperparameter optimization*.

Finally, the meta-algorithm presented in Section 4.2.5 demonstrates that Randomized Local Search and evolutionary algorithms can be seen as two configurations of the meta-algorithm. Parameter control, or, in this context possibly more suitably referred to as online algorithm configuration, offers the possibility to interpolate between these algorithms (and even more drastically, randomized heuristics). Given the significant advances in the context of algorithm configuration witnessed by the EC and machine learning communities, we believe that such meta-models carry significant potential to exploit and profit from advantages of different heuristics. Note here that the configuration of meta-algorithms offers much more flexibility than the algorithm selection approach classically taken in EC, e.g., in most works on hyper-heuristics. Related work *algorithm selection* will also be discussed in Chapter 7.

4.3 The Impact of Crossover Probability for GA

4.3.1 Background

In this section, we look closely into the performance of a $(\mu + \lambda)$ GA on LeadingOnes. The work of this section is motivated by the investigation of the effectiveness of mutation and crossover, more details will be introduced in Section 5.2.

We observe some very interesting effects, that we believe may motivate the theory community to look at the question of usefulness of crossover from a different angle. More precisely, we find that, against our intuition that uniform crossover cannot be beneficial on LeadingOnes, the performance of the $(\mu + \lambda)$ GA on LeadingOnes improves when p_c takes values greater than 0 (and smaller than 1), see Figure 4.9. The performances are quite consistent, and we can observe clear patterns, such as a

tendency for the optimal value of p_c (displayed in Table 4.2) to increase with increasing μ, and to decrease with increasing problem dimension. The latter effect may explain why it is so difficult to observe benefits of crossover in theoretical work: they disappear with the asymptotic view that is generally adopted in runtime analysis.

We have also performed similar experiments on OneMax (see our data [171]), but the good performance of the $(\mu+\lambda)$ GA configurations using crossover is less surprising for this problem, since this benefit has previously been observed for genetic algorithms that are very similar to the $(\mu+\lambda)$ GA; see [25, 29, 30, 143] for examples and further references. In contrast to a large body of literature on the benefit of crossover for solving OneMax, we are not aware of the existence of such results for LeadingOnes, apart from the highly problem-specific algorithms that were developed and analyzed in [1, 52].

4.3.2 A Family of $(\mu+\lambda)$ Genetic Algorithms

We investigate a meta-model, which allows us to easily transition from a mutation-only to a crossover-only algorithm. Algorithm 8 presents this framework, which, for ease of notation, we refer to as the family of the $(\mu+\lambda)$ GA in the following.

The $(\mu+\lambda)$ GA initializes its population uniformly at random (u.a.r., lines 1-2). In each iteration, it creates λ offspring (lines 6–16). For each offspring, we first decide whether to apply crossover (with probability p_c, lines 8–11) or whether to apply mutation (otherwise, lines 12–15). Offspring that differ from their parents are evaluated, whereas offspring identical to one of their parents inherit this fitness value without function evaluation (see [25] for a discussion). The best μ of parent and offspring individuals form the new parent population of the next generation (line 17).

Note the unconventional use of *either* crossover *or* mutation. As mentioned, we consider this variant to allow for a better attribution of the effects to each of the operators. Moreover, note that in Algorithm 8 we decide for each offspring individually which operator to apply. We call this scheme the $(\mu+\lambda)$ **GA with offspring-based variator choice**. We also study the performance of the $(\mu+\lambda)$ **GA with population-based variator choice**, which is the algorithm that we obtain from Algorithm 8 by swapping lines 7 and 6.

4.3.3 Experimental Results

Before we go into the details of the experimental setup and our results, we recall that for the optimization of LeadingOnes, the fitness values only depend on the first bits,

Algorithm 8: A Family of $(\mu + \lambda)$ Genetic Algorithms

Input: Population sizes μ, λ, crossover probability p_c, mutation rate p;
Initialization: **for** $i = 1, \dots, \mu$ **do** sample $x^{(i)} \in \{0,1\}^n$ uniformly at random (u.a.r.), and evaluate $f(x^{(i)})$;
Set $P = \{x^{(1)}, x^{(2)}, \dots, x^{(\mu)}\}$;
Optimization: **for** $t = 1, 2, 3, \dots$ **do**
    $P' \leftarrow \emptyset$;
    for $i = 1, \dots, \lambda$ **do**
        Sample $r \in [0,1]$ u.a.r.;
        if $r \leq p_c$ **then**
            select two individuals x, y from P u.a.r. (w/ replacement);
            $z^{(i)} \leftarrow \text{Crossover}(x, y)$;
            if $z^{(i)} \notin \{x, y\}$ **then** evaluate $f(z^{(i)})$ **else** infer $f(z^{(i)})$ from parent;
        else
            select an individual x from P u.a.r.;
            $z^{(i)} \leftarrow \text{Mutation}(x)$;
            if $z^{(i)} \neq x$ **then** evaluate $f(z^{(i)})$ **else** infer $f(z^{(i)})$ from parent;
        $P' \leftarrow P' \cup \{z^{(i)}\}$;
    P is updated by the best μ points in $P \cup P'$ (ties broken u.a.r.);

whereas the tail is randomly distributed and has no influence on the selection. More precisely, a search point x with LeadingOnes-value $f(x)$ has the following structure: the first $f(x)$ bits are all 1, the $f(x)+1$st bit equals 0, and the entries in the tail (i.e., in positions $[f(x)+2..n]$) did not have any influence on the optimization process so far. For many algorithms, it can be shown that these tail bits are uniformly distributed, see [39] for an extended discussion.

Experimental Setup. We fix in this section the variator choice to the *offspring-based setting*. We test $(\mu + \lambda)$ GA with following parameter combinations on 100-dimensional LeadingOnes: $\mu \in \{2, 3, 5, 8, 10, 20, 30, \dots, 100\}$ (14 values), $\lambda \in \{1, \mu\}$ (3 values), $p_c \in \{0.1k \mid k \in [0..9]\} \cup \{0.95\}$ (11 values), and mutation and crossover operators are fixed standard bit mutation with $p_m = 1/n$ and *uniform crossover*. For each of the settings listed there, we perform 100 independent runs, with a maximal budget of $5n^2$ each.

Overall Running Time. We first investigate the impact of the crossover probability on the average running time, i.e., on the average number of function evaluations that the algorithm performs until it evaluates the optimal solution for the first time. The results for the $(\mu + 1)$ and the $(\mu + \mu)$ GA using uniform crossover and standard

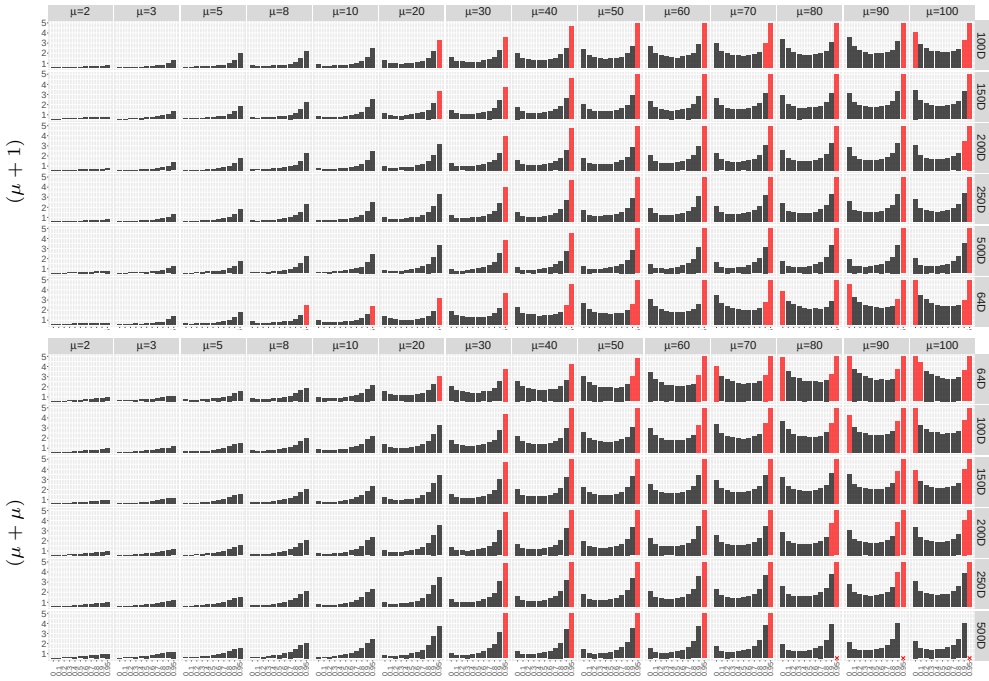


Figure 4.9: By n^2 normalized ERT values for the $(\mu + \lambda)$ GA using standard bit mutation and uniform crossover on LeadingOnes, for different values of μ and for $\lambda = 1$ (top) and for $\lambda = \mu$ (bottom). Results are grouped by the value of μ (main columns), by the crossover probability p_c (minor columns), and by the dimension (rows). The ERTs are computed from 100 independent runs for each setting, with a maximal budget of $5n^2$ fitness evaluations. ERTs for algorithms which successfully find the optimum in all 100 runs are depicted as black bars, whereas ERTs for algorithms with success rates in $(0, 1)$ are depicted as red bars. All bars are capped at 5.

bit mutation are summarized in Figure 4.9. Since not all algorithms managed to find the optimum within the given time budget, we plot as red bars the ERT values for such algorithms with success ratio strictly smaller than 1, whereas the black bars are reserved for algorithms with 100 successful runs. All values are normalized by n^2, to allow for a better comparison.

As a first observation, we note that the pattern of the results are quite regular. As can be expected, the dispersion of the running times is rather small. To give an impression for the concentration of the running times, we report that the standard deviation of the $(50 + 1)$ GA on the 100-dimensional LeadingOnes function is approximately 14% of the average running time across all values of p_c. As can be expected for a genetic algorithm on LeadingOnes, the average running time increases with increasing

population size μ, see [142] for a proof of this statement when $p_c = 0$.

Next, we compare the sub-plots in each row, i.e., fixing the dimension. We see that the $(\mu + \lambda)$ GA suffers drastically from large p_c values when μ is smaller, suggesting that the crossover operator hinders performance. But as μ gets larger, the average running time at moderate crossover probabilities (p_c around 0.5) is significantly smaller than that in two extreme cases, $p_c = 0$ (mutation-only GAs), and $p_c = 0.95$. This observation holds for all dimensions and for both algorithm families, the $(\mu + 1)$ and the $(\mu + \mu)$ GA.

Looking at the sub-plots in each column (i.e., fixing the population size), we identify another trend: for those values of μ for which an advantage of $p_c > 0$ is visible for the smallest tested dimension, $n = 64$, the relative advantage of this rate decreases and eventually disappears as the dimension increases.

Finally, we compare the results of the $(\mu + 1)$ GA with those of the $(\mu + \mu)$ GA. Following [87], it is not surprising that for $p_c = 0$, the results of the $(\mu + 1)$ GA are better than those of the $(\mu+\mu)$ GA (very few exceptions to this rule exist in our data, but in all these cases the differences in average runtime are negligibly small), and following the theoretical analysis [56, Theorem 1], it is not surprising that the differences between these two algorithmic families are rather small: the typical disadvantage of the $(\mu + \lceil \mu/2 \rceil)$ GA over the $(\mu + 1)$ GA is around 5% and it is around 10% for the $(\mu + \mu)$ GA, but these relative values differ between the different configurations and dimensions.

Optimal Crossover Probabilities. To make our observations on the crossover probability clearer, we present in Table 4.2 a heatmap of the values p_c^* for which we observed the best average running time (with respect to all tested p_c values). We see the same trends here as mentioned above: as μ increases, the value of p_c^* increases, while, for fixed μ its value decreases with increasing problem dimension n. Here again we omit details for the $(\mu + \lceil \mu/2 \rceil)$ GA and for the fast mutation scheme, but the patterns are identical, with very similar absolute values.

Fixed-Target Running Times. We now study where the advantage of the crossover-based algorithms stems from. We demonstrate this using the example of the $(50 + 50)$ GA in 200 dimensions. We recall from Table 4.2 that the optimal crossover probability for this setting is $p_c^* = 0.3$. The left plot in Fig. 4.10 is a fixed-target plot, in which we display for each tested crossover probability p_c (different lines) and each fitness value $i \in [0..200]$ (x-axis) the average time needed until the respective algorithm evaluates for the first time a search point of fitness at least i. The mutation-only configuration ($p_c = 0$) performs on par with the best configurations for the first

	n \ μ	2	3	5	8	10	20	30	40	50	60	70	80	90	100
$(\mu+1)$	64	0.0	0.1	0.1	0.1	0.2	0.3	0.5	0.4	0.5	0.5	0.6	0.7	0.6	0.7
	100	0.0	0.1	0.1	0.1	0.1	0.3	0.4	0.4	0.4	0.5	0.5	0.5	0.4	0.6
	150	0.0	0.1	0.1	0.1	0.1	0.2	0.3	0.3	0.4	0.4	0.5	0.4	0.4	0.5
	200	0.0	0.0	0.1	0.1	0.1	0.2	0.2	0.3	0.3	0.3	0.4	0.4	0.4	0.4
	250	0.0	0.0	0.1	0.1	0.1	0.2	0.2	0.3	0.3	0.3	0.4	0.4	0.3	0.4
	500	0.0	0.0	0.0	0.1	0.1	0.1	0.1	0.2	0.2	0.3	0.3	0.3	0.3	0.3
$(\mu+\mu)$	64	0.0	0.2	0.1	0.1	0.2	0.2	0.4	0.4	0.6	0.5	0.5	0.7	0.5	0.7
	100	0.0	0.0	0.1	0.1	0.2	0.3	0.3	0.3	0.5	0.4	0.5	0.5	0.6	0.5
	150	0.0	0.0	0.1	0.1	0.2	0.2	0.3	0.3	0.5	0.4	0.5	0.5	0.5	0.5
	200	0.0	0.0	0.1	0.1	0.1	0.1	0.3	0.3	0.3	0.3	0.4	0.5	0.4	0.5
	250	0.0	0.0	0.1	0.1	0.1	0.2	0.3	0.3	0.3	0.3	0.3	0.3	0.4	0.4
	500	0.0	0.0	0.0	0.1	0.1	0.1	0.1	0.2	0.2	0.3	0.3	0.3	0.3	0.3

Table 4.2: On LeadingOnes, the optimal value of p_c for the $(\mu+1)$ and the $(\mu+\mu)$ GA with uniform crossover and standard bit mutation, for various combinations of dimension n (rows) and μ (columns). Values are approximated from 100 independent runs each, probing $p_c \in \{0.1k \mid k \in [0..9]\} \cup \{0.95\}$.

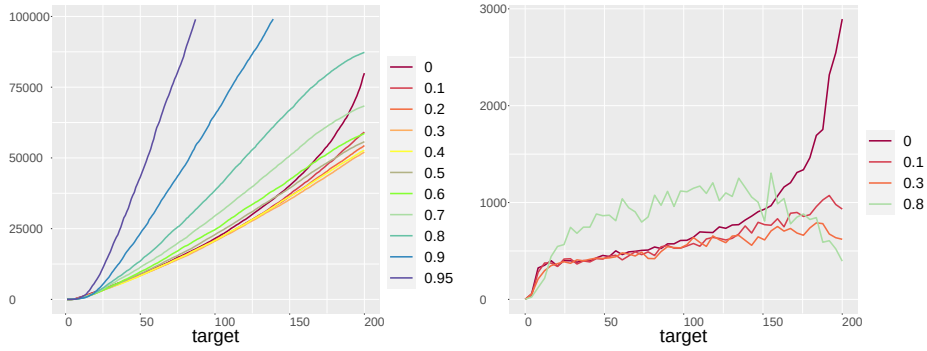


Figure 4.10: **Left:** Average fixed-target running times of the $(50+50)$ GA with uniform crossover and standard bit mutation on LeadingOnes in 200 dimensions, for different crossover probabilities p_c. Results are averages of 100 independent runs. **Right:** Gradient of selected fixed-target curves.

part of the optimization process, but then loses in performance as the optimization progresses. The plot on the right shows the gradients of the fixed-target curves. The gradient can be used to analyze which configuration performs best at a given target value. We observe an interesting behavior here, namely that the gradient of the configuration $p_c = 0.8$, which has a very bad fixed-target performance on all targets (left plot), is among the best in the final parts of the optimization. Inspired by the plot on the right therefore, we investigate the $(\mu+\lambda)$ GA using adaptive an choice of p_c in Section 7.2.

4.3.4 Summary

In this section, by varying the value of the crossover probability p_c of the $(\mu + \lambda)$ GA, we discovered on LeadingOnes that its optimal value p_c (with respect to the average running time) increases with the population size μ, whereas for fixed μ it decreases with increasing dimension n.

Our results raise the interesting question of whether a non-asymptotic runtime analysis (i.e., bounds that hold for a fixed dimension rather than in big-Oh notation) could shed new light on our understanding of evolutionary algorithms. We note that a few examples of such analyses can already be found in the literature, e.g., in [23, 27]. The regular patterns observed in Figure 4.9 suggest the presence of trends that could be turned into formal knowledge. In addition, the result in this section inspires the work on dynamic p_c for the the $(\mu + \lambda)$ GA in Section 7.2.

Chapter 5

Benchmarking Algorithms on IOHprofiler Problems

IOHprofiler provides a benchmark suite of pseudo-Boolean optimization, which allows us to investigate the performance of algorithms on a wide range of problems. In this chapter, we compare the performance of twelve different heuristics on the first twenty-three PBO problems to show how to apply IOHprofiler for such a benchmarking study. Moreover, we investigate how (or whether) crossover can be beneficial for the genetic algorithm by testing the $(\mu + \lambda)$ GA with different parameter settings on the PBO problems.

5.1 Benchmarking Heuristics

5.1.1 Background

As the discussion on the design of IOHprofiler, our goal is to make the platform as flexible as possible so that the user can easily test their algorithms on the problems and with respect to performance criteria of their choice (see Chapter 3 for the discussion). However, the original framework only provided the experimental setup, but did not fix any benchmark problems or reference algorithms. In this chapter, we present the results of 12 different heuristics for the original 23 PBO problems, which serves as the first baseline for the performance evaluation of user-defined heuristics. All performance data is available in our data repository and can be straightforwardly assessed through the web-based version of IOHanalyzer (http://iohprofiler.liacs.nl/), which

is easily accessible for future comparative studies. An important by-product of our contribution is the identification of additional statistics, which are included within the IOHprofiler. This section provides extensive examples of assessing algorithms' performance over a set of problems concerning different perspectives (such as fixed-target result, fixed-budget result, and ECDF.)

5.1.2 Summary of Baseline Algorithms

High-level Description

We evaluate a total number of twelve different algorithms on the first 23 problems described in Section 2.5. We have chosen algorithms that may serve for future references, since they all have some known strengths and weaknesses that will become apparent in the following discussions. The selection therefore shows a clear bias towards algorithms for which theoretical analyses are available.

Note that most algorithms are parametrized, and we use here in this work only standard parametrizations (e.g., we use standard bit mutation with $1/n$ as mutation rates, etc.). Analyzing the effects of different parameter values as was done, for example in [32, 133], would be very interesting, related work on parameter tuning will be introduced in Section 6.

We also note that, except for the so-called vGA, our implementations (deliberately) deviate slightly from the text-book descriptions referenced below. Following the discussion in previous chapters, we enforce that offspring created by mutation are different from their parent and resample without further evaluation if needed. Likewise, we do not evaluate recombination offspring that are identical to one of their immediate parents.

All algorithms start with uniformly chosen initial solution candidates.

We list here the twelve implemented algorithms, and provide further details and pseudo-codes:

1. **gHC:** A (1+1) greedy hill climber, which goes through the string from left to right, flipping exactly one bit per each iteration, and accepting the offspring if it is at least as good as its parent.

2. **RLS:** Randomized Local Search, the elitist (1+1) strategy flipping one uniformly chosen bit in each iteration. That is, RLS and gHC differ only in the choice of the bit which is flipped. While RLS is unbiased in the sense of Section 2.5.1, gHC is not permutation-invariant and thus biased.

3. $(1+1)$ **EA:** The $(1+1)$ EA with static mutation rate $p = 1/n$. This algorithm differs from RLS in that the number of uniformly chosen, pairwise different bits to be flipped is sampled from the conditional binomial distribution $\mathrm{Bin}_{>0}(n,p)$. Details refer to Algorithm 2 in Section 4.1.

4. **fGA:** The "fast GA" proposed in [50] with $\beta = 1.5$. Its *mutation strength* (i.e., the number of bits flipped in each iteration) follows a power-law distribution with exponent β. This results in a more frequent use of large mutation-strength, while maintaining the property that small mutation strengths are still sampled with reasonably large probability.

5. $(1+10)$ **EA:** The $(1+10)$ EA with static $p = 1/n$, which differs from the $(1+1)$ EA only in that 10 offspring are sampled (independently) per each iteration.

6. $(1+10)$ $\mathbf{EA}_{r/2,2r}$**:** The two-rate EA with self-adjusting mutation rates suggested and analyzed in [46] (see Algorithm 3 in Section 4.2).

7. $(1+10)$ $\mathbf{EA}_{\text{norm.}}$**:** a variant of the $(1+10)$ EA sampling the mutation strength from a normal distribution $N(pn, pn(1-p))$ with a self-adjusting choice of p (see Algorithm 5 in Section 4.2).

8. $(1+10)$ $\mathbf{EA}_{\text{var.}}$**:** The $(1+10)$ $\mathrm{EA}_{\text{norm.}}$ with an adaptive choice of the variance in the normal distribution from which the mutation strengths are sampled (see Algorithm 6 in Section 4.2).

9. $(1+10)$ $\mathbf{EA}_{\text{log-n.}}$ The (1+10) EA with log-normal self-adaptation of the mutation rate proposed in [7].

10. $(1+(\lambda,\lambda))$ **GA:** A binary (i.e., crossover-based) EA originally suggested in [43]. We use the variant with self-adjusting λ analyzed in [41].

11. **vGA:** A $(30, 30)$ "vanilla" GA (following the so-called traditional GA, as described, for example, in [69, 5]).

12. **UMDA:** A univariate marginal distribution algorithm from the family of estimation of distribution algorithms (EDAs). UMDA was originally proposed in [120].

Detailed Description of the Algorithms

Detailed description of $(1+1)$ EA, $(1+10)$ EA, $(1+10)$ $\mathrm{EA}_{r/2,2r}$, $(1+10)$ $\mathrm{EA}_{\text{norm.}}$, and $(1+10)$ $\mathrm{EA}_{\text{var.}}$ can be found in Section 4, and descriptions of the remaining

algorithms follow. An operator frequently used in these descriptions is the $\text{flip}_\ell(\cdot)$ mutation operator, which flips the entries of ℓ pairwise different, uniformly at random chosen bit positions. Details can be found in Algorithm 1.

Again, to avoid useless evaluations of offspring that are identical to their parents, we make use of the conditional binomial distribution $\text{Bin}_{>0}(n,p)$, which assigns probability $\text{Bin}(n,p)(k)/(1-(1-p)^n)$ to each positive integer $k \in [n]$, and probability zero to all other values. Sampling from $\text{Bin}_{>0}(n,p)$ is identical to sampling from $\text{Bin}(n,p)$ until a positive value is returned ("resampling strategy").

Greedy Hill Climber The greedy hill climber (gHC, Algorithm 9) uses a deterministic mutation strength, and flips one bit in each iteration, going through the bit string from left to right, until being stuck in a local optimum, see Algorithm 9.

Algorithm 9: Greedy hill climber (gHC)

Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
Optimization: **for** $t = 1, 2, 3, \ldots$ **do**
  $x^* \leftarrow x$;
  Flip in x^* the entry in position $1 + (t \mod n)$ and evaluate $f(x^*)$;
  if $f(x^*) \geq f(x)$ **then** $x \leftarrow x^*$;

Randomized Local Search RLS uses a deterministic mutation strength, and flips one randomly chosen bit in each iteration, see Algorithm 10.

Algorithm 10: Randomized local search (RLS)

Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
Optimization: **for** $t = 1, 2, 3, \ldots$ **do**
  create $x^* \leftarrow \text{flip}_1(x)$, and evaluate $f(x^*)$;
  if $f(x^*) \geq f(x)$ **then** $x \leftarrow x^*$;

Fast Genetic Algorithm The *fast Genetic Algorithm* (fGA) chooses the mutation length ℓ according to a power-law distribution $D^\beta_{n/2}$, which assigns to each integer $k \in [n/2]$ a probability of $\Pr[D^\beta_{n/2} = k] = (C^\beta_{n/2})^{-1} k^{-\beta}$, where $C^\beta_{n/2} = \sum_{i=1}^{n/2} i^{-\beta}$. We use the (1+1) variant of this algorithm with $\beta = 1.5$.

Algorithm 11: Fast genetic algorithm (fGA) from [50]

Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
Optimization: **for** $t = 1, 2, 3, \ldots$ **do**
  for $i = 1, \ldots, \lambda$ **do**
    Sample $\ell^{(i)} \sim D^{\beta}_{n/2}$;
    create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$;
  $x^* \leftarrow \arg\max\{f(y^{(1)}), \ldots, f(y^{(\lambda)})\}$ (ties broken by favoring the largest index);
  if $f(x^*) \geq f(x)$ **then** $x \leftarrow x^*$;

The EA with log-Normal self-adaptation of mutation rate The $(1 + \lambda)$ $\text{EA}_{\text{log-n.}}$, Algorithm 12, uses a self-adaptive choice of the mutation rate.

Algorithm 12: The $(1 + \lambda)$ $\text{EA}_{\text{log-n.}}$ with log-Normal self-adaptation of the mutation rate

Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
$p = 0.2$;
Optimization: **for** $t = 1, 2, 3, \ldots$ **do**
  for $i = 1, \ldots, \lambda$ **do**
    $p^{(i)} = \left(1 + \frac{1-p}{p} \cdot \exp(0.22 \cdot \mathcal{N}(0,1))\right)^{-1}$;
    Sample $\ell^{(i)} \sim \text{Bin}_{>0}(n, p^{(i)})$;
    create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$;
  $i \leftarrow \min\left\{j \mid f(y^{(j)}) = \max\{f(y^{(k)}) \mid k \in [\lambda]\}\right\}$;
  $p \leftarrow p^{(i)}$;
  $x^* \leftarrow \arg\max\{f(y^{(1)}), \ldots, f(y^{(\lambda)})\}$ (ties broken by favoring the smallest index);
  if $f(x^*) \geq f(x)$ **then** $x \leftarrow x^*$;

The Self-Adjusting $(1 + (\lambda, \lambda))$ GA The self-adjusting $(1 + (\lambda, \lambda))$ GA, Algorithm 13, was introduced in [43] and analyzed in [41]. The offspring population size λ is updated after each iteration, depending on whether or not an improving offspring could be generated. Since both the mutation rate and the crossover bias (see Algorithm 14 for the definition of the biased crossover operator cross) depend on λ, these two parameters also change during the run of the $(1 + (\lambda, \lambda))$ GA. In our implementation we use update strength $F = 3/2$.

Algorithm 13: The self-adjusting $(1 + (\lambda, \lambda))$ GA

Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
Optimization: **for** $t = 1, 2, 3, \ldots$ **do**
  Mutation phase:
    Sample $\ell \sim \text{Bin}_{>0}(n, \lambda/n)$;
    for $i = 1, \ldots, \lambda$ **do** create $y^{(i)} \leftarrow \text{flip}_\ell(x)$, and evaluate $f(y^{(i)})$;
    $x^* \leftarrow \arg\max\{f(y^{(1)}), \ldots, f(y^{(\lambda)})\}$ (ties broken by favoring the largest index);
  Crossover phase:
    for $i = 1, \ldots, \lambda$ **do** create $y^{(i)} \leftarrow \text{cross}_c(x, x^*)$, and evaluate $f(y^{(i)})$;
    $y^* \leftarrow \arg\max\{f(y^{(1)}), \ldots, f(y^{(\lambda)})\}$ (ties broken by favoring the largest index);
  Selection phase:
    if $f(y^*) > f(x)$ **then** $x \leftarrow y^*$; $\lambda \leftarrow \max\{\lambda/F, 1\}$;
    if $f(y^*) = f(x)$ **then** $x \leftarrow y^*$; $\lambda \leftarrow \min\{\lambda F^{1/4}, n\}$;
    if $f(y^*) < f(x)$ **then** $\lambda \leftarrow \min\{\lambda F^{1/4}, n\}$;

Algorithm 14: Crossover operation $\text{cross}_c(x, x^*)$ with crossover bias c

$y \leftarrow x$;
Sample $\ell \sim \text{Bin}_{>0}(n, c)$;
Select ℓ different positions $\{i_1, \ldots, i_l\} \in [n]$;
for $j = 1, 2, \ldots, \ell$ **do** $y_{i_j} \leftarrow x^*_{i_j}$;

The "Vanilla" GA The vanilla GA (vGA, Algorithm 16) constitutes a textbook realization of the so-called Traditional GA [5, 69]. The algorithm holds a parental population of size μ. It employs the Roulette-Wheel-Selection (RWS, that is, probabilistic fitness-proportionate selection which permits an individual to appear multiple times) as the sexual selection operator to form $\mu/2$ pairs of individuals that generate the offspring population. 1-point crossover (Algorithm 15) is applied to every pair with a fixed probability of $p_c = 0.37$. A mutation operator is then applied to every individual, flipping every bit with a fixed probability of $p_m = 2/n$. This completes a single cycle.

Algorithm 15: 1-Point crossover of two parents $x^{(1)}$ and $x^{(2)}$

Sample $\ell \in [n]$ uniformly at random;
for $i = 1, 2, \ldots, \ell$ **do** Set $y_i^{(1)} \leftarrow x_i^{(1)}$ and $y_i^{(2)} \leftarrow x_i^{(2)}$;
for $i = \ell + 1, \ldots, n$ **do** Set $y_i^{(1)} \leftarrow x_i^{(2)}$ and $y_i^{(2)} \leftarrow x_i^{(1)}$;

Algorithm 16: The (μ, μ)-"Vanilla-GA" with mutation rate p_m and crossover probability p_c

Initialization:
  for $i = 1, \ldots, \mu$ **do** sample $x^{(i)} \in \{0,1\}^n$ uniformly at random and evaluate $f(x^{(i)})$;
Optimization: **for** $t = 1, 2, 3, \ldots$ **do**
  Parent selection phase: Apply roulette-wheel selection to $\{x^{(1)}, \ldots, x^{\mu}\}$ to select μ parent individuals $y^{(1)}, \ldots, y^{(\mu)}$;
  Crossover phase:
    for $i = 1, \ldots, \mu/2$ **do** with probability p_c replace $y^{(i)}$ and $y^{(2i)}$ by the two offspring that result from a 1-point crossover of these two parents, for a randomly chosen crossover point $j \in [n]$;
  Mutation phase:
    for $i = 1, \ldots, \mu$ **do** Sample $\ell^{(i)} \sim \mathrm{Bin}(n, p_m)$, set $y^{(i)} \leftarrow \mathrm{flip}_{\ell^{(i)}}(y^{(i)})$, and evaluate $f(y^{(i)})$;
  Replacement:
    for $i = 1, \ldots, \mu$ **do** Replace $x^{(i)}$ by $y^{(i)}$;

The Univariate Marginal Distribution Algorithm The univariate marginal distribution algorithm (UMDA, Algorithm 17) is one of the simplest representatives of the family of so-called estimation of distribution algorithms (EDAs). The algorithm maintains a population of size s (we use $s = 50$ in our experiments) and uses the best $s/2$ of these to estimate the marginal distribution of each decision variable, by simply counting the relative frequency of ones in the corresponding position. These frequencies are capped at $1/n$ and $1 - 1/n$, respectively. In the t-th iteration, a new population is created by sampling from these marginal distributions. Building upon previous work made in [119], the UMDA was introduced in [120]. Theoretical results for this algorithm are summarized in [100].

Algorithm 17: The Univariate Marginal Distribution Algorithm (UMDA), representing the family of EDAs

Initialization:
  for $i = 1, \ldots, s$ **do** sample $x^{(0,i)} \in \{0,1\}^n$ uniformly at random and evaluate $f(x^{(0,i)})$;
  Let $\boldsymbol{P_0}$ be the collection of the best $s/2$ of these search points, ties broken uniformly at random (u.a.r.);
Optimization:
  for $t = 1, 2, 3, \ldots$ **do**
    for $j = 1, \ldots, n$ **do**
      $p_j \leftarrow 2|\{x \in \boldsymbol{P_{t-1}} \mid x_j = 1\}|/s$;
      if $p_j < 1/n$ **then** $p_j = 1/n$;
      if $p_j > 1 - 1/n$ **then** $p_j = 1 - 1/n$;
    for $i = 1, \ldots, s$ **do** sample $x^{(t,i)} \in \{0,1\}^n$ by setting, independently for all $j \in [n]$, $x_j^{(t,i)} = 1$ with probability p_j and setting $x_j^{(t,i)} = 0$ otherwise. Evaluate $f(x^{(t,i)})$;
    Let $\boldsymbol{P_t}$ be the collection of the best $s/2$ of the points $x^{(t,1)}, x^{(t,2)}, ..., x^{(t,s)}$, ties broken u.a.r.;

5.1.3 Experimental Results

Experimental Setup

Our experimental setup can be summarized as follows:

- 23 test-functions F1-F23, described in Section 2.5
- Each function is assessed over the four problem dimensions $n \in \{16, 64, 100, 625\}$
- Each algorithm is run on 11 different instances of each of these 92 (F, n) pairs, yielding a total number of $1,012$ different runs per each algorithm. Each run is granted a budget of $100n^2$ function evaluations for dimensions $n \in \{16, 64, 100\}$ and a budget of $5n^2$ function evaluations for $n = 625$. More precisely, each algorithm performs one run on each of the instances $1 - 6$ and $51 - 55$ described in Section 2.5.1.

  Most of the tested algorithms are unbiased and comparison-based. For these algorithms all 11 instances look the same, i.e., performing one run each is equivalent to 11 independent runs on instance 1, which is the "pure" problem instance without fitness scaling nor any other transformation applied to it. However, in order to understand how the transformations impact the behavior of vGA and

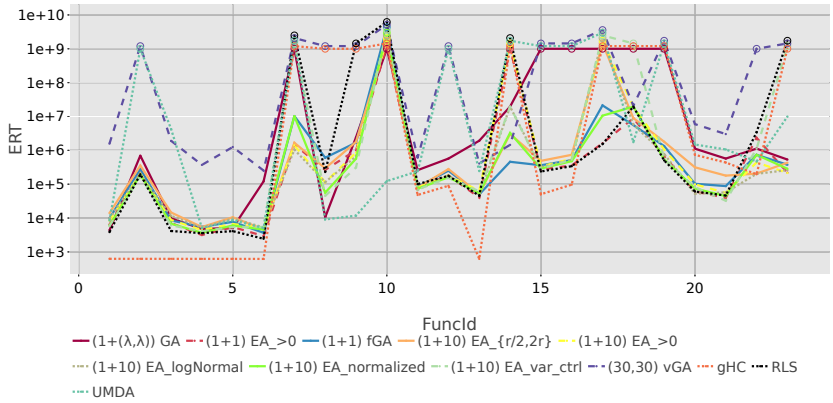


Figure 5.1: ERT values of the twelve baseline algorithms for the 625-dimensional test suite, with respect to the best solution quality found by any of the algorithms in any of the eleven runs. These target values can be found in Table 5.2.

gHC, we also performed 11 independent runs of each algorithm on instance 1 of each (F, n) pair, yielding another $1,012$ runs per each algorithm.

- For each run we store the current and the best-so-far function value at each evaluation. This setup allows very detailed analyses, since we can zoom into each range of fixed budgets and/or fixed-targets of choice, and obtain our anytime performance statistics in terms of quantiles, averages, probabilities of success, ECDF curves, etc.

  For some of the algorithms we also store information about the self-adjusting parameters, for example the value of λ in the $(1 + (\lambda, \lambda))$ GA and the mutation rates for the (1+10) $\text{EA}_{r/2,2r}$, the (1+10) $\text{EA}_{\text{var.}}$, and the (1+10) $\text{EA}_{\text{norm.}}$. From this data we can derive how the parameters evolve with respect to the time elapsed and with respect to the quality of the best-so-far solutions.

Concerning the number of repetitions, we note that with 11 runs we already get a good understanding of the key differences between the algorithms. 11 runs can be enough to get statistical significance, if the differences in performance are substantial. We refer the interested reader to the tutorial [76], which argues that for a first experiment a small number of experiments can suffice.

Function-wise Raw Observations Across Dimensions Figures 5.1 and 5.2 depict the ERT of the baseline algorithms on the 625-dimensional and the 64-dimensional

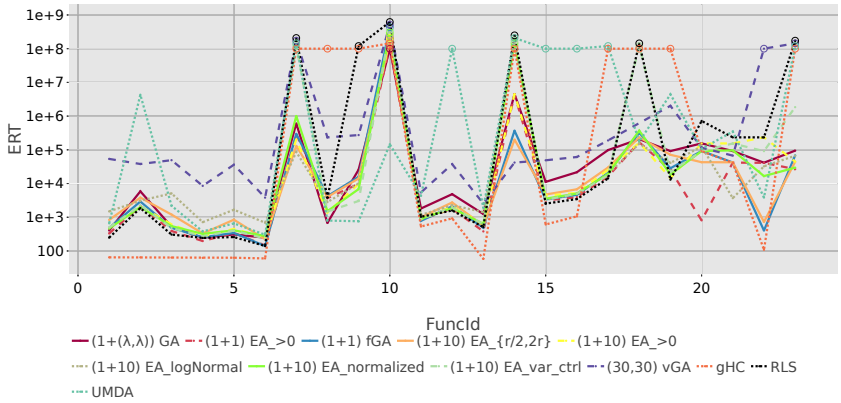


Figure 5.2: ERT values of the twelve baseline algorithms for the 64-dimensional test suite, with respect to the best solution quality found by any of the algorithms in any of the eleven runs. These target values can be found in Table 5.1.

funcId	$F1$	$F2$	$F3$	$F4$	$F5$	$F6$	$F7$	$F8$	$F9$	$F10$	$F11$	$F12$
$n = 64$	64	64	2,080	32	57	21	64	33	64	63.2	32	57
$n = 625$	625	625	195,625	312	562	208	576.4	314	625	625	312	562

funcId	$F13$	$F14$	$F15$	$F16$	$F17$	$F18$	$F19$	$F20$	$F21$	$F22$	$F23$
$n = 64$	21	43.8	33	64	64	3.981492	64	128	192	28	8
$n = 625$	208	36.6	314	625	625	4.2655266	621	1,200	1,775	268.4	24

Table 5.1: Target values for which the ERT curves in Figures 5.1 and 5.2 are computed.

functions, respectively, when considering the best function value found by any of the algorithms in any of the runs. These target values are summarized in Table 5.1.

We summarize a few basic observations for each function.

F1: This baseline OneMax problem is easily solved, having the gHC winning (it solves each n-dimensional OneMax instance in at most $n + 1$ queries), the majority of the algorithms clustered with a practically-equivalent performance, the (1+10)-EA$_{r/2,2r}$ lagging behind, and the vGA outperformed by far. All algorithms locate the global optimum eventually. Figure 5.3 presents the average fixed-target performance of the algorithms on F1 at $n = 625$, in terms of ERT. Evidently, the vGA and the UMDA obtain a clear advantage in the beginning of the optimization process, although the vGA eventually uses the largest number of evaluations, by far, to locate the optimum. We also see here that, as expected, the performances of the unbiased algorithms (i.e., all algorithms except the vGA) are identical for the 11 runs on instance 1 and the 1 run on 11 different instances. For the vGA this is clearly not the case, the fixed-target performances of these two settings

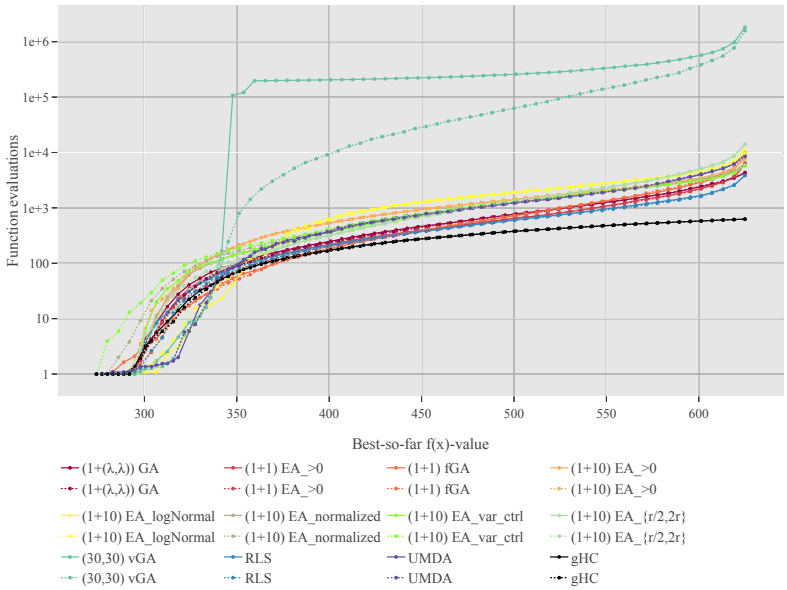


Figure 5.3: ERT values for F1 (OneMax) at dimension $n = 625$ in a fixed-target perspective. The dashed lines are the average running times of 11 independent runs on instance 1, while the solid lines are average running times for one run on each of the eleven different instances 1-6 and 51-55. Note that this figure is not generated by IOHanalyzer.

differ substantially.

F2: The LeadingOnes problem introduces more difficulty when compared to F1, with the ERT consistently shifting upward, but it is still easily solved. The gHC wins, the vGA loses, and the majority of the algorithms are again clustered, but now the $(1+(\lambda,\lambda))$-GA lags behind. The UMDA fails to find the optimum within the given time budget, for all tested dimensions except for $n = 16$. An example of the evolution of the parameter λ in the $(1+(\lambda,\lambda))$ GA is visualized in Figure 5.4. We observe – as expected – that larger function values are evidently correlated with larger population sizes (and, thus, larger mutation rates).

F3: The behavior on this problem, the linear function with harmonic weights, is similar to F1 for most algorithms. Exceptions are the vGA, for which it is slightly easier, and the UMDA, which shows worse performance on F3 than on F1.

F4: This problem, OneMax with 50% dummy variables, is the most easily-solved

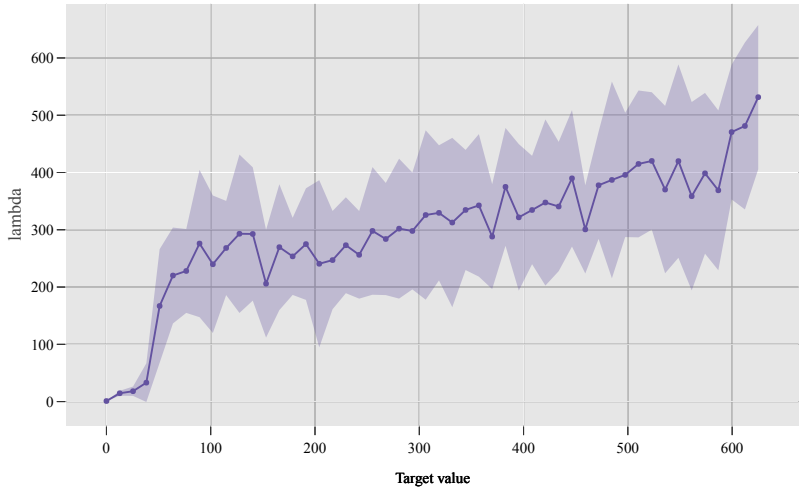


Figure 5.4: Evolution of the population size λ of the $(1 + (\lambda, \lambda))$ GA on the LeadingOnes problem F2 at dimension n = 625, correlated to the best-so-far objective function values (horizontal axis). The line shows the average value of λ for iterations starting with a best-so-far solution of the value indicated by the x-axis. The shade represents the standard deviation.

problem in the suite, with an even simpler performance classification – the gHC performs at the top, the vGA at the bottom, and the rest are tightly clustered. Given the consistent correlation with the F1 performance profiles, across all twelve algorithms, it seems debatable whether or not to keep this function in a benchmark suite, since it seems to offer only limited additional insights, which could be of a rather specialized interest, e.g., for theoretical investigations addressing precise running times of the algorithms.

F5: Solving this problem, OneMax with 10% dummy variables, exhibits equivalent behavior to F1. Similarly to F4, we suggest to ignore this setup for future benchmarking activities. Note, however, that the exclusion of F4 and F5 does *not* imply that the dummy variables do not play an interesting role – in an ongoing evaluation of the W-model, we are currently investigating their impact when combined with other W-model transformations.

F6: The neutrality ("majority vote") transformation apparently introduces difficulty to the $(1 + (\lambda, \lambda))$ GA, which exhibits deteriorated performance compared to F1. The vGA, despite a slightly better performance compared to F1, is the worst among the twelve algorithms. At the same time, the (1+10)-$EA_{\text{log-n.}}$ lags behind

its competitors in the beginning, but it eventually shows a competitive result in the later optimization process, ending up with an overall fine ERT value. The gHC outperforms all other algorithms also on this function.

F7: The introduction of local permutations to OneMax, within the current problem, introduced difficulties to all the algorithms. The ability to locate the global optimum within the designated budget deteriorated for all of them, except for the (1+10)-EA$_{r/2,2r}$ on "low-dimensional" scales ($n \in \{16, 64, 100\}$). Figure 5.5 depicts the ERT values of the algorithms on F7 at $n = 625$, where it is evident that they all failed to locate the global optimum. Note that this figure encompasses results for both instantiations (a single instance or 11 instances). The twelve algorithms' performances are clustered in two groups that are associated with two fitness regions (and likely two basins of attraction) - the first around an objective function value of 500 (including the UMDA, with the gHC being the fastest to approach it and get stuck), and the other below 600. It seems that the latter cluster could use additional budget to further improve the results.

F8: Being OneMax with the small fitness plateaus induced by the ruggedness function r_1, the UMDA performs best on this problem, with the $(1 + (\lambda, \lambda))$ GA following very closely. It seems to introduce medium difficulty to all the algorithms, except for the gHC, whose performance is dramatically hampered and becomes worse than the vGA. Interestingly, the ERT values are distributed sparsely compared to other OneMax variants.

F9: The UMDA also performs best for this problem, with the (1+10) EA$_{\text{var.}}$ being the runner-up. Generally, the behavior on this problem, OneMax with small fitness perturbations, is close to F8, but with certain differences. F9 is evidently harder, as the algorithms meet larger ERT values. Importantly, unlike F8, RLS always fails on F9 (since it gets stuck in local optima), and "joins" the gHC and vGA at the bottom of the performance table. The $(1 + (\lambda, \lambda))$ GA also shows worse performance on F9 than on F8.

F10: This problem, OneMax with fitness perturbations of block size five, presents a dramatic difficulty to all the algorithms, including the UMDA, which, however, clearly outperforms all other algorithms. It is evidently the hardest OneMax variant for all the tested algorithms, among the eight variants studied in this work. For $n = 625$ the UMDA finds the optimum after an average of $141,243$ evaluations, while none of the other algorithms finds a solution better than 575.

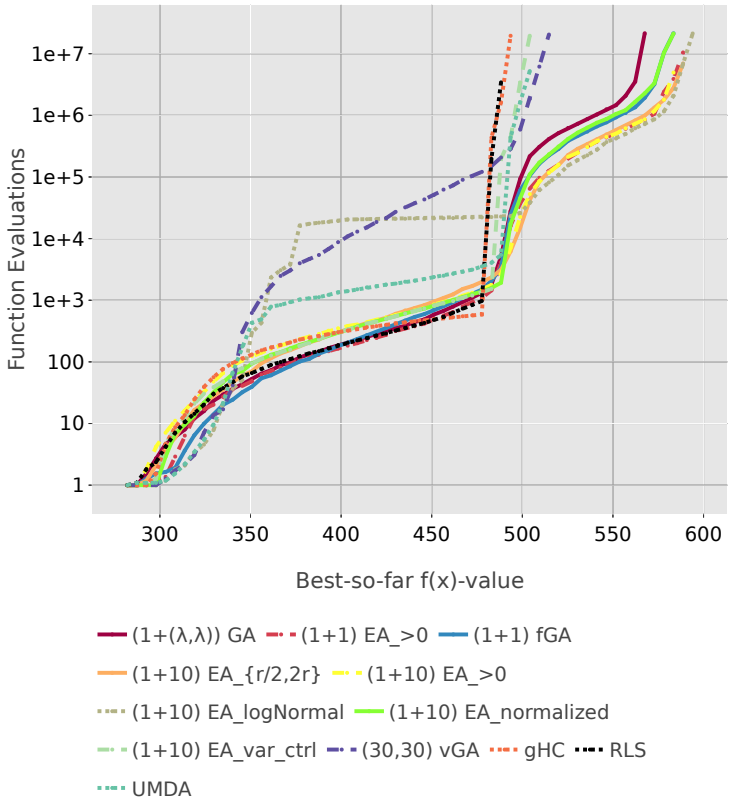


Figure 5.5: ERT values for F7 at dimension $n = 625$ in a fixed-target perspective.

F11: The gHC performs strongly on this problem, namely the LeadingOnes with 50% dummy variables, consistently with its winning behavior on F2. The vGA performs poorly, and the UMDA is also at the bottom of the table. Notably, the problem should become easier compared to F2, since the effective number of variables is reduced. RLS, however, which generally performs well on LeadingOnes, only ranks third from the bottom on ERT values when solving this problem.

F12: The behavior of the algorithms on this problem, LeadingOnes with 10% dummy variables, is very similar to F11, with excellent performance of the gHC. However, one major difference is the dramatic deterioration of UMDA and vGA, which fail to find the optimum with given time budget for $n = 625$ (see Figure 5.1). UMDA performs better than vGA for $n = 64$, but still obtains clear disadvantage comparing to other algorithms (see Figure 5.2).

F13: The introduction of neutrality on LeadingOnes makes this problem easier in practice (that is, by observing ERT decrease compared to F2). The gHC wins, while the vGA, $(1 + (\lambda, \lambda))$ GA as well as the UMDA lag behind the other methods. The poor performance of these three algorithms is consistent with their performance on LeadingOnes.

F14: Being LeadingOnes with epistasis, this problem introduces high difficulty. For high dimensions, $n \in \{64, 100, 625\}$, none of the algorithms was capable of locating an optimal solution within the allocated budget. The vGA tops the ERT values on this problem, followed by the (1+1)-fGA, the (1+10)-$\text{EA}_{r/2,2r}$, and the (1+10)-$\text{EA}_{\text{norm.}}$. On the other hand, three algorithms, namely the gHC, the UMDA, and the RLS, seem to get trapped with low objective function values.

F15: The introduction of fitness perturbations to LeadingOnes makes this problem difficult. The UMDA exhibits the worst performance among the competing methods. The remainder of the algorithms, except for the vGA and the $(1+(\lambda, \lambda))$ GA, are still able to hit the optimum of this problem, but with significantly larger ERT values. The gHC performs best, and the first runner-up is the RLS.

F16: The obtained ranks of algorithms, with respect to the ERT values, are similar to those of F15, but generally exhibit higher ERT values. Notably, the UMDA is still the worst performer.

F17: As expected, the rugged LeadingOnes function is the second-hardest among the LeadingOnes variants, following F14. Only the RLS and the (1+1)-EA are able to hit the optimum in dimension 625, while the gHC has a diminished performance on this problem. This can be explained by the fact that the gHC has a very high probability of getting stuck in a local traps, while the RLS is capable of performing random walks about local optima, until eventually escaping them (e.g., by flipping the right bit when all the consecutive four bits are also identical to the target string). This is of course a rare event, and the ERT values are therefore significantly worse than all other LeadingOnes variants, except F14. As on the previous two functions, the UMDA performs poorly, with similar ERT values as the gHC.

Comparing to F10, the effect of the fitness permutation r_3 on LeadingOnes is not as significant as on OneMax, which can be explained by the ability of most of the algorithms to perform random walks to deal with local traps, through which the four first bits of the tail are eventually set correctly, at which point flipping

the significant bit (i.e., the bit in position $\mathrm{LO}(x)+1$) results in a LeadingOnes fitness increase of at least five, and consequently a fitness increase of at least one for the problem $r_3 \circ$ LeadingOnes. This candidate solution is thus accepted by all of our algorithms, and the next phase of optimizing the following consecutive five bits begins.

F18: The LABS problem is the most complex problem in our assessment. For the higher dimensions, $n \in \{64, 100, 625\}$, none of the algorithms obtained the maximally attainable values, or got fitness values close to those of the best-known sequences (see, e.g., [125]). Additionally, a couple of algorithms (e.g., the gHC and the RLS) did not succeed to escape low-quality "local traps" on most dimensions. Surprisingly, the vGA was superior to the other algorithms at $n = 16$ but, as expected, over the higher dimensions presented weaker performance. Notably, the UMDA outperforms the other methods at $n = 625$.

F19: The simplest problem among the Ising instances. Most of the algorithms exhibited similar performance, except for the vGA, the UMDA and the gHC, which obtained weak results. The latter preformed worst among all algorithms, and obtained the lowest objective function values across all the dimensions for the given time budget. As a demonstration of the performance statistics that IOHprofiler provides, average fixed-target and fixed-budget running times are provided in Figure 5.6. This figure illustrates that ERT values tell only one side of the story: the performance of UMDA is comparable to that of the other algorithms for all targets up to around 85; only then it starts to perform considerably worse.

F20: In contrast to its poor performance on the 1D-Ising (F19), the gHC outperformed the other algorithms on the 2D-Ising for target values up to around $1,136$ ($d = 625$), after which its performance becomes worse than most of the other algorithms, except for the vGA, which is consistently the worst except for a few initial target values.For $d = 625$, however, the (1+10) $\mathrm{EA}_{\text{log-n.}}$ achieves the best ERT value for the target recorded in Table 5.2, followed by the (1+1) EA and RLS.

F21: As expected, the most complex among the Ising model instances. The observed performances resemble the observations on F20. For $d = 625$, the best ERT is obtained by the (1+10)-$\mathrm{EA}_{\text{var.}}$.

F22: None of the algorithms succeeded in locating the global optimum across all dimensions of this problem. This is explained by the existence of a local optimum with a

strong basin of attraction. The gHC and the vGA exhibited inferior performance compared to the other algorithms.

F23: Some algorithms failed to locate the global optimum of the N-Queens problem in high dimensions, yet the vGA, the gHC and the UMDA constantly possessed the worst ERT values. Fine performance was observed for the (1+10)-EA$_{>0}$ and the (1+10) EA$_{\text{log-n.}}$.

Grouping of Functions and Algorithms In the following we are aiming to recognize patterns and identify classes within (i) the set of all functions, and (ii) the set of all algorithms.

Functions' Empirical Grouping: It is evident that problems F1-F6, F11-F13 and F15-F16 are treated relatively easily by the majority of the algorithms, with those functions based on LeadingOnes (i.e., F2, F11-13, F15, F16) being more challenging within this group. On the other extreme, F7, F9-F10, F14, F18-F19 and F22 evidently constitute a class of hard problems, on which all algorithms consistently exhibit difficulties (except for $n = 16$); the LABS function (F18) seems the most difficult among them. F8, the instances of the Ising model (F19-F21), as well as the NQP (F23), constitute a class of moderate level of difficulty.

Algorithms' Observed Trends: The gHC and the vGA usually exhibited extreme performance with respect to the other algorithms. The vGA consistently suffers from poor performance over all functions, while the gHC either leads the performance on certain functions or performs very poorly on others. The gHC's behavior is to be expected, since it is correlated with the existence of local traps (by construction) – for instance, it consistently performs very well on F1-F6, while having difficulties on F7-F10. Clearly, RLS also gets trapped by the deceptive functions, while at the same time it shows fine performance on most of the non-deceptive problems. The UMDA's performance stands out. Evidently, it performs well on the OneMax-based problems, but fails to optimize the LeadingOnes function F2 and its derivatives F11-F17, with the exception of F11 and F13 – a behavior that might be interesting to analyze further in future work. Otherwise, we observe one primary class of algorithms exhibiting equivalent performance over all problems in all dimensions: The seven algorithms $(1 + (\lambda, \lambda))$-GA, (1+1)-EA, (1+10)-EA$_{\text{var.}}$, (1+10)-EA, (1+10)-EA$_{\text{norm.}}$, (1+10)-EA$_{r/2,2r}$, and (1+1)-fGA behave consistently, typically exhibiting fine performance. In terms of ERT values, the (1+10)-EA$_{\text{log-n.}}$ could also be grouped into this class of seven algorithms, but it behaves quite differently during the optimization process, often showing an opposite trend of convergence speed at the early stages of the

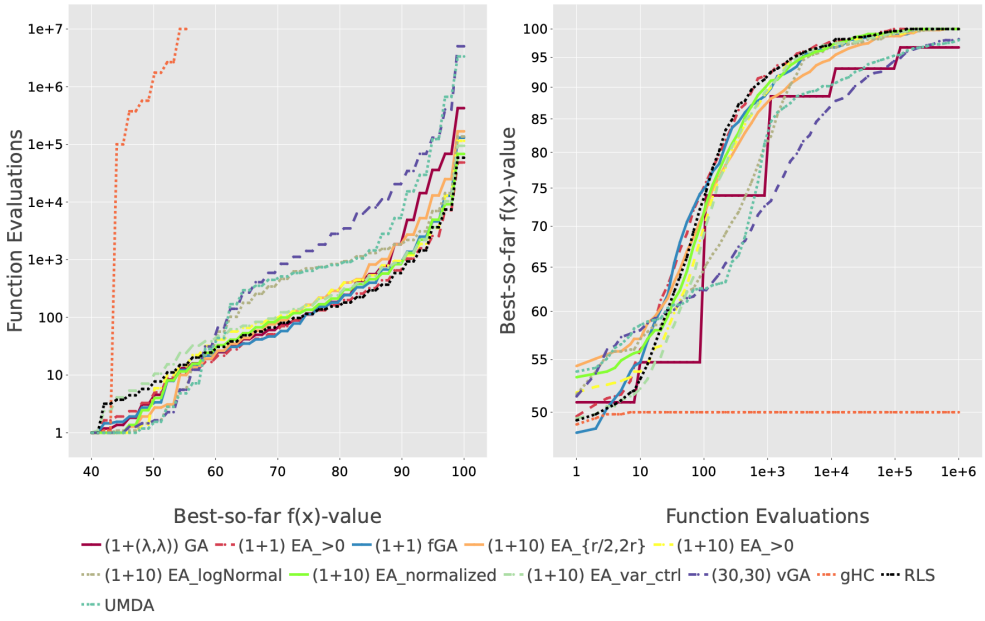


Figure 5.6: Demonstration of the basic performance plots for F19 at dimension $n = 100$: **Left:** best obtained values as a function of evaluations calls ("fixed-target perspective"), versus **Right:** evaluations calls as a function of best obtained values ("fixed-budget perspective"). For F19, these patterns of relative behavior are observed across all dimensions.

optimization procedure.

Ranking: We also examined the overall number of runs per test-function in which an algorithm successfully located the best recorded value – the so-called *hitting number*. We then grouped those hitting numbers by dimension, and ranked the algorithms per each dimension. The (1+10)-EA$_{r/2,2r}$ consistently leads the grouped hitting numbers on the "low-dimensional" functions ($n \in \{16,\ 64\}$), with (1+1)-fGA and (1+10)-EA$_{\text{norm.}}$ being together the first runner-up. The (1+10)-EA also exhibits high ranking across all dimensions. (1+10)-EA$_{\text{norm.}}$ leads the grouped hitting numbers on $n = 100$, whereas the (1+1)-EA leads the hitting numbers on the "high-dimensional" functions at $n = 625$, with (1+10)-EA being the runner-up. Across all dimensions, UMDA, gHC and vGA are with the lowest rankings.

Visual Analytics: As a demonstration of the performance statistics offered by IOHprofiler, we provide snapshots of visual analytics that supported our examination. Figure 5.6 depicts basic performance plots for F19 at dimension $n = 100$, in

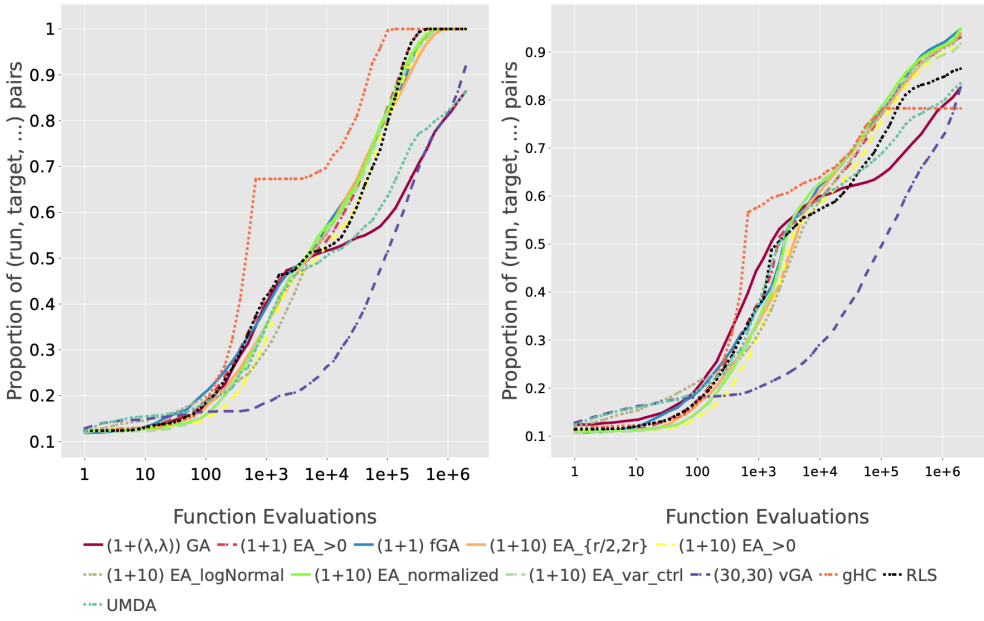


Figure 5.7: ECDF curve for the class of "easily-solved" functions in dimension $n = 625$: F1-F6, F11-F13, and F15-F16 [LEFT] and of all 23 functions [RIGHT], with respect to equally spaced target values.

so-called *fixed-target* and *fixed-budget* perspectives. For clarity of the plots we only show the ERT values and the average function values achieved per each budget, respectively. Standard deviations as well as the $2, 5, 10, 15, 50, 75, 90, 95, 98\%$ quantiles are available on http://iohprofiler.liacs.nl/.

In Figure 5.7 we provide two plots obtained from our new module which computes ECDF curves for user-specified target values. The plot on the left depicts an ECDF curve for the "easily-solved" functions identified above (i.e., F1-F6, F11-F13, and F15-F16) in dimension $n = 625$. The one on the right shows the ECDF curves across all 23 benchmark functions. For both figures we have chosen ten equally spaced target values per each function, with the largest value being again the best function value identified by any of the algorithms in any of the runs. Since the number of "easy" problems dominates our overall assessment the curves on the right are to a large extent dominated by the performances depicted on the left. This indicates once again the need for a thorough revision of our benchmark selection.

Unbiasedness: Following our experimental planning to test the hypothesized

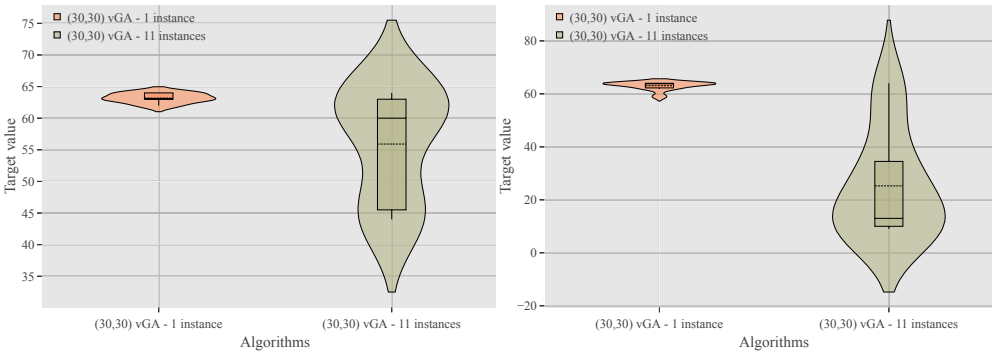


Figure 5.8: Statistical box-plots for vGA's attained function values on instance one alone versus on the eleven different instances 1-6 and 51-55, after exploiting the entire budget (namely, 409,600 function evaluations): F1 [LEFT] and F2 [RIGHT]. Both plots are for $n = 64$.

"biasedness" effect for the vGA, we compared its averaged performance on instance 1 versus on all the other instances (1-6 and 51-55) altogether. Figure 5.8 depicts a comparison of attained objective function values, by means of box-plots, on F1 and on F2 for $n = 64$. Performance deterioration is indeed evident on the permuted instances; that is, instances 51-55, for which the base functions are composed with a σ-transformation of the bit strings, as described in Section 2.5.1. The box-plots in Figure 5.8 show very clearly that the vGA treats the plain F1 and F2 much better, in terms of attained target values, than their transformed variants. The plots are for $n = 64$ and after exhausting the full budget of $100n^2$ function evaluations.

5.1.4 Summary

This section presented results of the 12 heuristics on the first 23 PBO problems, which contributes a baseline for future comparative studies. This work has inspired many directions for IOHprofiler, and some of them are already under development.

Additional Performance Measures: While this section presents a very detailed assessment of algorithms' performance, we are continuously strengthening the statistical repertoire of IOHanalyzer by introducing new performance measures and by devising better procedures. Currently, IOHanalyzer also supports pairwise Kolmogorov-Smirnov test, Glicko2-based ranking, the Deep Statistical Comparison (DSC) analysis [59], and Contribution to portfolio (Shapley-values).

Combinations of W-model Transformations: As discussed in Section 2.5.6,

the transformations of the W-model can be combined with each other. To analyze the individual effects of each transformation, and in order to keep the size of the experimental setup reasonable, we have not considered such combinations in this work. A critical consideration of adding such combinations, and of extending the base transformations (e.g., with respect to the fitness transformation, but also the size of the neutrality transformation, etc.) forms another research line that we are currently addressing in a parallel work stream.

Integration of Algorithm Design Software: IOHs are to a large extent modular algorithms, whose components can be exchanged and executed in various different ways. This has letted the community to develop software which enables an easier algorithmic design. Examples for such software are ParadisEO [24] for single-objective and multi-objective optimization and jMetal [58] for multi-objective algorithms. Building or integrating such software could allow much more comprehensive algorithm benchmarking, and could eventually automate the detection of promising algorithmic variants. We are glad to see that IOHprofiler has contributed to this domain, for example, by being integrated with other frameworks for the work of large scale automated algorithm design [3, 153, 168, 170].

5.2 Benchmarking a $(\mu + \lambda)$ Genetic Algorithm with Configurable Crossover Probability

5.2.1 Background

Classic evolutionary computation methods build on two main variation operators: *mutation* and *crossover*. While the former can be mathematically defined as unary operators (i.e., families of probability distributions that depend on a single argument), crossover operators sample from distributions of higher arity, with the goal to "recombine" information from two or more arguments.

There is a long debate in evolutionary computation about the (dis-)advantages of these operators, and about how they interplay with each other [118, 140]. In lack of generally accepted recommendations, the use of these operators still remains a rather subjective decision, which in practice is mostly driven by users' experience. Little guidance is available on which operator(s) to use for which situation, and how to most efficiently interleave them. The question how crossover can be useful can therefore be seen as far from being solved.

Of course, significant research efforts are spent to shed light on this question,

which is one of the most fundamental ones that evolutionary computation has to offer. While in the early years of evolutionary computation (see, for example, the classic works [5, 34, 69]) crossover seems to have been widely accepted as an integral part of an evolutionary algorithm, we observe today two diverging trends. Local search algorithms such as GSAT [136] for solving Boolean satisfiability problems, or such as the general-purpose Simulated Annealing [96] heuristic, are clearly very popular optimization methods in practice – both in academic and in industrial applications. These purely mutation-based heuristics are nowadays more commonly studied under the term *stochastic local search*, which forms a very active area of research. Opposed to this is a trend to reduce the use of mutation operators, and to fully base the iterative optimization procedure on recombination operators; see [152] and references therein. However, despite the different recommendations, these opposing positions find their roots in the same problem: we hardly know how to successfully dovetail mutation and crossover.

In addition to large bodies of empirical works aiming to identify useful combinations of crossover and mutation [34, 63, 85, 121], the question how (or whether) crossover can be beneficial has also always been one of the most prominent problems in *runtime analysis*, the research stream aiming at studying evolutionary algorithms by mathematical means [25, 29, 30, 31, 43, 47, 48, 88, 89, 98, 107, 123, 141, 143, 157, 162], most of these results focus on very particular algorithms or problems, and are not (or at least not easily) generalizable to more complex optimization tasks.

In this section, we study a simple variant of the $(\mu+\lambda)$ GA mentioned in Section 4.3, which allows us to conveniently scale the relevance of crossover and mutation, respectively, via a single parameter. More precisely, our algorithm is parameterized by a crossover probability p_c, which is the probability that we generate in the reproduction step an offspring by means of crossover. The offspring is generated by mutation otherwise, so that $p_c = 0$ corresponds to the mutation-only $(\mu + \lambda)$ EA, whereas for $p_c = 1$ the algorithm is entirely based on crossover. Note here that we *either* use crossover *or* mutation, so as to better separate the influence of the two operators.

We study the performance of different configurations of the $(\mu + \lambda)$ GA on 25 IOHprofiler problems. We observe that the algorithms using crossover perform significantly better on some simple functions as OneMax (F1) and LeadingOnes (F2), but also on some problems that are considered hard, e.g., the 1-D Ising model (F19).

$F1$	$F2$	$F3$	$F4$	$F5$	$F6$	$F7$	$F8$	$F9$	$F10$	$F11$	$F12$	$F13$	$F14$
100	100	5,050	50	90	33	100	51	100	100	50	90	33	7
$F15$	$F16$	$F17$	$F18$	$F19$	$F20$	$F21$	$F22$	$F23$	$F24$	$F25$			
51	100	100	4.215852	98	180	260	42	9	17.196	-0.2965711			

Table 5.2: Target values used for computing the ERT value in Figure 5.9.

5.2.2 Experimental Results

Experiment setup In order to probe into the empirical performance of the $(\mu + \lambda)$ GA, we test it on the 25 problems mentioned in Section 2.5, with a total budget of $100n^2$ function evaluations. We perform 100 independent runs of each algorithm on each problem. For the $(\mu+\lambda)$ GA (see Algorithm 8 in Section 4.3), we study three different crossover operators, *one-point crossover*, *two-point crossover*, and *uniform crossover*, and two different mutation operators, *standard bit mutation* and the *fast mutation* scheme suggested in [50]. These variation operators are briefly described as follows.

- *One-point crossover*: a crossover point is chosen from $[1..n]$ u.a.r. and an offspring is created by copying the bits from one parent until the crossover point and then copying from the other parent for the remaining positions.
- *Two-point crossover*: similarly, two different crossover points are chosen u.a.r. and the copy process alternates between two parents at each crossover point.
- *Uniform crossover* creates an offspring by copying for each position from the first or from the second parent, chosen independently and u.a.r.
- *Standard bit mutation:* a mutation strength ℓ is sampled from the conditional binomial distribution $\mathrm{Bin}_{>0}(n,p)$, which assigns to each k a probability of $\binom{n}{k}p^k(1-p)^{n-k}/(1-(1-p)^n)$ [25]. Thereafter, ℓ distinct positions are chosen u.a.r. and the offspring is created by first copying the parent and then flipping the bits in these ℓ positions. Still, we restrict our experiments to the standard mutation rate $p = 1/n$.
- *Fast mutation* [50]: operates similarly to standard bit mutation except that the mutation strength ℓ is drawn from a power-law distribution: $\Pr[L = \ell] = (C^\beta_{n/2})^{-1}\ell^{-\beta}$ with $\beta = 1.5$ and $C^\beta_{n/2} = \sum_{i=1}^{n/2} i^{-\beta}$.

Moreover, we test the algorithm with $\mu \in \{10, 50, 100\}$ and $\lambda \in \{1, \lceil \mu/2 \rceil, \mu\}$. Detailed results for the different configurations of the $(\mu + \lambda)$ GA are available in our data repository at [171].

Results on IOHprofiler problems In Figure 5.9, we highlight a few basic results of this experimentation for $n = 100$, where the mutation operator is fixed to the

5.2. Benchmarking a $(\mu + \lambda)$ Genetic Algorithm with Configurable Crossover Probability

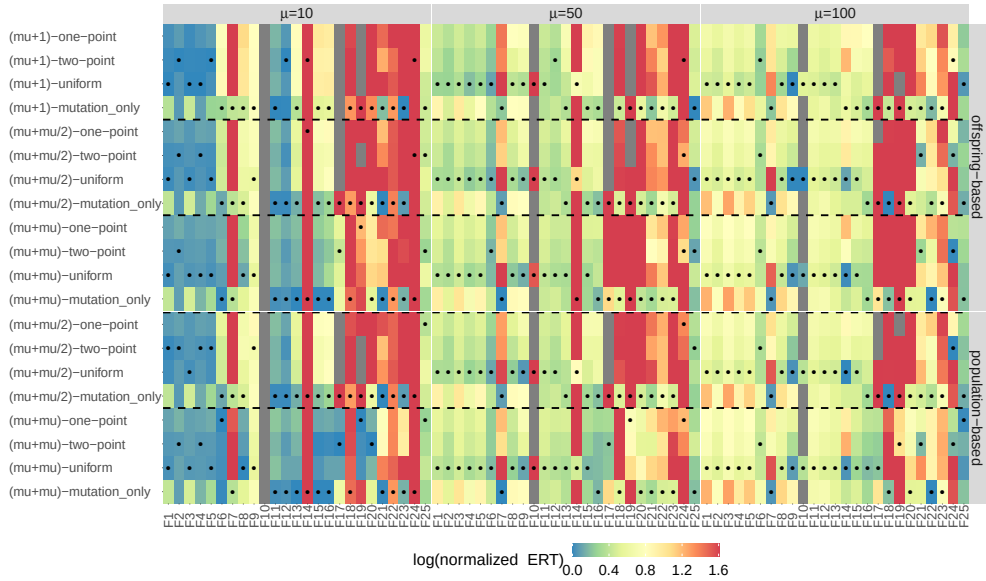


Figure 5.9: Heat map of normalized ERT values of the $(\mu+\lambda)$ GA with offspring-based (top part) and population-based (bottom part) variator choice for the 100-dimensional benchmark problems, computed based on the target values specified in Table 5.2. The crossover probability p_c is set to 0.5 for all algorithms except the mutation-only ones (which use $p_c = 0$). The displayed values are the quotient of the ERT and $\mathrm{ERT}_{\mathrm{best}}$, the ERT achieved by the best of all displayed algorithms. These quotients are capped at 40 to increase interpretability of the color gradient in the most interesting region. The three algorithm groups – the $(\mu + 1)$, the $(\mu + \lceil \mu/2 \rceil)$, and the $(\mu + \mu)$ GAs – are separated by dashed lines. A dot indicates the best algorithm of each group of four. A grey tile indicates that the $(\mu + \lambda)$ GA configuration failed, in all runs, to find the target value within the given budget.

standard bit mutation. More precisely, we plot in this figure the normalized expected running time (ERT), where the normalization is with respect to the best ERT achieved by any of the algorithms for the same problem. Table 5.2 provides the target values for which we computed the ERT values. For each problem and each algorithm, we first calculated the 2% percentile of the best function values. We then selected the largest of these percentiles (over all algorithms) as target value.

On the OneMax-based problems F1, F4, and F5, the $(\mu + \lambda)$ GA outperforms the mutation-only GA, regardless of the variator choice scheme, the crossover operator, and the setting of λ. When looking at problem F6, we find that when $\mu = 10$ the mutation-only GA surpasses most of $(\mu + \lambda)$ GA variants except the population-based $(\mu+\mu)$ GA with one-point crossover. On F8-10, the $(\mu+\lambda)$ GA takes the lead in general,

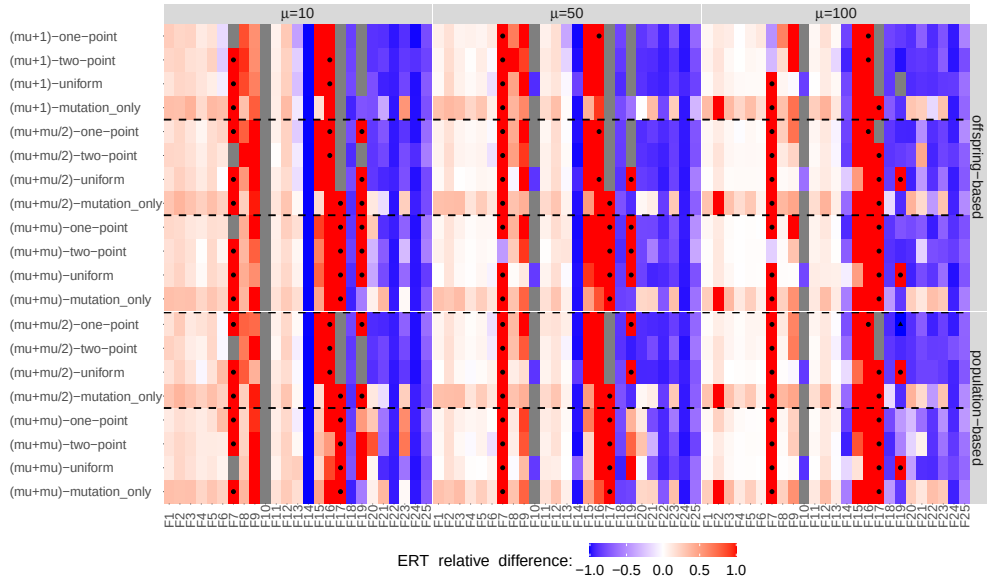


Figure 5.10: Heat map comparing the $(\mu + \lambda)$ GAs using the standard bit mutation (sbm) with the $(\mu + \lambda)$ GAs using the fast mutation on the 25 problems from Sec. 2.5 in dimensions $n = 100$. Plotted values are $(\mathrm{ERT}_{\mathrm{fast}} - \mathrm{ERT}_{\mathrm{sbm}})/\mathrm{ERT}_{\mathrm{sbm}}$, for ERTs computed wrt the target values specified in Table 5.2. p_c is set to 0.5 for all crossover-based algorithms. Values are bounded in $[-1, 1]$ to increase visibility of the color gradient in the most interesting region. A black dot indicates that the $(\mu + \lambda)$ GA with fast mutation failed in all runs to find the target with the given budget; the black triangle signals failure of standard bit mutation, and a gray tile is chosen for settings in which the $(\mu + \lambda)$ GA failed for both mutation operators.

whereas it cannot rival the mutation-only GA on F7. Also, only the configuration with uniform crossover can hit the optimum of F10 within the given budget.

On the linear function F3 we observe a similar behavior as on OneMax. On LeadingOnes (F2), the $(\mu + \lambda)$ GA outperforms the mutation-only GA again for $\mu \in \{50, 100\}$ while for $\mu = 10$ the mutation-only GA becomes superior with one-point and uniform crossovers. On F11-13 and F15-16 (the W-model extensions of LeadingOnes), the mutation-only GA shows a better performance than the $(\mu+\lambda)$ GA with one-point and uniform crossovers and this advantage becomes more significant when $\mu = 10$. On problem F14, that is created from LeadingOnes using the same transformation as in F7, the mutation-only GA is inferior to the $(\mu + \lambda)$ GA with uniform crossover.

On problems F18 and F23, the mutation-only GA outperforms the $(\mu + \lambda)$ GA for

most parameter settings. On F21, the $(\mu + \lambda)$ GA with two-point crossover yields a better result when the population size is larger (i.e., $\mu = 100$) while the mutation-only GA takes the lead for $\mu = 10$. On problems F19 and F20, the $(\mu + \mu)$ GA with the population-based variator choice significantly outperforms all other algorithms, whereas it is substantially worse for the other parameter settings. On problem F24, the $(\mu+\mu/2)$ GA with two-point crossover achieves the best ERT value when $\mu = 100$. None of the tested algorithms manages to solve F24 with the given budget. The target value used in Figure 5.9 is 17.196, which is below the optimum 20. On problem F25, the mutation-only GA and the $(\mu + \lambda)$ GA are fairly comparable when $\mu \in \{10, 50\}$. Also, we observe that the population-based $(\mu+\mu)$ GA outperforms the mutation-only GA when $\mu = 100$.

In general, we have made the following observations: (1) on problems F1-6, F8-9, and F11-13, all algorithms obtain better ERT values with $\mu = 10$. On problems F7, F14, and F21-25, the $(\mu + \lambda)$ GA benefits from larger population sizes, i.e., $\mu = 100$; (2) The $(\mu + \mu)$ GA with uniform crossover and the mutation-only GA outperform the $(\mu + \lceil \mu/2 \rceil)$ GA across all three settings of μ on most of the problems, except F10, F14, F18, and F22. For the population-based variator choice scheme, increasing λ from one to μ improves the performance remarkably on problems F17-24. Such an improvement becomes negligible for the offspring-based scheme; (3) Among all three crossover operators, the uniform crossover often surpasses the other two on OneMax, LeadingOnes, and the W-model extensions thereof.

To investigate the impact of mutation operators on GA, we plot in Figure 5.10 the relative ERT difference between the $(\mu+\lambda)$ GA configurations using fast and standard bit mutation, respectively. As expected, fast mutation performs slightly worse on F1-6, F8, and F11-13. On problems F7, F9, and F15-17, however, fast mutation becomes detrimental to the ERT value for most parameter settings. On problems F10, F14, F18, and F21-25, fast mutation outperforms standard bit mutation, suggesting a potential benefit of pairing the fast mutation with crossover operators to solve more difficult problems. Interestingly, with an increasing μ, the relative ERT of the $(\mu + \lambda)$ GA quickly shrinks to zero, most notably on F1-6, F8, F9, F11-13.

Interestingly, in [117], an empirical study has shown that on a randomly generated maximum flow test generation problem, fast mutation is significantly outperformed by standard bit mutation when combined with uniform crossover. Such an observation seems contrary to our findings on F10, F14, F18, and F21-25. However, it is made on a standard $(100 + 70)$ GA in which both crossover and mutation are applied to the parent in order to generate offspring.

5.2.3 Summary

In this section, we have analyzed the performance of a family of $(\mu + \lambda)$ GAs, in which offspring are either generated by crossover (with probability p_c) or by mutation (probability $1 - p_c$). On the PBO problem set, it has been shown that this random choice mechanism reduces the expecting running time on OneMax, LeadingOnes, and many W-model extensions of those two problems.

It would certainly also be interesting to extend the study to a $(\mu + \lambda)$ GA variant using (dynamic) tuned values for the relevant parameters μ, λ, crossover probability p_c, and mutation rate p. Therefore, based on the results in this section, we will introduce our work on *algorithm configuration* in Chapter 6 and *dynamic algorithm selection* in Chapter 7.

Chapter 6

Automatic Configuration of Genetic Algorithms

We compare in this chapter the results from Section 5.2 with those obtained from three different types of automated algorithm configuration methods, one based on iterated racing (we use Irace [111]), one surrogate-assisted technique (we use the mixed-integer parallel efficient global optimization MIP-EGO [151]), and a classic heuristic optimization method (we use the mixed-integer evolutionary strategies MIES suggested in [109]). Also, we configure the $(\mu + \lambda)$ GA considering two different objectives, i.e., expected running time and the area under empirical cumulative distribution function curve.

6.1 Background

It was discussed in Section 5.2 that there is a long debate about the effectiveness of the two main variators, crossover and mutation, and their combinations [118, 140]. Several works study the synergy of mutation and crossover, both by empirical and by theoretical means, see [34, 63, 85, 121] and [144], respectively, as well as references mentioned therein. However, most of these results focus on specific algorithms and problems. Widely accepted guidelines for their deployment are scarce, leading to a situation in which users often rely on their own experience. To reduce the bias inherent to such manual decisions, a number of automated algorithm configuration techniques have been developed, to assist the user with data-driven suggestions. To

deploy these techniques, one formulates the operator choice and/or their intensity as a meta-optimization problem, widely referred to as the *algorithm configuration (AC)* or the *hyperparameter optimization* problem.

The AC problem was classically addressed by standard search heuristics such as mixed-integer evolution strategies [4, 71, 109]. More specific AC tools have been developed in recent years, among them surrogate-based models (e.g., SPOT [13], SMAC [82], MIP-EGO [151]), racing-based methods (Irace [111], F-race [18]) and optimization-based methods (ParamILS [83]).

The result in this chapter is built on Section 5.2 in which we analyzed a configurable framework of $(\mu+\lambda)$ GAs that scales the relevance of mutation and crossover by means of the crossover probability $p_c \in [0,1]$. Recall that, in Algorithm 8, the framework creates new solution candidates by either applying mutation (with probability $1-p_c$) or by applying crossover (with probability p_c). This way, it can separate the influence of these operators from each other. While we have studied several operator choices in Section 5.2 by means of grid search, we consider here only one type of crossover (uniform crossover) and one type of mutation (standard bit mutation) to keep the search space manageable and to better highlight our key findings.

Note that, to distinguish it from the budget of the configurators, we denote the budget of the GAs as *cutoff time* in the following.

Automated algorithm configuration for improving the anytime performance of algorithms has been applied in several works, both with respect to classical CPU time (e.g., for the travel salesperson problem [20], for MAX-MIN ant systems, and for mixed-integer programming [112]) and for the here-considered function evaluation budgets (see [3] for a recent example). However, we are not aware of any works using anytime measures with the objective to identify configurations that minimize ERT values. On the other hand, previous work has studied the impact of the cutoff time for algorithm configurators [73, 74]. In [74], the authors conclude that considering the best-found fitness is more efficient than considering the expected running time. Our work considers AUC, a measure that takes both the found fitness and running time into account, and we conclude that the cutoff time can influence AUC less, when compared to ERT.

All our data is publicly available at [169]. Section 4.3 describes the configurable $(\mu+\lambda)$ GA (Algorithm 13). The benchmark problems were introduced in Section 2.5, and the cost metrics used to evaluate the algorithms were explained in Section 2.4.

6.2 Configurators

We briefly introduce the three AC methods that are applied for configuring the $(\mu + \lambda)$ GA: Irace [111], a mixed-integer parallel efficient global optimization (MIP-EGO [155]), and the mixed-integer evolution strategy (MIES [109]), which we briefly describe in the following paragraphs. All configurators work with a user-defined *configuration budget*, which is the maximal number of algorithm runs that the AC method is allowed to perform before recommending its final outcome.

Irace [111] is a so-called iterated racing method designed for hyperparameter optimization. The main steps of Irace are (1) sampling values of parameters from particular distributions, after which (2) the algorithms with the corresponding sampled parameter settings are evaluated across a set of instances, then (3) elitist ones will be selected by the racing method and (4) the sampling distributions are updated based on the elitist configurations, to bias the sampling towards the elitist space. The sampling distributions of parameters are independent of each other unless user-defined constraints and conditions exist. For the racing method, the sampled parameter settings are evaluated on instances. After several steps, parameter settings that are statistically worse than others are discarded. At the end of the configuration process, one or several *elite* configurations are returned to the user, along with their performance observed during the configuration.

Efficient global optimization (EGO), also known as Bayesian optimization, is designed to solve costly-to-evaluate global optimization problems. For our configuration problem, we use an EGO-variant called mixed-integer parallel EGO (**MIP-EGO** [155]) capable of handling mixed-integer search space. EGO starts by randomly sampling solution candidates $\{\theta_1, \theta_2, \ldots\}$ and evaluating their fitness $\{c_1, c_2, \ldots\}$. From these observations EGO learns a predictive distribution of the fitness value for each unseen configuration using stochastic models, e.g., random forests or Gaussian processes. Aiming at balancing the trade-off between the accuracy and uncertainty of this predictive distribution, EGO uses a so-called *acquisition function* to decide which solution candidates to sample next. Common acquisition functions are expected improvement and probability of improvement; see [67, 137] for an overview. For this work, we use the moment-generating function of improvement (MGFI) [155], which is defined as the weighted combination of all moments of the predictive distribution. For the weights, we took a robust setting in [154], obtained by an extensive empirical study on the BBOB problem set. To learn the predictive distribution, we choose a random forest model as it deals with the mixed-integer/categorical search space more

naturally than Gaussian processes.

The mixed-integer evolution strategy (MIES) uses principles from evolution strategies for handling continuous, discrete, and nominal parameters by using self-adaptive mutation operators for all three parameter types [109]. MIES starts with a randomly initialized parent population of size μ, and then it generates λ offspring candidates for each generation iteratively. Offspring are generated by recombining two randomly selected parents and then mutating the solution resulting from the recombination, after which (μ, λ) selection is applied to the offspring population, i.e., the μ best of the λ offspring form the parent population of the next iteration.

6.3 Experimental Results

6.3.1 Experimental Setup

Each AC method is granted a budget of $5,000$ *target runs*, where each target run corresponds to ten independent runs of the $(\mu + \lambda)$ GA using the configuration that the AC method wishes to evaluate. As previously mentioned, we use ERT and AUC as performance metric, and these values are computed from the 10 independent runs. Irace requires a set of instances for the tuning process. We imitate these *instances* by the independent runs of the (stochastic) solvers. This is in line with previous approaches, suggested, for example, in [32]. MIP-EGO starts with 10 initial candidates by the default setting of the package [155]. We use a $(4, 28)$ MIES, following the parameter settings suggested in [109].

To obtain a useful baseline against which we can compare other algorithms, we configure the $(\mu+\lambda)$ GA *on each PBO problem separately.* We consider $n = 100$ for all problems and take ERT or AUC (see Section 2.4) calculated from 10 independent GA runs as the cost metric, respectively. Also, we perform 100 independent validation runs with the suggested parameter settings from a configurator, which is meant to mitigate the randomness of the cost metric, thereby producing a fair empirical comparison. All experimental data discussed later are obtained using this validation procedure.

For evaluating GA candidates during the configuration process, the ERT values are calculated with respect to the targets listed in Table 5.2 and to the cutoff time of $50,000$ function evaluations. This is half the budget used in Section 5.2, but, according to the results presented there, our cutoff time is still larger than the number of function evaluations needed to hit the corresponding targets – except for F18, which is a very challenging problem.

For the AUC, the set of targets are 100 values, equally spaced in the interval $[\phi_{\min}, \phi_{\max}]$, where $\phi_{\max}$ is equal to the ERT targets listed in Table 5.2 and $\phi_{\min}$ is 0 except for the following functions: $\phi_{\min} = -19,590$ for function F22, $\phi_{\min} = -3,950,000$ for function F23, and $\phi_{\min} = -1$ for function F25. We evaluate the AUC for each point in $[50,000]$, i.e., in the notation of Definition 2.5 we use $T = [50,000]$.

The configuration space of the AC problem is $\Theta = \{\mu, \lambda, p_c, p_m\}$, where $\mu \in [100]$ is the parent population size, $\lambda \in [100]$ is the offspring population size, $p_c \in [0,1]$ is the crossover probability, and $p_m \in [0.005, 0.5]$ is the mutation rate. A positive crossover probability $p_c > 0$ requires $\mu > 1$, which otherwise renders a configuration infeasible. The results for the grid search are based on our work in Section 5.2, where we used $\mu \in \{10, 50, 100\}$, $\lambda \in \{1, \mu/2, \mu\}$, $p_c \in \{0, 0.5\}$, and $p_m = 0.01$. Note that this is a considerably smaller search space, whose full enumeration requires only 18 different configurations, which is much less than the budget allocated to the automated configuration techniques. We will nevertheless observe that for some problems none of the automated configurators could find hyperparameter settings that are equally good as those provided by this small grid search.

Table 6.1: Configurations of the $(\mu + \lambda)$ GA obtained by grid search, Irace, MIP-EGO, and MIES. Results for maximizing AUC are obtained independently from those obtained for minimizing ERT.

F	**C**	**Grid Search**				**Irace**				**MIP-EGO**				**MIES**			
		μ	λ	p_m	p_c	μ	λ	p_m	p_c	μ	λ	p_m	p_c	μ	λ	p_m	p_c
1	ERT	10	1	0.01	0.5	15	5	0.007	0.782	3	4	0.024	0.774	2	2	0.006	0.778
	AUC	10	1	0.01	0.5	1	1	0.006	0	3	11	0.006	0.208	2	2	0.005	0.521
2	ERT	10	1	0.01	0.5	1	39	0.006	0	2	9	0.006	0.008	2	3	0.005	0.026
	AUC	10	1	0.01	0	1	34	0.013	0	22	5	0.009	0.461	2	1	0.008	0.005
3	ERT	10	1	0.01	0.5	4	32	0.006	0.867	3	24	0.009	0.391	2	3	0.005	0.357
	AUC	10	1	0.01	0.5	4	23	0.011	0.550	2	3	0.007	0.511	2	1	0.006	0.203
4	ERT	10	1	0.01	0.5	1	1	0.013	0	2	3	0.022	0.655	2	2	0.024	0.198
	AUC	10	1	0.01	0.5	1	1	0.013	0	12	9	0.052	0.668	2	1	0.055	0.614
5	ERT	10	5	0.01	0.5	3	17	0.005	0.415	4	2	0.026	0.294	2	3	0.006	0.321
	AUC	10	1	0.01	0.5	8	8	0.012	0.882	11	20	0.008	0.318	2	2	0.006	0.358
6	ERT	10	10	0.01	0	30	23	0.104	0.849	3	11	0.031	0.033	24	12	0.135	0.908
	AUC	10	10	0.01	0.5	1	1	0.033	0	2	1	0.032	0.424	2	3	0.059	0.236
7	ERT	50	50	0.01	0	45	22	0.460	0.887	95	46	0.018	0.971	54	18	0.268	0.907
	AUC	10	10	0.01	0	1	58	0.028	0	2	20	0.016	0.082	23	27	0.025	0.672
8	ERT	10	10	0.01	0.5	25	48	0.014	0.643	78	7	0.087	0.991	5	9	0.009	0.507
	AUC	10	10	0.01	0.5	3	6	0.015	0.441	11	12	0.007	0.499	21	25	0.011	0.800
9	ERT	100	50	0.01	0.5	64	90	0.056	0.938	76	10	0.488	0.977	54	32	0.013	0.643
	AUC	50	25	0.01	0.5	46	40	0.029	0.815	6	7	0.020	0.158	18	25	0.025	0.505
10	ERT	100	50	0.01	0.5	95	69	0.325	0.978	87	84	0.478	0.962	79	17	0.056	0.320
	AUC	100	1	0.01	0.5	99	13	0.085	0.982	76	16	0.321	0.986	92	32	0.413	0.989
11	ERT	10	1	0.01	0	1	72	0.031	0	4	57	0.031	0.004	2	2	0.025	0.000
	AUC	10	1	0.01	0	1	13	0.035	0	2	1	0.040	0.003	2	1	0.042	0.016
12	ERT	10	1	0.01	0	1	32	0.006	0	4	1	0.012	0.016	2	1	0.005	0.000

	AUC	10	1	0.01	0	1	29	0.010	0	2	1	0.017	0.016	2	1	0.014	0.005
13	ERT	10	10	0.01	0	1	10	0.044	0	10	19	0.032	0.282	2	5	0.032	0.008
	AUC	10	10	0.01	0	1	1	0.044	0	2	4	0.038	0.023	3	8	0.047	0.036
14	ERT	100	100	0.01	0.5	1	60	0.343	0	57	34	0.077	0.988	57	23	0.028	0.012
	AUC	100	50	0.01	0	48	83	0.409	0.812	5	80	0.456	0.287	54	9	0.481	0.480
15	ERT	10	10	0.01	0	5	64	0.014	0.042	5	15	0.008	0.018	6	67	0.011	0.023
	AUC	10	10	0.01	0	21	71	0.020	0.165	3	9	0.016	0.078	2	3	0.011	0.024
16	ERT	10	10	0.01	0	6	34	0.011	0.025	2	2	0.006	0.069	2	4	0.005	0.002
	AUC	10	10	0.01	0	1	1	0.010	0.000	22	66	0.009	0.138	2	3	0.007	0.001
17	ERT	10	10	0.01	0.5	60	57	0.133	0.512	79	62	0.345	0.430	2	3	0.006	0.003
	AUC	10	10	0.01	0	1	1	0.019	0	2	5	0.012	0.007	2	4	0.019	0.034
18	ERT	100	50	0.01	0	50	44	0.006	0.010	13	77	0.019	0.002	26	21	0.006	0.002
	AUC	10	5	0.01	0	20	10	0.005	0.031	2	22	0.006	0.011	23	83	0.006	0.010
19	ERT	10	10	0.01	0.5	98	94	0.174	0.324	2	5	0.014	0.286	2	12	0.006	0.061
	AUC	50	50	0.01	0	1	5	0.005	0	7	18	0.014	0.834	2	7	0.005	0.133
20	ERT	50	50	0.01	0	1	32	0.008	0	6	53	0.007	0.012	5	66	0.005	0.033
	AUC	10	10	0.01	0	1	30	0.013	0	5	7	0.011	0.483	6	75	0.006	0.095
21	ERT	10	5	0.01	0	1	86	0.014	0	7	52	0.009	0.144	5	66	0.009	0.077
	AUC	10	10	0.01	0	4	33	0.006	0.412	7	9	0.006	0.833	2	3	0.005	0.219
22	ERT	100	100	0.01	0	16	48	0.025	0.783	18	12	0.019	0.820	2	3	0.005	0.270
	AUC	10	10	0.01	0.5	1	1	0.020	0	2	3	0.007	0.250	2	2	0.010	0.088
23	ERT	10	1	0.01	0	62	39	0.006	0.252	10	40	0.013	0.010	5	1	0.006	0.006
	AUC	10	5	0.01	0	29	25	0.005	0.016	4	4	0.006	0.170	8	16	0.005	0.038
24	ERT	100	50	0.01	0	53	85	0.025	0.004	58	45	0.031	0.077	21	75	0.060	0.014
	AUC	10	5	0.01	0	7	86	0.026	0.776	8	3	0.032	0.277	8	33	0.045	0.273
25	ERT	50	1	0.01	0	13	52	0.296	0.170	68	91	0.024	0.801	98	40	0.009	0.941
	AUC	10	10	0.01	0	26	53	0.028	0.790	4	3	0.023	0.107	9	1	0.021	0.535

Table 6.2: Absolute ERT and AUC values for the (1+1) EA and relative improvement of ERT and AUC for the configurations suggested by the four configuration methods, in comparison against the (1 + 1) EA values. Compared to the ERT and AUC values of the (1 + 1) EA on each problem (indicated by "EA"), the relative improvement obtained from the automated configuration the are shown for each AC method, where both measures are calculated from hitting times of 100 validation runs for (1 + 1) EA and AC methods. We also indicate the statistical significance in the empirical distributions of the hitting time (**** for $p < 0.001$, *** for $p < 0.01$, ** for < 0.01, and * for $p < 0.05$) for each pair of the AC result and that of the (1 + 1) EA. The Mann–Whitney U test is applied with the Benjamini and Hochberg method for all 120 pairwise comparisons to control the false discovery rate. Runs are cut off at 50 000 evaluations if it does not hit the final target. The significant comparisons are colour-coded with respect to the relative improvement, where a darker colour signifies a more considerable improvement.

F	**ERT**					**AUC**				
	EA	**GS**	**Irace**	**MIP-EGO**	**MIES**	**EA**	**GS**	**Irace**	**MIP-EGO**	**MIES**
1	665	-0.54****	-0.47****	-0.34****	0.01	0.9987	-0.11****	-0.49**	-1.21****	-0.70
2	5,574	-0.64****	-0.12****	-0.03	0.05	0.9514	-1.37****	-0.51****	-4.64****	0.25
3	694	-0.62****	-0.53****	-0.78****	0.03	0.9989	-0.11****	-1.12****	-0.57	-0.50
4	344	-0.87****	-0.04	-0.30****	-0.11*	0.999	-0.06****	-0.66	-1.20****	-0.73****
5	598	-0.50****	-0.41****	-0.55****	0.04	0.9987	-0.11****	-1.17****	-1.90****	-0.68**
6	271	-1.96****	-0.91****	-0.58****	-7.66****	0.9993	-0.11****	-0.43	-1.32****	-0.54***
7	Inf	—	—	—	Inf	0.8995	-0.33	1.69	0.70	0.24
8	7,926	0.78****	0.77****	-0.01****	0.86****	0.9938	0.25****	-0.37****	-0.65****	-0.75****
9	22,670	0.85****	0.85****	0.15****	0.77****	0.9877	0.60****	0.42****	0.50****	0.41****
10	Inf	Inf****	Inf****	Inf****	Inf****	0.6362	54.86****	55.60****	55.50****	55.73****
11	2,071	-0.29****	-0.35****	-0.45****	0.12**	0.9807	-0.36****	-1.05	-0.89	-0.93
12	4,691	-0.45****	-0.11***	-0.12***	0.06*	0.9601	-1.51****	-0.64***	-0.15*	0.13
13	997	-0.83****	-0.14*	-0.70****	-0.05	0.9902	-0.57****	-0.8	-0.99	-1.26****
14	8,171	0.98****	0.99****	0.98****	0.96****	0.596	47.69****	47.29****	36.75****	41.05****
15	6,668	-1.46****	-1.17****	-0.50****	-1.23****	0.9404	-7.04****	-8.98****	-2.05****	-0.88****

16	9,520	-2.05****	-0.97****	-0.43****	-0.23****	0.9174	-17.54****	0.41	-12.97****	-1.78****
17	45,964	-Inf****	-Inf****	-Inf****	-0.96****	0.6698	-48.43****	2.45***	-14.40****	-7.93****
18	130,863	0.85****	0.80****	0.69****	0.85****	0.9432	1.86***	1.32	0.06	1.26****
19	9,467	-525.05****	-Inf****	-1.72****	-0.72****	0.9899	-6.19****	0.00	-4.96****	-0.37**
20	1,460	-9.77****	-0.83****	-2.16****	-2.47****	0.9956	-1.26****	-0.20****	-0.25****	-0.68****
21	948	-12.75****	-3.37****	-3.66****	-3.95****	0.9965	-0.40****	-0.56****	-0.42****	-0.04****
22	3,366	-2.60****	-0.11****	-0.57****	-0.15	0.9993	-0.02****	-0.05****	-0.06	-0.02****
23	3,066	0.08***	-1.00****	-0.68****	-1.03*	0.9993	0.01**	-0.12****	-0.40***	-0.23****
24	Inf	Inf****	Inf****	Inf****	Inf****	0.9545	1.11**	1.42****	0.90	1.71****
25	48,946	0.22*	-Inf****	0.66****	0.56****	0.6953	0.07**	-0.00****	0.04	-0.03**
#improvements		8	6	6	13	-	8	9	7	8

6.3.2 Results Obtained by Automated Configuration

The $(1+1)$ EA with $p_m = 1/n$ has shown competitive results in [55] for the PBO problems, so we use it as the baseline against which we compare the GAs obtained by the configurators. Although it is part of the GA framework in Algorithm 8 and could therefore be identified by the configuration methods, we still manually add it as a baseline and we compare the results obtained by the three configurators to it. Table 6.1 lists the configurations of the $(\mu+\lambda)$ GA obtained by the grid search and by the three AC methods. Table 6.2 compares the performance of these configurations, by listing the ERT and AUC values of the $(1+1)$ EA and the corresponding relative deviations of the configured GAs. More precisely, the ERT and AUC values of the AC methods result from using ERT and AUC as the cost metric, respectively. The relative improvement of ERT is computed as ($\text{ERT}_{(1+1)-\text{EA}}$ - ERT) / $\text{ERT}_{(1+1)-\text{EA}}$, and the relative improvement of AUC is computed as (AUC - $\text{AUC}_{(1+1)-\text{EA}}$) / $\text{AUC}_{(1+1)-\text{EA}}$. Grey tiles indicate that the result is better than the corresponding result of the $(1+1)$ EA, and the degree of gray represents the degree of improvement. We use in this table the Mann-Whitney U test to compare the average running times of the (1+1) EA and the GAs suggested by the configuration methods (pairwise comparisons). Runs that did not hit the final target within the cutoff time of $50,000$ evaluations are capped at this value. Asterisks in Table 6.2 indicate that the average hitting time of the GAs suggested by the corresponding configuration methods is significantly different from the average hitting time of the $(1+1)$ EA.

ERT Results

For the OneMax-based problems F1 and F4-F6, configurations of the $(\mu+\lambda)$ GA using crossover outperform the mutation-only GA with $\mu \geq 10$ [170]. However, according to the values in Table 6.2, the $(1+1)$ EA outperforms the configurations with $p_c > 0$, relatively large μ, and also relatively large λ. This observation matches our expectation because our previous study has shown that the $(1+1)$ EA is efficient on OneMax. Meanwhile, we observe an interesting configuration with $p_m < 0.01, \mu = \lambda = 2$, and $p_c > 0$ that achieves competitive ERT values against the $(1+1)$ EA for F1. This configuration ties well with the results on different $(\mu+1)$ GAs that were shown to outperform the $(1+1)$ EA (and any mutation-based algorithm, in fact) in a series of recent works [25, 29, 143].

On F4, F6, F7, F13, F15-F17, and F19-22, none of the configurations returned by the AC methods was able to outperform the $(1+1)$ EA, whereas on F9-10, F14, F18,

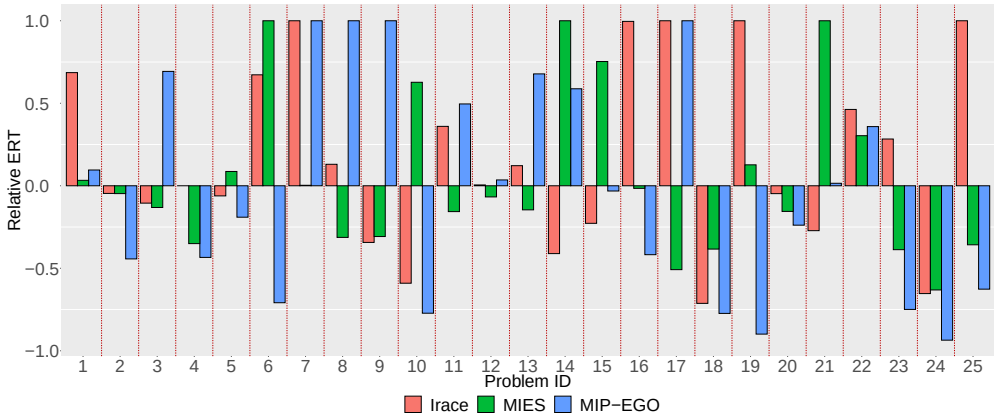


Figure 6.1: Relative ERT values of the GAs obtained by Irace, MIES, and MIP-EGO using ERT as the cost metric compared to the ERT of the GAs obtained by the same method when using AUC as the cost metric during the configuration process. Plotted values are $(\text{ERT}_{\text{using ERT}}\text{-}\text{ERT}_{\text{using AUC}})/\text{ERT}_{\text{using AUC}}$, capped at -1 and 1. Positive values therefore indicate that configurator obtains better results when configuring AUC.

F24, all AC methods find configurations that perform much better than the (1+1) EA.

On LeadingOnes, a slightly better result compared to the ERT value 5 574 of the (1+1) EA is found by MIES. This result is quite sensitive with respect to the mutation rate. When changing it from the MIES-suggestion of $p_m = 0.005$ to $p_m = 0.01$ we obtain an ERT value of 5 829. MIES also obtains an improvement on F11, which corresponds to a $(2+2)$ GA with $p_m = 0.0245$. On F14, we already observed in [170] that mutation-only GAs with $p_m = 1/n$ are inferior to other GA configurations with a larger offspring population size and higher mutation rate. As expected, all three methods easily suggest configurations that outperform the $(1+1)$ EA by a great margin.

On F18 and F24, all configurators unanimously suggest fairly small values for the crossover probabilities. For F25, however, GAs with $p_c > 0.8$ show the (by far) best performance.

AUC Results

Since we evaluate the AUC at each budget $[50, 000]$, the AUC values tend to be very close to 1, especially for the GAs that require much fewer than $50, 000$ evaluations to find an optimal solution.

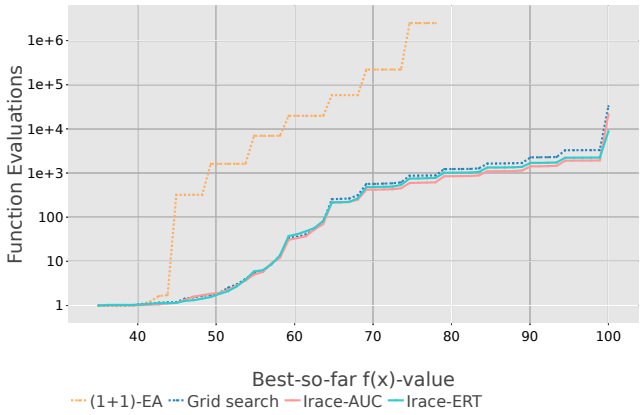


Figure 6.2: Fixed-target ERT values of the configurations suggested for F10. The suffix "-ERT/AUC" indicates which cost metric is used

On 13 out of the 25 problems, none of the configuration methods is able to identify a hyperparameter setting that yields better AUC values than that of the (1+1) EA. In some, but clearly not all of these cases, the achieved AUC values are not much worse than that of the $(1+1)$ EA. The largest improvements are obtained on functions F7, F10, F14, F17, F18, and F24. In most of these cases all four configuration techniques found improvements over the (1+1) EA except for F17, where Irace is the only method finding an improvement and except for F7, where all three automated configuration techniques find an improvement but not the grid search (the inverse is true on F23 but the advantage of the grid search is fairly small, not statistically significant, and the obtained algorithm is a mutation-only (10+1) GA with $p_m = 1/n$, which is very close to the $(1+1)$ EA).

We next discuss the results for the functions for which large improvement over the $(1+1)$ EA were obtained.

On **F7**, the configuration obtained by Irace (which achieves the best improvement) is a $(1+\lambda)$ EA with $p_m \approx 3/n$, the one obtained by MIP-EGO is a $(2+20)$ GA with small crossover probability $p_c = 0.082$, and the one obtained by MIES is a (23+27) GA with large crossover probability $p_c = 0.672$. These results indicate that the fraction of configurations achieving better AUC value than the $(1+1)$ EA may be fairly large.

On **F10**, large crossover probabilities seem beneficial; all three automated configuration techniques return settings with $p_c > 0.98$. The mutation rate seems to have less impact on the results, and the suggested settings vary from $8.5/n$ (Irace) to $41.3/n$

(MIES, the best AUC value). Interestingly, also two of the three configurations tuned for minimizing ERT have large crossover probabilities, with the exception of the one returned by MIES ($p_c = 0.32$). The performances of all these configurations are very similar, as we can see in Figure 6.2.

On **F14**, the best improvement is obtained by a mutation-only (100+50) GA using $p_m = 1/n$, closely followed by the Irace result, which is a (48+83) GA using crossover probability $p_c = 0.812$ and $p_m \approx 41/n$. We cannot observe any clear pattern in the results, and the four suggested configurations all differ quite a bit.

For **F17**, as mentioned, only Irace finds a better configuration, which is also a (1+1) EA, but using a slightly larger mutation rate of $p_m = 1.9/n$ instead of $1/n$.

For **F18**, all configurators return settings with small crossover probability $p_c < 0.031$ and small mutation rate $p_m < 0.01$, which indicates that local search methods may be more suitable for this problem. Interestingly, the performance of randomized local search is not very good (see [55] for details), which suggests that a positive probability for escaping local optima via small jumps in the search space or via crossover are needed to be efficient on this problem.

For **F24**, no clear pattern can be observed in the suggested configurations, and also the crossover probabilities differ widely, from 0 for the grid search, values around 0.27 for MIP-EGO and MIES, to 0.7 for Irace.

On the LeadingOnes problem F2, the $(2 + 1)$ GA with $p_m = 0.008$ and $p_c = 0.005$, found by MIES, yields a (small) improvement over the $(1 + 1)$ EA, whereas the configurations found by the other methods perform worse.

All in all, we find that on several problems the suggested configurations differ widely, far more than we would have expected and this across all four parameters. Analyzing the landscape of the AC problem suggests itself as an interesting follow-up study, which would require an effort that goes beyond the scope of this work. However, we have seen related work being conducted in [129, 146]

6.3.3 Discussions on the Configurators' Performance

We now compare the performance of the three automated AC methods. The last row of Table 6.2 summarizes for how many settings each method was able to find configurations that outperform the $(1 + 1)$ EA. These numbers are rather balanced between the different methods, with the notable exception of the minimizing ERT objective, for which MIES suggested 13 improvements, compared to 6-8 improvements found by the other methods. MIES also suggested the best configurations in most

of the cases, but the improvements over the $(1 + 1)$ EA are, however, rather minor in several of these cases, so that barely counting them does not give justice to the complex behavior observed in Table 6.2, from which we cannot derive a clear winning configurator. We can nevertheless make a few observations.

Handling Conditional Parameter Spaces

We easily see from Table 6.1 that Irace is the only method that obtains mutation-only GAs, and in all of these cases it returns a $(1 + \lambda)$ EA. We recall that setting $\mu = 1$ requires to set $p_c = 0$; the configuration is infeasible otherwise. This advantage of Irace lies in its handling of conditional parameters: Irace samples non-conditional parameters first, and samples conditional parameters only if the condition is satisfied. In contrast, the two other AC methods, MIP-EGO and MIES, sample parameter values from independent distributions and give penalties to infeasible settings. With this strategy, the two methods can avoid infeasible candidates, but the probability of sampling feasible conditional candidates may be too small. For example, MIP-EGO can find a configuration with $\mu = 2$ and $p_c = 0.0065$ on F16 in Table 6.1, but it cannot obtain the competitive configuration of $(1 + \lambda)$ mutation-only GA because the probability of sampling $\mu = 1$ and $p_c = 0$ simultaneously is too small. We observe a similar performance of MIES on F17.

Impact of the Cost Metric

We have already observed that MIES obtains better configurations for more problems when using ERT as the cost metric. For AUC, in contrast, Irace finds more configurations that improve over the $(1 + 1)$ EA, which can be explained as follows. In the first few iterations, AUC is able to differentiate the performance of two poor configurations if both fail to find the final target, whereas the ERT value will be infinite and thereby incomparable in this case. Hence, using AUC as the cost metric, Irace could learn to avoid evaluating those poor configurations in the following iterations. It is worth noting that such an observation is also supported by a case study of Irace [127], in which the authors discovered that Irace would spend too much time on poor configurations if the mean running time is taken as the cost metric. As a solution, the adaptive capping strategy [83] is introduced to Irace in this work. Interestingly, this discussion connotes that the AUC metric realizes a similar effect as with adaptive capping for minimizing the running time of an optimization algorithm. This behaviour also indicates that the choice of the cost metric might be a factor to consider when choosing which AC

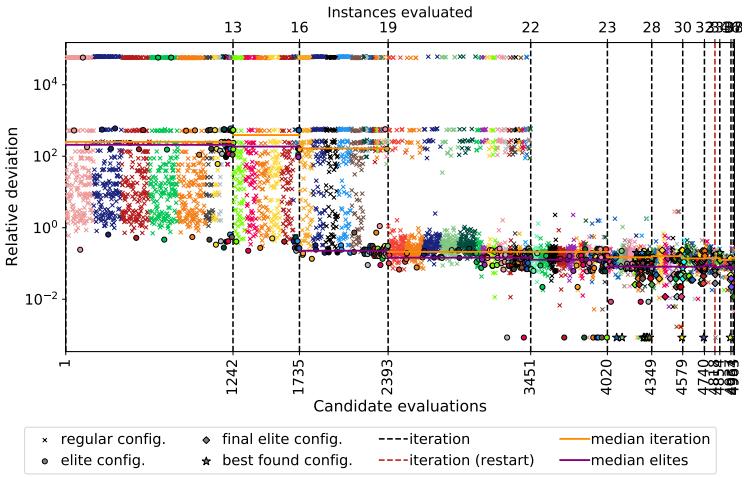


Figure 6.3: The relative deviation from the best-known ERT value of the GAs obtained during the configuration process of Irace for tuning the $(\mu + \lambda)$ GA for OneMax in dimension $n = 100$, with the objective to minimize the ERT for the optimum $f(x) = 100$. The maximal number of configurations that can be tested by Irace is set to 5,000. The figure is produced by the acviz tool [36].

technique to apply.

For an algorithm that cannot hit the target in all runs, the variance of its ERT values can be high due to the uncertain success rate. Besides, ERT can not distinguish algorithms that can not hit the target in any runs, even though their performance may differ in terms of results for other targets. This shortcoming is mitigated when tuning for large AUC, since this performance metric also takes into account the hitting times for easier targets.

Figure 6.3 plots the relative deviations from the best-known ERT value of the configurations obtained during one run of Irace when using ERT as the cost metric. We observe that many configurations show large relative deviation values, which stem from GAs that cannot hit the optimum within the given budget. These configurations do not provide much useful information, since they all look equally bad for the configurator.

6.3.4 The Choice of the Cost Metric

We now evaluate how well configurations that are obtained by tuning for the AUC cost metric perform in terms of ERT. Figure 6.1 summarizes these result, by plotting the relative advantage of the configurations tuned for AUC, compared to

those that were explicitly tuned for ERT. More precisely, we plot ($\text{ERT}_{\text{using ERT}}$-$\text{ERT}_{\text{using AUC}}$)/$\text{ERT}_{\text{using AUC}}$, so that positive values indicate that tuning for AUC give better ERT values than the configurations obtained when tuning for ERT. We see that this is the case for 13, 12, and 9 out of the 25 problems when using Irace, MIP-EGO, and MIES, respectively.

We now zoom into the results obtained by Irace. We abbreviates "Irace-ERT" ("Irace-AUC") the configurations obtained when using ERT (AUC) as cost metric. For the problems on which the ERT of Irace-AUC was worse than that of Irace-ERT, we plot in Figure 6.6 violin plots for the running times of the 100 validation runs. We observe that F15 is the only problem where Irace-ERT significantly outperforms Irace-AUC. On F2-3, F5, and F20, we observe that the result of most runs of Irace-AUC and Irace-ERT are close, but the variances of the results of Irace-AUC are higher than for Irace-ERT. On the remaining problems, we observe high variances for the result of both Irace-ERT and Irace-AUC. Irace-ERT finds the configurations with fewer *unsuccessful* runs, which makes sense because the number of *unsuccessful* runs significantly affects the ERT value. However, AUC does not only consider the evaluations needed to hit the final target, so we observe more *unsuccessful* runs and *competitive* partial runs for Irace-AUC, i.e., in cases of F21 and F24.

Although Irace-AUC does not obtain better ERT values than Irace-ERT, it can still provide valuable insights concerning the resulting configurations and performance profiles. Figure 6.4 plots the fixed-target ERT values of the GAs obtained by Irace-ERT and Irace-AUC for F21. We observe that the result of Irace-ERT outperforms the result of Irace-AUC for the final target $f(x) = 260$. However, for the long period when $f(x) < 258$, Irace-AUC performs better. This observation indicates that configuring AUC can provide novel instances to investigate how the GA performs during the optimization process.

We plot in Figure 6.7 the violin plots of the running times for the problems where Irace-AUC obtains better ERT values than Irace-ERT. The advantage of Irace-AUC is significant on several problems, i.e., F1, F6, F11, F16, and F22. Moreover, Irace-ERT can not find the final targets of F7, F17, F19, and F25 within the cutoff time, whereas Irace-AUC hits the targets in some (F7, F17, and F25) or all (F19) of the runs.

Figure 6.5 plots the fixed-target result of different GAs on F8. Compared to Irace-ERT, we observe that Irace-AUC outperforms the other algorithm at the final target and also exhibits advantages over other algorithms in most of the optimization process. Figure 6.8 plots the fixed-target result of different GAs on F7. The figure shows that Irace-AUC is the only one that hits the optimum $f(x) = 100$, and the best-found

fitness of Irace-ERT is less than 90. We observe that none of the GAs shown in the figure hits the optimum in all runs (because the ERT values are larger than the cutoff time of 50,000 function evaluations).

The results of Irace-AUC and Irace-ERT on the PBO problems reveal the questions of how the cost metric affects the performance of Irace for different configuration tasks for future study. We study in the following the impact of the cutoff time concerning the behavior of Irace on OneMax and LeadingOnes.

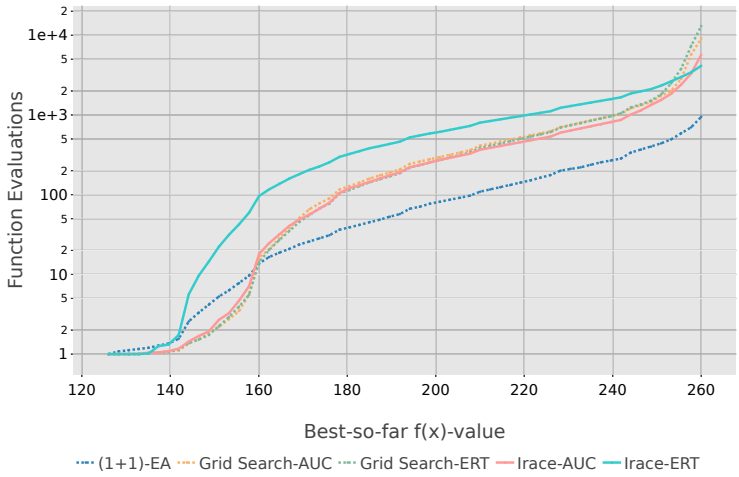


Figure 6.4: Fixed-target ERT values of the GAs listed in Table 6.1 for F21.

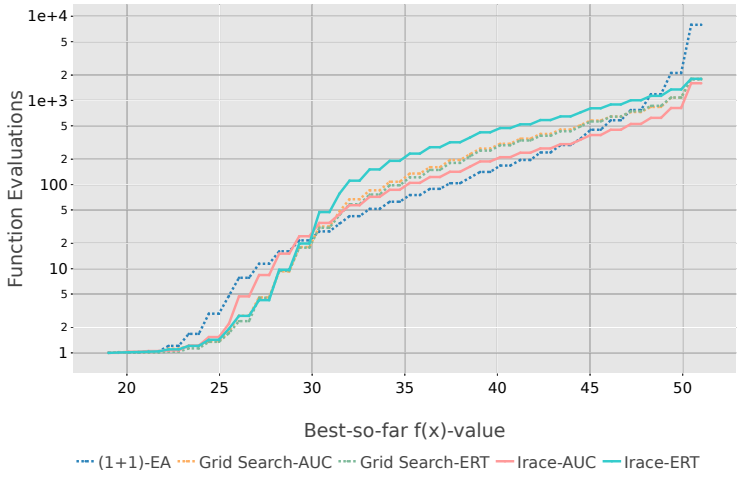


Figure 6.5: Fixed-target ERT values of the GAs listed in Table 6.1 for F8.

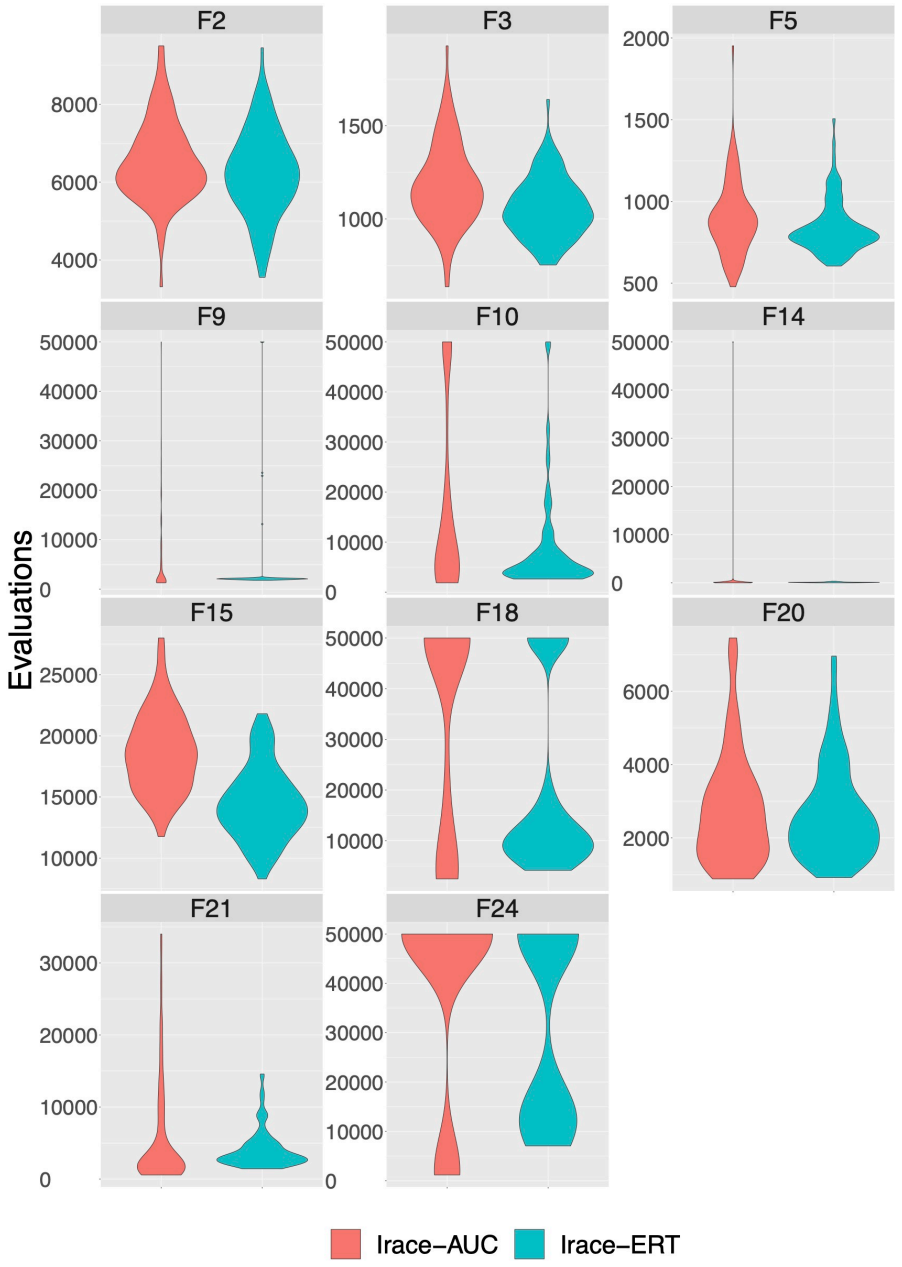


Figure 6.6: Violin plots of first runtimes hitting the targets for the configurations found by Irace when tuning for ERT and AUC, respectively. Only showing results for problems on which Irace-ERT outperforms Irace-AUC. Results are from the 100 independent validation runs. Targets are listed in Table 5.2, and the configurations of the GAs can be found in Table 6.1. For each run, values are capped at the budget $50,000$ if the algorithm can not find the target.

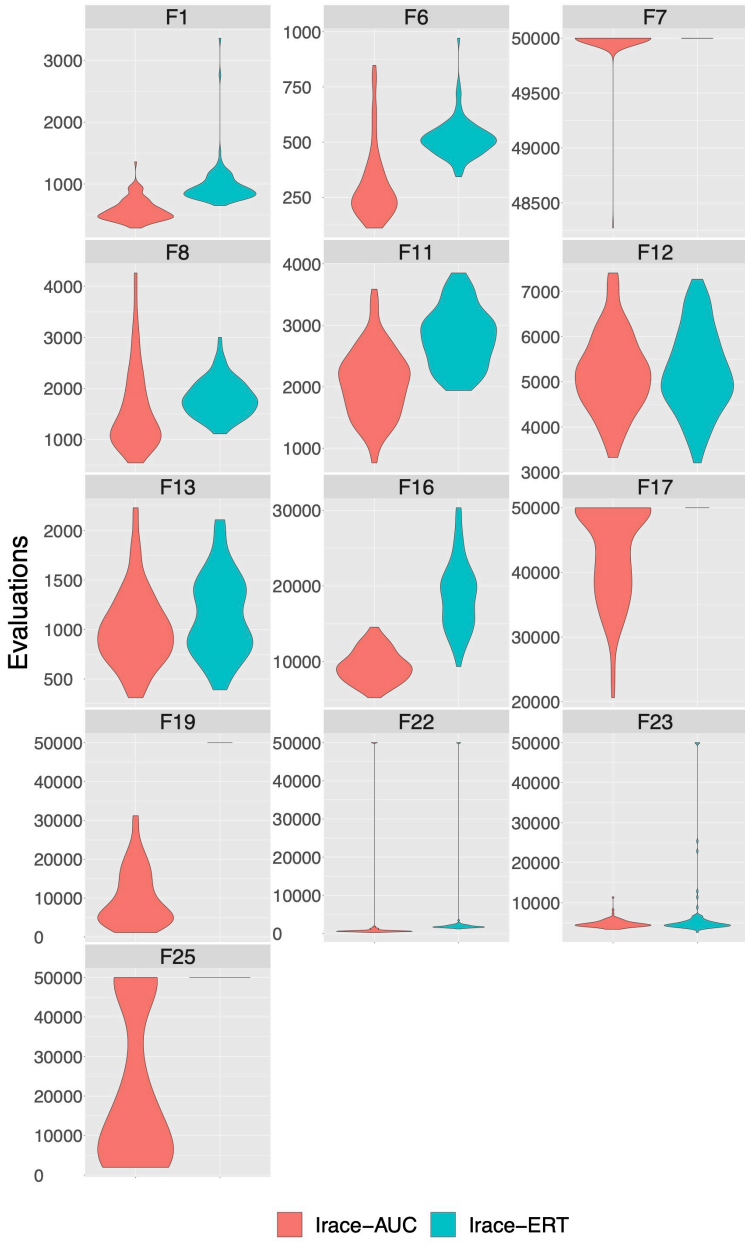


Figure 6.7: Violin plots of first runtimes hitting the targets for the configurations found by Irace when tuning for ERT and AUC, respectively, for problems on which Irace-AUC outperforms Irace-ERT. Results are from the 100 independent validation runs. Targets are listed in Table 5.2, and the configurations of the GAs can be found in Table 6.1. For each run, values are capped at the budget $50,000$ if the algorithm can not find the target.

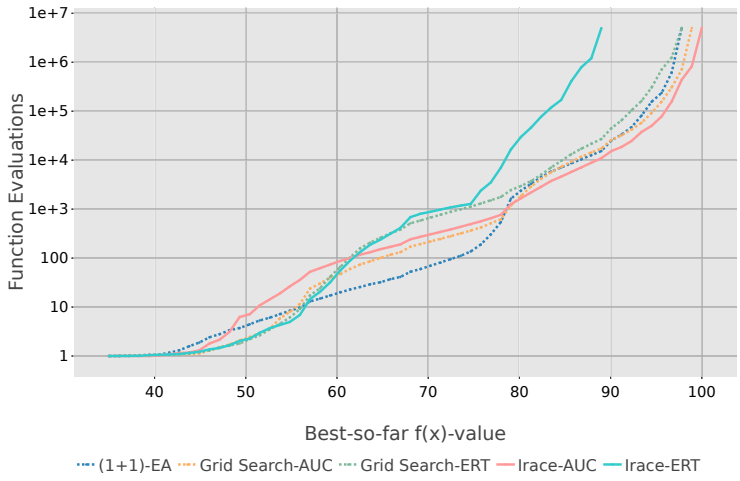


Figure 6.8: Fixed-target ERT values of the GAs listed in Table 6.1 for F7.

Sensitivity with respect to the cutoff time

Inspired by the result in Figure 6.8, we study the sensitivity of ERT and AUC with respect to the cutoff time of the GAs. To this end, we consider the set $\{(0.5+0.1t) \times \mathrm{ERT}_{(1+1)\ \mathrm{EA}} \mid t \in [0..15]\}$ of 16 different cutoff times. For each of these cutoff times, for each of the two cost metrics (AUC and ERT), and for each of F1 and F2, we run Irace 20 independent times with the same configuration budget of 5,000 target runs (where each target run corresponds again to ten independent runs of the respective $(\mu + \lambda)$ GA configuration). Figure 6.9 plots ERT values of the GAs obtained this way (as before, each ERT value is based on 100 independent validation runs). For comparison, the red line indicates the performance of the $(1 + 1)$ EA.

On OneMax, we observe that Irace-ERT can not find promising configurations when the cutoff time of the GAs is too small to hit the optimum. This is the case for cutoff time of the budgets smaller than 665. However, Irace-AUC can work with small budgets that are not sufficient to hit the optimum. Even with the cutoff time of the budgets larger than 665, Irace-AUC still obtains better ERT values than Irace-ERT. Similarly, the result for LeadingOnes shows that Irace-ERT cannot find promising configurations with insufficient cutoff time of GAs. Still, Irace-AUC performs well across all given cutoff time.

Overall, we thus see that tuning with respect to AUC is much less sensitive with respect to the cutoff time.

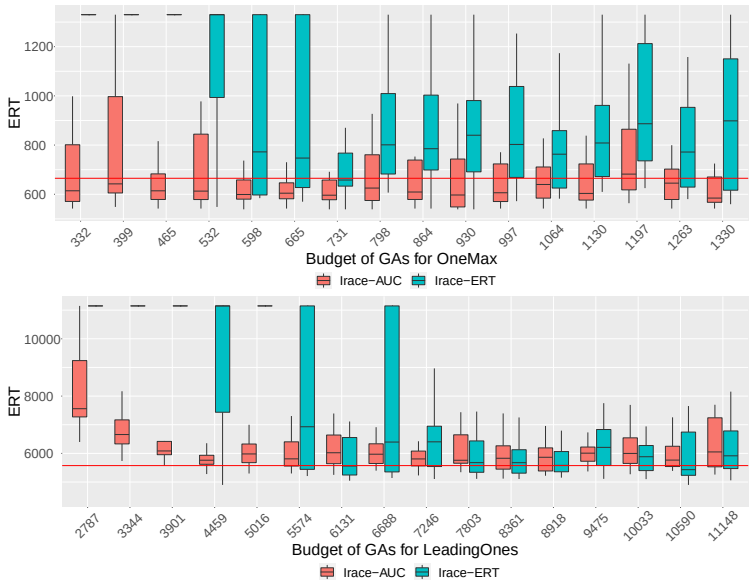


Figure 6.9: ERT values (y-axis) of the GAs obtained by Irace for OneMax and LeadingOnes in dimension $n = 100$, for different budgets B that the GAs can spend to find the optimum (x-axis). Showing results for $B \in \{(0.5 + 0.1t)\text{ERT}_{(1+1)\text{ EA}} \mid t \in [15]\}$. For comparison, the ERT values of the $(1 + 1)$ EA are plotted by horizontal red lines. Results are for the best found configurations obtained from 20 independent runs of Irace, and each of the ERT values is with respect to 100 independent validation runs.

Sensitivity with respect to the configuration budget of Irace

We also analyze the sensitivity of the results with respect to the configuration budget, i.e., the number of target runs that the configurator can perform before it suggests a configuration. We use Irace for this purpose. Figure 6.10 plots the ERT values of the configurations suggested by Irace, for 8 selected problems from the PBO suite. Interestingly, the ERT values are not monotonically decreasing, as one might have expected, at least for the configurations that are explicitly tuned for small ERT. Tuning for AUC gave the best ERT values for F1, 8, 19, 20, and 21.

6.4 Summary

In this chapter, we extended the analysis on the performance of a family of $(\mu + \lambda)$ GAs, based on the work of Section 5.2. Four different configuration methods have

been applied for finding promising configurations of GAs: the grid search and three automated techniques.

The experimental results showed that mutation-only GAs usually benefit from small parent population size. On the contrary, crossover-based GAs require sufficient population sizes. On the PBO problem set, the (1+1) EA outperforms the other tested GAs on OneMax, LeadingOnes, and some of their W-model extensions. However, crossover can be beneficial for the W-model extensions with epistasis and ruggedness, concatenated trap, and NK-landscapes.

We have also investigated the performance of AC methods: Irace, MIP-EGO, and MIES. Irace is the only method that has found conditional configurations of $(1 + \lambda)$ mutation-only GAs. It handles the non-conditional parameters first and samples the conditional parameters when the condition is satisfied, but the other two automated methods sample all parameters independently, leading to worse results in cases where mutation-only is beneficial.

We also observed that the cost metric used as tuning objective has a major impact on the performance of AC methods. When using ERT, the AC methods cannot obtain useful information from configurations that cannot hit the optimum. But not only for these cases we observed that tuning for AUC gave better ERT values than when directly tuning for ERT.

Our results have also demonstrated that none of the configuration methods clearly outperforms all others, suggesting to either combine them or to develop guidelines that can help users select a most suitable configuration technique for their concrete problem at hand. Finally, we also observe that in several cases none of the techniques could find configurations that outperform or perform on par with the (1+1) EA, which may indicate improvement potential for these configuration methods.

Figure 6.10: ERT values (y-axis) of the GAs obtained by Irace with different configuration budgets B_T (the number of configurations that Irace can test, x-axis). Results are for $B_T \in \{(0.5 + 0.25t)5,000 \mid t \in [4]\}$. Each ERT value is for the 100 validation runs of the configuration suggested by Irace after a single run, i.e., one for each budget.

Chapter 7

Dynamic Algorithm Selection

We analyze in this chapter how well existing benchmark data can be used for the selection of suitable algorithm combinations. Precisely, we investigate using dynamic crossover probability for the $(\mu + \lambda)$ GA and study "one-shot" dynamic algorithm selection (dynAS) policies based on the results of the benchmarking study presented in Section 5.2. The study in this chapter highlights the research topics for the future work of dynAS, namely automatic detection of alternating timing and "warm-start" strategies for adjusting parameters.

7.1 Background

It is well known that different algorithms or different instantiations of the same algorithm are best suited for different problems and even for different stages of the optimization process. Automated algorithm selection [95] as well as dynamic parameter selection [92] are therefore intensively studied meta-optimization problems in EC. However, the former has a strong requirement on being able to run different algorithms (or algorithm configurations) prior to making a decision which algorithm to apply to the problem at hand. Parameter control and related concepts (including hyper-heuristics, adaptive operator control, etc.), in contrast, assume that the selection has to be made on the fly, without leveraging existing data from previous or related runs. With the rise of artificial intelligence methods, EC is currently facing a paradigm shift, in that we aim to actively exploit existing performance data to select which algorithms to apply, and how to possibly adjust them during the run. We are, however, still far from achieving a fully automated informed online selection.

We study in this chapter how well we can predict from existing performance data which algorithm instances to combine for a given problem at hand. While we do allow for switching between different algorithms, the decision when to switch has to be made prior to the run, and depends, in our case, on the solution quality of the evaluated solution candidates. More precisely, we use the benchmarking data in Section 5.2 as starting point to investigate, for each of the 25 individual problems, how well we can predict which single-switch algorithm combinations would show good performance. For some functions we easily obtain algorithm combinations that outperform the best static algorithms. For other functions the results are rather mixed. On three functions, none of the 100 tested single-switch algorithm combinations was able to outperform the best static solver. The prediction quality of the approach suggested in [153] varies a lot between the different functions. While for LeadingOnes, for example, the performance predictions are rather accurate, large discrepancies between predicted and actual performance can be observed for more complex function. In particular for multi-modal functions the approach can get trapped by a first algorithm that is very efficient in converging to a local optimum from which the second algorithm cannot escape easily.

7.2 Dynamic Crossover Probability Selection: A Study Case on LeadingOnes

Based on our finding in [170] (see Section 4.3) that, on LeadingOnes, the optimal crossover probability of Algorithm 8 is dynamic along the problem dimension and population size, we start in this section with an investigation of using dynamic crossover probabilities for the $(\mu + \lambda)$ GA.

To obtain the *"optimal"* crossover probability at different stages of the optimization process, we test the $(10 + 10)$ GA using standard bit mutation with $p = 1/n$ and uniform crossover with different $p_c \in \{0.1k \mid k \in [9]\}$. Algorithms run at the stages of fitness value $f \in [s, s + 5], s \in \{5i \mid i \in [19]\}$ on 100-dimensional LeadingOnes. Practically, we initialize the population of the GAs with all the individual's fitness values equal to s, and the algorithms terminate once a solution with $f(x) \geq s + 5$ is found.

Figure 7.1 plots function evaluations used by the GAs at each stage. It shows that the GA with $p_c = 0$ uses the least function evaluations at the early stages $s \leq 40$, but other GAs with $p_c > 0$ use less function evaluations as s is increasing. Therefore, we

expect to improve the performance of the $(\mu+\lambda)$ GA by using the *"optimal"* crossover probabilities at all the stages of the optimization process. Figure 7.2 plots the fixed-target ERTs of GAs with static p_c and dynamic ones. The dynamic policy selects the corresponding best p_c at each stage. Practically, when the GA finds a solution with $s_1 \leq f(x) < s_2, s_2 = s_1 + 5, s \in \{5i \mid i \in [19]\}$, the p_c will be adjusted by the corresponding best value in Figure 7.1.

Figure 7.2 shows that the GA with dynamic p_c outperforms other GAs along the entire optimization process. The dynamic policy successfully hits the optimum $f(x) = 100$ using the smallest ERT $7,194$, while ERT of the best runner-up (the GA with $p_c = 0.2$) is $7,661$. This corresponds to a 6% improvement of the dynamic GA over the best static one. This performance empirically proves that the GA can benefit from dynamic crossover probability, and it displays a successful case of applying the dynAS for the GA. However, the dynAS problem is not usually coming with the ideal condition that candidate algorithms differ by only one parameter. Therefore, we are working on the GAs with more combinations of parameters in the next section.

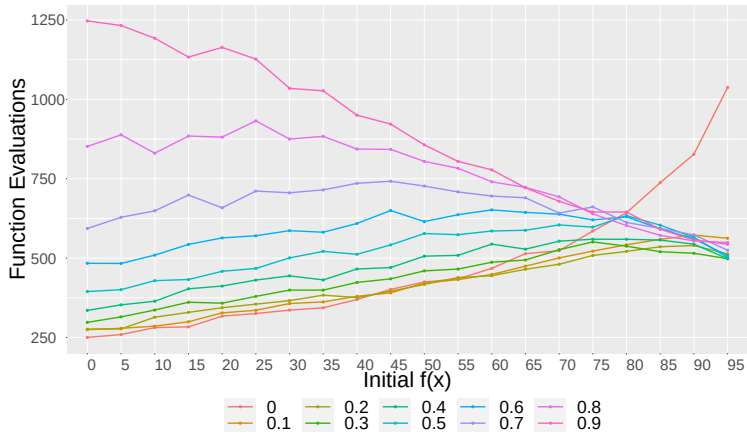


Figure 7.1: Average number of function evaluations needed by different $(10+10)$ GAs to find a solution y with $f(y) \geq s+5$ on the 100-dimensional LeadingOnes function when all ten points in the initial population are uniformly chosen from the set of points x that satisfy $f(x) = s$, for $s \in \{5i \mid i \in [19]\}$. The GAs differ only in the crossover probability $p_c \in \{0.1k \mid k \in [9]\}$ (different lines). Results are averaged of $1,000$ independent runs. The connecting lines are only meant to help visual interpretation, the data points are only at the values 0, 5, 10, ..., 95.

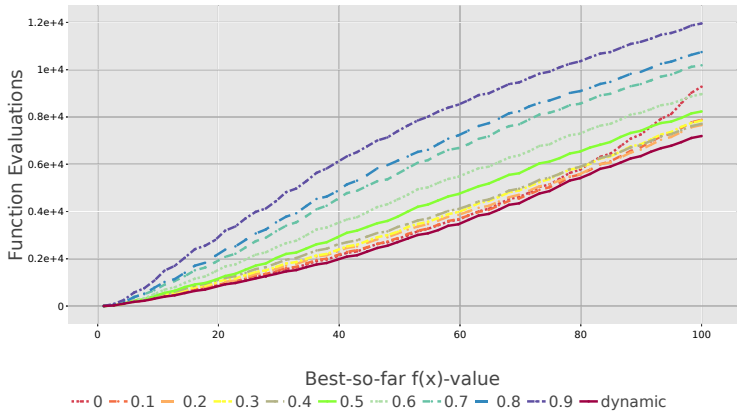


Figure 7.2: Fixed-target ERTs of GAs on 100-dimensional LeadingOnes. The legend presents values of p_c, and the *dynamic* method adjusts its p_c to the *optimal* value at each target $f(x) = s, s \in \{5i \mid i \in [19]\}$, based on the result in Figure 7.1. Results are average of 100 independent runs.

7.3 Dynamic Algorithm Section for the PBO Problems

Since the LeadingOnes case shows significant improvement by using dynamic crossover probabilities, which is a particular case of the dynAS, we study the behavior of the dynAS on a broader range of problems and GAs. We take as input the benchmarking data from Section 5.2, which comprise detailed performance records for 80 genetic algorithms on the 25 functions provided by IOHprofiler. We focus on ERT as performance measure. Detailed data can be found in [166].

Following the approach suggested in [153] we compute a "theoretical" ERT value for all combinations (A_1, A_2, ϕ_s), where A_1 is the first algorithm, A_2 the second, and ϕ_s the target value at which we switch from A_1 to A_2. To this end, we simply compute $\mathrm{ERT}(A_1, P, \phi_s) + \mathrm{ERT}(A_2, P, \phi_f) - \mathrm{ERT}(A_2, P, \phi_s)$, where all these ERT values are based on the performance records provided in Section 5.2. In total, we consider 42 possible switching points ϕ_s, which we select within the interval $[\phi_m, \phi_f]$ between the smallest fitness value ϕ_m of the problem and the best found target ϕ_f according to Table 5.2. We consider evenly spaced targets, for the original and for the log-scaled interval, respectively. For each problem, we consider only algorithms that hit the final target value with probability at least 80% according to the data from Section 5.2. Using this approach, we select for each problem the 100 best combinations (A_1, A_2, ϕ_s)

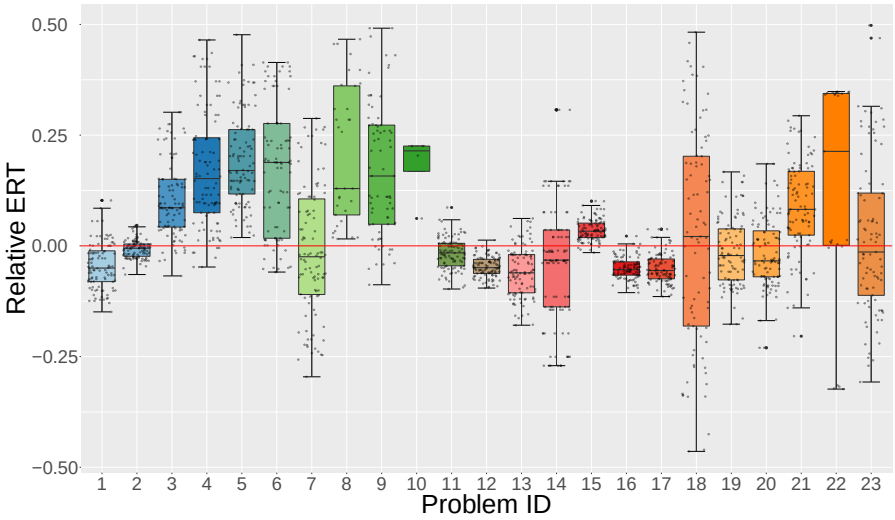


Figure 7.3: Relative ERT values of 100 1-switch combinations (A_1, A_2, ϕ_s) $(dERT)$ for 23 out of 25 IOHprofiler problems in dimension $d = 100$, compared to the ERT of the best static GAs according to the results in Section 5.2 $(sERT)$. Each black dot represents one ERT value. The relative deviation is calculated by $(dERT - sERT)/sERT$ so that negative values (below the red line) correspond to an advantage of the dynamic combination over the best static algorithm. We only display values between -0.5 and 0.5 so that the results of F24-F25 are missing here with values larger than 1. All ERT values are based on 100 independent runs.

and we then run the combination 100 independent times on the problem that they have been selected for.

In Figure 7.3 we compare the so-obtained ERT values with the best ERT value reported in Section 5.2, which we refer to as the *best static algorithm* (BSA). For combinations (A_1, A_2, ϕ_s) for which the parent population sizes μ_1 of A_1 is larger than the parent population size μ_2 of A_2 we selected the best μ_2 points to initialize the parent population of A_2. Where $\mu_1 < \mu_2$, the new parent population comprises all μ_1 points, as additional $\lfloor \mu_2/2 \rfloor - \mu_1$ copies of the best points, and $\lceil \mu_2/2 \rceil$ randomly added individuals.

For some of the problems (e.g., F1, F2, F7, F11-14, F16-23)), the ERT of several combinations (A_1, A_2, ϕ_s) outperform that of the BSA. For other functions, and in particular for F10, F24, and F25, none of the combinations (A_1, A_2, ϕ_s) is able to outperform the BSA.

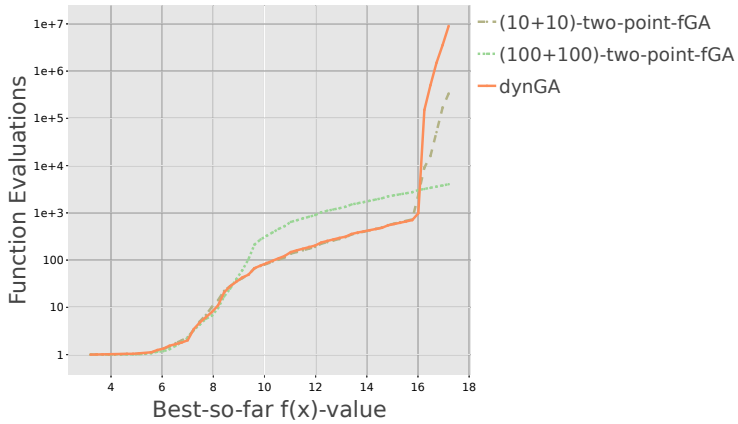


Figure 7.4: Fixed-target ERTs of GAs on F24 in dimension $d = 100$. The dynGA switches from the $(10 + 10)$-two-point-fGA to the $(100 + 100)$-two-point-fGA at the target $f(x) = 15.81$. Results are from 100 independent runs.

Local Optima are Deceptive An intuitive explanation for the cases where dynAS fails is that this is caused by local optima. Recall that in the computation of the predicted ERT, the contribution of A_1 to the predicted ERT is decided by its ERT hitting the target $f(x) = \phi_s$. However, using the ERT as the cost metric, we can not obtain information to estimate whether the algorithm is trapped or around a local optimum.

Figure 7.4 plots the fixed-target result of the best tested dynamic genetic algorithm (dynGA) on F24, which uses a $(10 + 10)$-two-point-fGA at first and the switches to a $(100 + 100)$-two-point-fGA. By using a small population size 10 initially, the dynGA converges to the switching point ($f(x) = 15.81$) fast, but it is trapped there and could not follow the original trend of the $(100+100)$ GA later. We do not solve this problem here, but it is interesting to spot this issue for future work.

7.4 Summary

We have investigated in this chapter possibilities to leverage existing benchmark data to derive switch-once dynamic algorithm selection policies. While for some cases the "theoretical" approach suggested in [153] could indeed predict combinations that outperformed the best static solver, the results are less positive for others. One obstacle that hinders an accurate performance prediction are local optima: when the first al-

gorithm is very good at converging to a local optimum, it is likely to be chosen as A_1. It is then important, however, to continue the search with an algorithm that has a good enough exploration power to escape the local optimum. This ability, however, seems hard to infer from the pure performance profiles, and may require a "human in the loop".

Going forward, our long-term goal is the automated detection of situations in which switching from one algorithm to another can be beneficial. To this end, we will further investigate efficient strategies to *warm-start* the algorithms by actively using the information accumulated thus far. In the here-presented study, we have used ERT values as performance measure and as indicator to select which algorithm combinations to execute. In future work we will consider other performance measures, and in particular those that measure the anytime performance of the algorithms.

Chapter 8

Conclusions

In this thesis, we have developed the IOHprofiler benchmarking software, and we demonstrated the benefits of our software in benchmarking evolutionary computation methods and studying (dynamic) algorithm configuration.

In practice, we discussed in Chapter 1 the role of benchmarking in optimization and explained the demand for a new benchmarking environment. Thus, we presented our IOHprofiler benchmarking software and answered **research question 1** in Chapter 3. To address **research question 2,** we performed benchmarking studies on the evolutionary and genetic algorithms for a wide range of pseudo-Boolean optimization problems. Precisely, we investigated the impact of the population size and the mutation rate for the $(1 + \lambda)$ EAs and the impact of the crossover probability for the $(\mu + \lambda)$ GA on OneMax and LeadingOnes in Chapter 4. Moreover, we compared twelve heuristics and variants of a family of genetic algorithms on the twenty-five problems provided by IOHprofiler in Chapter 5. The benchmarking studies inspired us to work on self-adaptation and algorithm configuration. Precisely, for self-adaptation, we proposed the standard normalized bit mutation for the $(1 + \lambda)$ EAs and answered **research question 3** in Chapter 4. In addition, for algorithm configuration, we applied Irace, MIP-EGO, and MIES to tune the parameters of the $(\mu + \lambda)$ GA and discussed the impact of the cost metric for the configuration task in Chapter 6, answering **research question 4**. **Research question 5** was discussed in Chapter 7, where we explored the possibilities of leveraging benchmarking data for dynamic algorithm selection.

Despite the achievements described in this thesis, the goal of developing a guideline on which algorithms to favor for which kinds of the problem remains a challenging en-

deavor. Much research is yet to be done to answer if there is still improvement space for the design of the algorithm or how we can improve the-state-of-art algorithms. Among the possible avenues for future research, we consider the following topics particularly important: the development of IOHprofiler, the bi-objective algorithm configuration problem, advancing algorithm configuration techniques, parameter control, and dynamic algorithm selection.

- **Development of IOHprofiler** IOHprofiler will be an ongoing project, and more components will be added to bring more ideas together. The software has gained attention from the community, and we are glad to see expectations from different researchers for the future of IOHprofiler. For example, IOHprofiler, being integrated with other frameworks, contributes to the study of large scale automated algorithm design [3]. From the current development base, 1) we will consider more benchmark problems, even towards mixed-integer optimization problem and multi-objective optimization; 2) we will develop customized logging systems to support users observing algorithms' behaviour from different views of points; For example, a logger calculating the AUC values of algorithms have been applied for our study in Chapter 6; 3) novel evaluation criteria and visualization of algorithms' performance will also be a potential topic. For example, more and more criteria have been integrated into IOHanalyzer, such as the Deep Statistical Comparison analysis [59] and Shapley-values.

- **Bi-objective algorithm configuration** Based on the study of tuning $(\mu + \lambda)$ GA for minimizing ERT and maximizing AUC, respectively, we plan to study in what sense our observation that tuning for AUC can help find better configurations for ERT generalizes to other algorithm families and/or problems. Moreover, given that we can use both running time and anytime performance during the tuning process, **a bi-objective** (or even multi-objective, if considering different performance measures) optimization process might balance the advantages of the different cost metrics.

- **Advancing algorithm configuration techniques** Our results have also demonstrated that none of the configuration methods clearly outperforms all others, suggesting to either combine them or to develop **guidelines that can help users find the most suitable configuration technique for their concrete problem at hand.** We also observe that none of the techniques could find configurations that outperform or perform on par with the $(1 + 1)$ EA in several cases, indicating improvement potential for these configuration methods.

- **Parameter Control and Dynamic Algorithm Selection** The fact that optimal values of algorithm parameters change along the optimization process can be observed in the results in this thesis. We have studied self-adaptation of the mutation rate for the $(1 + \lambda)$ EAs. Also, the result in Chapter 4 has shown that the $(\mu + \lambda)$ GA benefits from dynamic crossover probability. It would certainly be interesting to extend our study to using dynamic values for the relevant parameters μ, λ, p_c, and p on more optimization problems. Interestingly, the meta-algorithm (Algorithm 7 in Chapter 4) demonstrates that RLS and EAs can be seen as two configurations of the meta-algorithm. This inspired us with the topic of *online algorithm configuration.*, which is studied in the context of dynamic algorithm selection in Chapter 7. We have studied how benchmarking data can be used to infer informed (one-shot) dynamic algorithm selection schemes for the solution of pseudo-Boolean optimization problems [167]. However, we do not obtain improvement for all the tested problems. Therefore, we will investigate efficient strategies to warm-start the algorithms by actively using the information accumulated thus far. We have used ERT values as the performance measure and as the indicator to select which algorithm combinations to execute. In future work, we will consider other performance measures, particularly those that measure the anytime performance of the algorithms.

Bibliography

[1] Peyman Afshani, Manindra Agrawal, Benjamin Doerr, Carola Doerr, Kasper Green Larsen, and Kurt Mehlhorn. The query complexity of a permutation-based variant of mastermind. *Discrete Applied Mathematics*, 2019.

[2] Aldeida Aleti and Irene Moser. A systematic literature review of adaptive parameter control methods for evolutionary algorithms. *ACM Computing Surveys*, 49(3):1–35, 2016.

[3] Amine Aziz-Alaoui, Carola Doerr, and Johann Dreo. Towards large scale automated algorithm design by integrating modular benchmarking frameworks. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'21)*, pages 1365–1374. ACM, 2021.

[4] Thomas Bäck. Parallel optimization of evolutionary algorithms. In *Proc. of Parallel Problem Solving from Nature (PPSN'94)*, pages 418–427. Springer, 1994.

[5] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, USA, 1996.

[6] Thomas Bäck and Sami Khuri. An evolutionary heuristic for the maximum independent set problem. In *Proc. of Conference on Evolutionary Computation (CEC'94)*, pages 531–535. IEEE, 1994.

[7] Thomas Bäck and Martin Schütz. Intelligent mutation rate control in canonical genetic algorithms. In *Proc. of International Symposium on Foundations of Intelligent Systems (ISMIS'96)*, pages 158–167. Springer, 1996.

[8] Golnaz Badkobeh, Per Kristian Lehre, and Dirk Sudholt. Unbiased black-box complexity of parallel search. In *Proc. of Parallel Problem Solving from Nature (PPSN'14)*, pages 892–901. Springer, 2014.

[9] Barrie M Baker and MA Ayechew. A genetic algorithm for the vehicle routing problem. *Computers & Operations Research*, 30(5):787–800, 2003.

[10] Egon Balas and Paolo Toth. *Branch and bound methods for the traveling salesman problem*. Carnegie-Mellon University, 1983.

[11] Francisco Barahona. On the computational complexity of Ising spin glass models. *Journal of Physics A Mathematical General*, 15:3241–3253, 1982.

[12] Thomas Bartz-Beielstein, Carola Doerr, Jakob Bossek, Sowmya Chandrasekaran, Tome Eftimov, Andreas Fischbach, Pascal Kerschke, Manuel Lopez-Ibanez, Katherine M Malan, Jason H Moore, et al. Benchmarking in optimization: Best practice and open issues. *arXiv preprint arXiv:2007.03488*, 2020.

[13] Thomas Bartz-Beielstein, Christian WG Lasarczyk, and Mike Preuß. Sequential parameter optimization. In *Proc. of Congress on Evolutionary Computation (CEC'05)*, pages 773–780. IEEE, 2005.

[14] Maike Basmer and Timo Kehrer. Encoding adaptability of software engineering tools as algorithm configuration problem: A case study. In *Proc. of International Conference on Automated Software Engineering Workshop (ASEW'19)*, pages 86–89. IEEE, 2019.

[15] Jordan Bell and Brett Stevens. A survey of known results and research areas for n-queens. *Discrete Mathematics*, 309(1):1–31, 2009.

[16] Yoshua Bengio. Gradient-based optimization of hyperparameters. *Neural Computation*, 12(8):1889–1900, 2000.

[17] André Biedenkapp, H Furkan Bozkurt, Theresa Eimer, Frank Hutter, and Marius Lindauer. Dynamic algorithm configuration: foundation of a new meta-algorithmic framework. In *Proc. of European Conference on Artificial Intelligence (ECAI'20)*, pages 427–434. IOS Press, 2020.

[18] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. F-race and iterated f-race: An overview. *Experimental Methods for the Analysis of Optimization Algorithms*, pages 311–336, 2010.

[19] Endre Boros and Peter L Hammer. Pseudo-boolean optimization. *Discrete Applied Mathematics*, 123(1-3):155–225, 2002.

[20] Jakob Bossek, Pascal Kerschke, and Heike Trautmann. Anytime behavior of inexact tsp solvers and perspectives for automated algorithm selection. In *Proc. of Congress on Evolutionary Computation (CEC'20)*, pages 1–8. IEEE, 2020.

[21] Süntje Böttcher, Benjamin Doerr, and Frank Neumann. Optimal fixed and adaptive mutation rates for the LeadingOnes problem. In *Proc. of Parallel Problem Solving from Nature (PPSN'10)*, pages 1–10. Springer, 2010.

[22] Patrick Briest, Dimo Brockhoff, Bastian Degener, Matthias Englert, Christian Gunia, Oliver Heering, Thomas Jansen, Michael Leifhelm, Kai Plociennik, Heiko Röglin, et al. The Ising model: Simple evolutionary algorithms as adaptation schemes. In *Proc. of Parallel Problem Solving from Nature (PPSN'04)*, pages 31–40. Springer, 2004.

[23] Nathan Buskulic and Carola Doerr. Maximizing drift is not optimal for solving onemax. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'19)*, pages 425–426. ACM, 2019.

[24] Sébastien Cahon, Nordine Melab, and EG Talbi. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2004.

[25] Eduardo Carvalho Pinto and Carola Doerr. A simple proof for the usefulness of crossover in black-box optimization. In *Proc. of Parallel Problem Solving from Nature (PPSN'18)*, pages 29–41. Springer, 2018.

[26] Marco Cavazzuti. Design of experiments. In *Optimization Methods*, pages 13–42. Springer, 2013.

[27] Francisco Chicano, Andrew M. Sutton, L. Darrell Whitley, and Enrique Alba. Fitness probability distribution of bit-flip mutation. *Evolutionary Computation*, 23(2):217–248, 2015.

[28] David A Cohen, Martin C Cooper, Peter G Jeavons, and Andrei A Krokhin. The complexity of soft constraint satisfaction. *Artificial Intelligence*, 170(11):983–1016, 2006.

[29] Dogan Corus and Pietro Simone Oliveto. Standard steady state genetic algorithms can hillclimb faster than mutation-only evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 22(5):720–732, 2018.

[30] Dogan Corus and Pietro Simone Oliveto. On the benefits of populations for the exploitation speed of standard steady-state genetic algorithms. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'19)*, pages 1452–1460. ACM, 2019.

[31] Duc-Cuong Dang, Tobias Friedrich, Timo Kötzing, Martin S Krejca, Per Kristian Lehre, Pietro S Oliveto, Dirk Sudholt, and Andrew M Sutton. Escaping local optima using crossover with emergent diversity. *IEEE Transactions on Evolutionary Computation*, 22(3):484–497, 2017.

[32] Nguyen Dang and Carola Doerr. Hyper-parameter tuning for the $(1 + (\lambda, \lambda))$ GA. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'19)*, pages 889–897. ACM, 2019.

[33] Lawrence Davis. Job shop scheduling with genetic algorithms. In *Proc. of International Conference on Genetic Algorithms*, pages 136–140. Lawrence Erlbaum Associates, 1985.

[34] Kenneth Alan De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, Ann Arbor, MI, USA, 1975.

[35] Jacob de Nobel, Diederick Vermetten, Hao Wang, Carola Doerr, and Thomas Bäck. Tuning as a means of assessing the benefits of new ideas in interplay with existing algorithmic modules. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'21), Companion Material*, pages 1375–1384. ACM, 2021.

[36] Marcelo de Souza, Marcus Ritt, Manuel López-Ibáñez, and Leslie Pérez Cáceres. Acviz: A tool for the visual analysis of the configuration of algorithms with irace. *Operations Research Perspectives*, 8:100186, 2021.

[37] Benjamin Doerr. Analyzing randomized search heuristics: Tools from probability theory. In *Theory of Randomized Search Heuristics*, pages 1–20. World Scientific Publishing, 2011.

[38] Benjamin Doerr. Better runtime guarantees via stochastic domination. In *Proc. of Evolutionary Computation in Combinatorial Optimization (EvoCOP'18)*, pages 1–17. Springer, 2018.

[39] Benjamin Doerr. Analyzing randomized search heuristics via stochastic domination. *Theoretical Computer Science*, 773:115–137, 2019.

[40] Benjamin Doerr and Carola Doerr. A tight runtime analysis of the (1+(λ,λ)) genetic algorithm on OneMax. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'15)*, pages 1423–1430. ACM, 2015.

[41] Benjamin Doerr and Carola Doerr. Optimal static and self-adjusting parameter choices for the $(1 + (\lambda, \lambda))$ genetic algorithm. *Algorithmica*, 80:1658–1709, 2018.

[42] Benjamin Doerr and Carola Doerr. Theory of parameter control mechanisms for discrete black-box optimization: Provable performance gains through dynamic parameter choices. In Benjamin Doerr and Frank Neumann, editors, *Theory of Randomized Search Heuristics in Discrete Search Spaces*. Springer, 2019.

[43] Benjamin Doerr, Carola Doerr, and Franziska Ebel. From black-box complexity to designing new genetic algorithms. *Theoretical Computer Science*, 567:87–104, 2015.

[44] Benjamin Doerr, Carola Doerr, and Timo Kötzing. Unknown solution length problems with no asymptotically optimal run time. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'17)*, pages 1367–1374. ACM, 2017.

[45] Benjamin Doerr, Carola Doerr, and Jing Yang. Optimal parameter choices via precise black-box analysis. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'16)*, pages 1123–1130. ACM, 2016.

[46] Benjamin Doerr, Christian Gießen, Carsten Witt, and Jing Yang. The $(1 + \lambda)$ evolutionary algorithm with self-adjusting mutation rate. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'17)*, pages 1351–1358. ACM, 2017.

[47] Benjamin Doerr, Edda Happ, and Christian Klein. Crossover can provably be useful in evolutionary computation. *Theoretical Computer Science*, 425:17–33, 2012.

[48] Benjamin Doerr, Daniel Johannsen, Timo Kötzing, Frank Neumann, and Madeleine Theile. More effective crossover operators for the all-pairs shortest path problem. *Theoretical Computer Science*, 471:12–26, 2013.

[49] Benjamin Doerr, Daniel Johannsen, and Carola Winzen. Multiplicative drift analysis. *Algorithmica*, 64:673–697, 2012.

[50] Benjamin Doerr, Huu Phuoc Le, Régis Makhmara, and Ta Duy Nguyen. Fast genetic algorithms. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'17)*, pages 777–784. ACM, 2017.

[51] Benjamin Doerr and Frank Neumann. *Theory of Evolutionary Computation: Recent Developments in Discrete Optimization*. Springer, 2020.

[52] Benjamin Doerr and Carola Winzen. Black-box complexity: Breaking the $O(n \log n)$ barrier of LeadingOnes. In *Proc. of International Conference on Artificial Evolution (EA'11)*, pages 205–216. Springer, 2011.

[53] Carola Doerr. Dynamic parameter choices in evolutionary computation. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'18), Companion Material*, pages 800–830. ACM, 2018.

[54] Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck. IOHprofiler: A benchmarking and profiling tool for iterative optimization heuristics. *arXiv preprint arXiv:1810.05281*, 2018.

[55] Carola Doerr, Furong Ye, Naama Horesh, Hao Wang, Ofer M Shir, and Thomas Bäck. Benchmarking discrete optimization heuristics with iohprofiler. *Applied Soft Computing*, 88:106027, 2020.

[56] Carola Doerr, Furong Ye, Sander van Rijn, Hao Wang, and Thomas Bäck. Towards a theory-guided benchmarking suite for discrete black-box optimization heuristics: profiling $(1 + \lambda)$ EA variants on OneMax and LeadingOnes. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'18)*, pages 951–958. ACM, 2018.

[57] Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomised Algorithms*. Cambridge University Press, 2009.

[58] Juan J Durillo and Antonio J Nebro. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760–771, 2011.

[59] Tome Eftimov, Gašper Petelin, and Peter Korošec. DSCTool: A web-service-based framework for statistical comparison of stochastic optimization algorithms. *Applied Soft Computing*, 87:105977, 2020.

[60] Katharina Eggensperger, Marius Lindauer, and Frank Hutter. Pitfalls and best practices in algorithm configuration. *Journal of Artificial Intelligence Research*, 64:861–893, 2019.

[61] Agoston Endre Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3:124–141, 1999.

[62] Hafsteinn Einarsson, Johannes Lengler, Marcelo Matheus Gauy, Florian Meier, Asier Mujika, Angelika Steger, and Felix Weissenberger. The linear hidden subset problem for the (1 + 1) EA with scheduled and adaptive mutation rates. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'18)*, pages 1491–1498. ACM, 2018.

[63] Saber M Elsayed, Ruhul A Sarker, and Daryl L Essam. Multi-operator based evolutionary algorithms for solving constrained optimization problems. *Computers & Operations Research*, 38(12):1877–1896, 2011.

[64] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[65] Simon Fischer and Ingo Wegener. The one-dimensional Ising model: Mutation versus recombination. *Theoretical Computer Science*, 344(2-3):208–225, 2005.

[66] Merrill M Flood. The traveling-salesman problem. *Operations Research*, 4(1):61–75, 1956.

[67] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[68] Christian Gießen and Carsten Witt. The interplay of population size and mutation probability in the $(1 + \lambda)$ EA on OneMax. *Algorithmica*, 78(2):587–609, 2017.

[69] David Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, 1989.

[70] Jens Gottlieb, Elena Marchiori, and Claudio Rossi. Evolutionary algorithms for the satisfiability problem. *Evolutionary Computation*, 10(1):35–50, 2002.

[71] John J Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):122–128, 1986.

[72] Jun Gu, Paul W Purdom, John Franco, and Benjamin W Wah. Algorithms for the satisfiability (SAT) problem: A survey. Technical report, Cincinnati University, 1996.

[73] George T Hall, Pietro S Oliveto, and Dirk Sudholt. On the impact of the cutoff time on the performance of algorithm configurators. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'19)*, pages 907–915. ACM, 2019.

[74] George T Hall, Pietro S Oliveto, and Dirk Sudholt. Analysis of the performance of algorithm configurators for search heuristics with global mutation operators. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'20)*, pages 823–831. ACM, 2020.

[75] Julia Handl, Douglas B Kell, and Joshua Knowles. Multiobjective optimization in bioinformatics and computational biology. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(2):279–292, 2007.

[76] Nikolaus Hansen. A practical guide to experimentation. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'18), Companion Material*, pages 432–447. ACM, 2018.

[77] Nikolaus Hansen, Anne Auger, and Dimo Brockhoff. Data from the BBOB workshops. https://coco.gforge.inria.fr/doku.php?id=algorithms-bbob, 2020.

[78] Nikolaus Hansen, Anne Auger, Dimo Brockhoff, Dejan Tušar, and Tea Tušar. Coco: Performance assessment. *arXiv preprint arXiv:1605.03560*, 2016.

[79] Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tušar, and Dimo Brockhoff. Coco: A platform for comparing continuous optimizers in a black-box setting. *Optimization Methods and Software*, 36(1):114–144, 2021.

[80] Nikolaus Hansen, Steffen Finck, Raymond Ros, and Anne Auger. *Real-parameter black-box optimization benchmarking 2009: Noiseless functions definitions*. PhD thesis, INRIA, 2009.

[81] Holger H Hoos and Thomas Stützle. Satlib: An online resource for research on sat. *Sat*, 2000:283–292, 2000.

[82] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. of International Conference on Learning and Intelligent Optimization (LION'11)*, pages 507–523. Springer, 2011.

[83] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[84] Hsien-Kuei Hwang, Alois Panholzer, Nicolas Rolin, Tsung-Hsi Tsai, and Wei-Mei Chen. Probabilistic analysis of the $(1+1)$-evolutionary algorithm. *Evolutionary Computation*, 26(2):299–345, 2018.

[85] Hyun-Sook Yoon and Byung-Ro Moon. An empirical study on the synergy of multiple crossover operators. *IEEE Transactions on Evolutionary Computation*, 6(2):212–223, April 2002.

[86] Thomas Jansen. *Analyzing Evolutionary Algorithms—The Computer Science Perspective*. Springer, 2013.

[87] Thomas Jansen, Kenneth A. De Jong, and Ingo Wegener. On the choice of the offspring population size in evolutionary algorithms. *Evolutionary Computation*, 13:413–440, 2005.

[88] Thomas Jansen and Ingo Wegener. The analysis of evolutionary algorithms–a proof that crossover really can help. *Algorithmica*, 34:47–66, 2002.

[89] Thomas Jansen and Ingo Wegener. Real royal road functions—where crossover provably is essential. *Discrete Applied Mathematics*, 149(1-3):111–125, 2005.

[90] Thomas Jansen and Ingo Wegener. On the analysis of a dynamic evolutionary algorithm. *Journal of Discrete Algorithms*, 4:181–199, 2006.

[91] Thomas Jansen and Christine Zarges. Analysis of evolutionary algorithms: from computational complexity analysis to algorithm engineering. In *Proc. of Foundations of Genetic Algorithms (FOGA'11)*, pages 1–14. ACM, 2011.

[92] G. Karafotias, M. Hoogendoorn, and A.E. Eiben. Parameter control in evolutionary algorithms: Trends and challenges. *IEEE Transactions on Evolutionary Computation*, 19:167–187, 2015.

[93] Giorgos Karafotias, Mark Hoogendoorn, and Ágoston E Eiben. Parameter control in evolutionary algorithms: Trends and challenges. *IEEE Transactions on Evolutionary Computation*, 19(2):167–187, 2014.

[94] Stuart Kauffman and Simon Levin. Towards a general theory of adaptive walks on rugged landscapes. *Journal of Theoretical Biology*, 128:11–45, 1987.

[95] Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary Computation*, 27(1):3–45, 2019.

[96] Scott Kirkpatrick, C. D. Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[97] Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka: Automatic model selection and hyperparameter optimization in weka. In *Proc. of Automated Machine Learning*, pages 81–95. Springer, 2019.

[98] Timo Kötzing, Dirk Sudholt, and Madeleine Theile. How crossover helps in pseudo-boolean optimization. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'11)*, pages 989–996. ACM, 2011.

[99] Natalio Krasnogor and Jim Smith. A memetic algorithm with self-adaptive local search: TSP as a case study. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'00)*, pages 987–994. Morgan Kaufmann, 2000.

[100] Martin S Krejca and Carsten Witt. Theory of estimation-of-distribution algorithms. *Theory of Evolutionary Computation*, pages 405–442, 2020.

[101] William B Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer Science & Business Media, 2013.

[102] Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231–247, 1992.

[103] Pedro Larranaga. A review on estimation of distribution algorithms. In *Estimation of Distribution Algorithms*, pages 57–100. Springer, 2002.

[104] Jörg Lässig and Dirk Sudholt. Adaptive population models for offspring populations and parallel evolutionary algorithms. In *Proc. of Foundations of Genetic Algorithms (FOGA'11)*, pages 181–192. ACM, 2011.

[105] Per Kristian Lehre and Pietro S Oliveto. Runtime analysis of population-based evolutionary algorithms: introductory tutorial at gecco 2017. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'17), Companion Material*, pages 414–434, 2017.

[106] Per Kristian Lehre and Carsten Witt. Black-box search by unbiased variation. *Algorithmica*, 64:623–642, 2012.

[107] Per Kristian Lehre and Xin Yao. Crossover can be constructive when computing unique input–output sequences. *Soft Computing*, 15(9):1675–1687, 2011.

[108] Johannes Lengler and Nicholas Spooner. Fixed budget performance of the (1+1) EA on linear functions. In *Proc. of Foundations of Genetic Algorithms (FOGA'15)*, pages 52–61. ACM, 2015.

[109] Rui Li, Michael TM Emmerich, Jeroen Eggermont, Thomas Bäck, Martin Schütz, Jouke Dijkstra, and Johan HC Reiber. Mixed integer evolution strategies for parameter optimization. *Evolutionary Computation*, 21(1):29–64, 2013.

[110] Andrei Lissovoi, Pietro S. Oliveto, and John Alasdair Warwicker. On the runtime analysis of generalised selection hyper-heuristics for pseudo-Boolean optimisation. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'17)*, pages 849–856. ACM, 2017.

[111] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The Irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[112] Manuel López-Ibánez and Thomas Stützle. Automatically improving the anytime behaviour of optimisation algorithms. *European Journal of Operational Research*, 235(3):569–582, 2014.

[113] Andrew Lucas. Ising formulations of many np problems. *Frontiers in Physics*, 2:5, 2014.

[114] Fernando H Magnago and Ali Abur. Fault location using wavelets. *IEEE transactions on Power Delivery*, 13(4):1475–1480, 1998.

[115] Vincent Mellor. Numerical simulations of the ising model on the union jack lattice. *arXiv preprint arXiv:1101.5015*, 2011.

[116] Burkhard Militzer, Michele Zamparelli, and Dieter Beule. Evolutionary search for low autocorrelated binary sequences. *IEEE Transactions on Evolutionary Computation*, 2(1):34–39, 1998.

[117] Vladimir Mironovich and Maxim Buzdalov. Evaluation of heavy-tailed mutation operator on maximum flow test generation problem. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'17), Companion Material*, pages 1423–1426. ACM, 2017.

[118] Melanie Mitchell, John H. Holland, and Stephanie Forrest. When will a genetic algorithm outperform hill climbing? In *Proc. of Neural Information Processing Systems Conference (NIPS'93)*, volume 6, pages 51–58. Morgan Kaufmann, 1993.

[119] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions i. binary parameters. In *Proc. of Parallel Problem Solving from Nature (PPSN'96)*, pages 178–187. Springer, 1996.

[120] Heinz Mühlenbein. The equation for response to selection and its use for prediction. *Evolutionary Computation*, 5:303–346, 1997.

[121] Tadahiko Murata and Hisao Ishibuchi. Positive and negative combination effects of crossover and mutation operators in sequencing problems. In *Proc. of Congress on Evolutionary Computation (CEC'16)*, pages 170–175. IEEE, 1996.

[122] Samadhi Nallaperuma, Frank Neumann, and Dirk Sudholt. Expected fitness gains of randomized search heuristics for the traveling salesperson problem. *Evolutionary Computation*, 25(4):673–705, 2017.

[123] Frank Neumann, Pietro Simone Oliveto, Günter Rudolph, and Dirk Sudholt. On the effectiveness of crossover for migration in parallel evolutionary algorithms. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'11)*, pages 1587–1594. ACM, 2011.

[124] Frank Neumann and Carsten Witt. *Bioinspired Computation in Combinatorial Optimization – Algorithms and Their Computational Complexity*. Springer, 2010.

[125] Tom Packebusch and Stephan Mertens. Low autocorrelation binary sequences. *Journal of Physics A: Mathematical and Theoretical*, 49(16):165001, 2016.

[126] IA Pasha, PS Moharir, and N Sudarshan Rao. Bi-alphabetic pulse compression radar signal design. *Sadhana*, 25(5):481–488, 2000.

[127] Leslie Pérez Cáceres, Manuel López-Ibáñez, Holger Hoos, and Thomas Stützle. An experimental study of adaptive capping in irace. In *Proc. of International Conference on Learning and Intelligent Optimization (LION'17)*, pages 235–250. Springer, 2017.

[128] Eduardo Carvalho Pinto and Carola Doerr. Discussion of a more practice-aware runtime analysis for evolutionary algorithms. In *Proc. of Artificial Evolution (EA'17)*, pages 298–305. Springer, 2017.

[129] Yasha Pushak and Holger Hoos. Algorithm configuration landscapes. In *Prof. of Parallel Problem Solving from Nature (PPSN'18)*, pages 271–283. Springer, 2018.

[130] Masoud Rabbani, M Aramoon Bajestani, and G Baharian Khoshkhou. A multi-objective particle swarm optimization for project selection problem. *Expert Systems with Applications*, 37(1):315–321, 2010.

[131] Jeremy Rapin and Olivier Teytaud. Nevergrad - A gradient-free optimization platform. https://GitHub.com/FacebookResearch/Nevergrad, 2018.

[132] Gerhard Reinelt. Tsplib-a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[133] Anna Rodionova, Kirill Antonov, Arina Buzdalova, and Carola Doerr. Offspring population size matters when comparing evolutionary algorithms with self-adjusting mutation rates. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'19)*, pages 855–863. ACM, 2019.

[134] Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok, editors. *Handbook of Natural Computing: Theory, Experiments, and Applications*. Springer, 2012.

[135] Shokri Z Selim and K1 Alsultan. A simulated annealing algorithm for the clustering problem. *Pattern Recognition*, 24(10):1003–1008, 1991.

[136] Bart Selman, Hector J. Levesque, and David G. Mitchell. A new method for solving hard satisfiability problems. In *Proc. of National Conference on Artificial Intelligence (AAAI'92)*, pages 440–446. AAAI, 1992.

[137] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[138] Irwin I Shapiro, Gordon H Pettengill, Michael E Ash, Melvin L Stone, William B Smith, Richard P Ingalls, and Richard A Brockelman. Fourth test of general relativity: preliminary results. *Physical Review Letters*, 20(22):1265, 1968.

[139] Ofer M. Shir, Carola Doerr, and Thomas Bäck. Compiling a benchmarking test-suite for combinatorial black-box optimization: a position paper. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'18), Companion Material*, pages 1753–1760. ACM, 2018.

[140] William M Spears. Crossover or mutation? In *Proc. of Foundations of genetic algorithms (FOGA'93)*, pages 221–237. Elsevier, 1993.

[141] Dirk Sudholt. Crossover is provably essential for the Ising model on trees. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'05)*, pages 1161–1167. ACM, 2005.

[142] Dirk Sudholt. A new method for lower bounds on the running time of evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 17:418–435, 2013.

[143] Dirk Sudholt. How crossover speeds up building block assembly in genetic algorithms. *Evolutionary Computation*, 25(2):237–274, 2017.

[144] Dirk Sudholt. The benefits of population diversity in evolutionary algorithms: A survey of rigorous runtime analyses. In Benjamin Doerr and Frank Neumann, editors, *Theory of Evolutionary Computation: Recent Developments in Discrete Optimization*, pages 359–404. Springer, 2020.

[145] Ponnuthurai N Suganthan, Nikolaus Hansen, Jing J Liang, Kalyanmoy Deb, Ying-Ping Chen, Anne Auger, and Santosh Tiwari. Problem definitions and evaluation criteria for the cec 2005 special session on real-parameter optimization. *KanGAL report*, 2005005(2005):2005, 2005.

[146] Ryoji Tanabe. Analyzing adaptive parameter landscapes in parameter adaptation methods for differential evolution. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO 20')*, pages 645–653. ACM, 2020.

[147] Dirk Thierens. On benchmark properties for adaptive operator selection. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'09)*, pages 2217–2218. ACM, 2009.

[148] Tea Tusar, Dimo Brockhoff, and Nikolaus Hansen. Mixed-integer benchmark problems for single- and bi-objective optimization. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'19)*, pages 718–726. ACM, 2019.

[149] Tea Tušar, Dimo Brockhoff, Nikolaus Hansen, and Anne Auger. Coco: The bi-objective black box optimization benchmarking (bbob-biobj) test suite. *arXiv preprint arXiv:1604.00359*, 2016.

[150] Sander van Rijn, Hao Wang, Matthijs van Leeuwen, and Thomas Bäck. Evolving the structure of evolution strategies. In *Proc. of Symposium Series on Computational Intelligence (SSCI'16)*, pages 1–8. IEEE, 2016.

[151] Bas van Stein, Hao Wang, and Thomas Bäck. Automatic configuration of deep neural networks with parallel efficient global optimization. In *Proc. of International Joint Conference on Neural Networks (IJCNN'19)*, pages 1–7. IEEE, 2019.

[152] Swetha Varadarajan and Darrell Whitley. The massively parallel *mixing genetic algorithm* for the traveling salesman problem. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'19)*, pages 872–879. ACM, 2019.

[153] Diederick Vermetten, Hao Wang, Thomas Bäck, and Carola Doerr. Towards dynamic algorithm selection for numerical black-box optimization: investigating bbob as a use case. In *Proc. Genetic and Evolutionary Computation Conference (GECCO'20)*, pages 654–662. ACM, 2020.

[154] Hao Wang, Michael Emmerich, and Thomas Bäck. Cooling strategies for the moment-generating function in bayesian global optimization. In *Proc. Congress on Evolutionary Computation (CEC'18)*, pages 1–8. IEEE, 2018.

[155] Hao Wang, Bas van Stein, Michael Emmerich, and Thomas Bäck. A new acquisition function for bayesian optimization based on the moment-generating function. In *Proc. of International Conference on Systems, Man, and Cybernetics (SMC'17)*, pages 507–512. IEEE, 2017.

[156] Hao Wang, Diederick Vermetten, Furong Ye, Carola Doerr, and Thomas Bäck. IOHanalyzer: Detailed performance analyses for iterative optimization heuristic. *ACM Transactions on Evolutionary Learning and Optimization*, 2022.

[157] Richard A. Watson and Thomas Jansen. A building-block royal road where crossover is provably essential. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'07)*, pages 1452–1459. ACM, 2007.

[158] Thomas Weise. Optimization benchmarking, 2016. Available at http://optimizationbenchmarking.github.io/.

[159] Thomas Weise. The W-Model, a tunable black-box discrete optimization benchmarking (BB-DOB) problem, implemented for the BB-DOB@GECCO workshop, 2018. Data is available at https://github.com/thomasWeise/BBDOB_W_Model.

[160] Thomas Weise and Zijun Wu. Difficult features of combinatorial optimization problems and the tunable w-model benchmark problem for simulating them. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'18), Companion Material*, pages 1769–1776. ACM, 2018.

[161] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, 1994.

[162] Darrell Whitley, Swetha Varadarajan, Rachel Hirsch, and Anirban Mukhopadhyay. Exploration and exploitation without mutation: Solving the jump function in $\theta(n)$ time. In *Proc. of Parallel Problem Solving from Nature (PPSN'18)*, pages 55–66. Springer, 2018.

[163] Carsten Witt. Tight bounds on the optimization time of a randomized search heuristic on linear functions. *Combinatorics, Probability & Computing*, 22:294–318, 2013.

[164] Bing Xue, Mengjie Zhang, Will N Browne, and Xin Yao. A survey on evolutionary computation approaches to feature selection. *IEEE Transactions on Evolutionary Computation*, 20(4):606–626, 2015.

[165] Furong Ye, Carola Doerr, and Thomas Bäck. Interpolating local and global search by controlling the variance of standard bit mutation. In *Proc. of Congress on Evolutionary Computation (CEC'19)*, pages 2292–2299. IEEE, 2019.

[166] Furong Ye, Carola Doerr, and Thomas Bäck. Data Sets for the study "Leveraging Benchmarking Data for Informed One-Shot Dynamic Algorithm Selection". https://doi.org/10.5281/zenodo.4501275, February 2021.

[167] Furong Ye, Carola Doerr, and Thomas Bäck. Leveraging benchmarking data for informed one-shot dynamic algorithm selection. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'21), Companion Material*, pages 245–246. ACM, 2021.

[168] Furong Ye, Carola Doerr, Hao Wang, and Thomas Bäck. Automated configuration of genetic algorithms by tuning for anytime performance. *IEEE Transactions on Evolutionary Computation*, 2022.

[169] Furong Ye, Carola Doerr, Hao Wang, and Thomas Bäck. Data sets for the study "Automated Configuration of Genetic Algorithms by Tuning for Anytime Performance". https://doi.org/10.5281/zenodo.4823492, May 2021.

[170] Furong Ye, Hao Wang, Carola Doerr, and Thomas Bäck. Benchmarking a $(\mu+\lambda)$ genetic algorithm with configurable crossover probability. In *Proc. of Parallel Problem Solving from Nature (PPSN'20)*, pages 699–713. Springer, 2020.

[171] Furong Ye, Hao Wang, Carola Doerr, and Thomas Bäck. Experimental Data Sets for the study "Benchmarking a $(\mu+\lambda)$ Genetic Algorithm with Configurable Crossover Probability". https://doi.org/10.5281/zenodo.3753086, April 2020.

Acronyms

AC Algorithm Configuration

AS Algorithm Selection

AUC Area Under the empirical cumulative distribution function Curve

BDA Best Dynamic Algorithm selection policy

BSA Best Static Algorithm

COCO COmparing Continuous Optimizers

DoE Design of Experiment

dynAS dynamic Algorithm Selection

EC Evolutionary Computation

ECDF Empirical Cumulative Distribution Function

EDA Estimation of Distribution Algorittm

ERT Expected Running Time

ES Evolution Strategy

Acronyms

fGA fast Genetic Algorithm

GA Genetic Algorithm

gHC greedy Hill Climber

GP Genetic Programming

GS Grid Search

IOH Iterative Optimization Heuristic

MIES Mixed Integer Evolution Strategies

MIP-EGO Mixed-Integer Parallel Efficient Global Optimization

PBO Pseudo-Boolean Optimization

RLS Randomized Local Search

SMAC Sequential Model-based Algorithm Configuration

UMDA Univariate Marginal Distribution Algorithm

vGA vanilla Genetic Algorithm

Samenvatting

In zowel het dagelijks leven als in wetenschappelijk onderzoek worden we vaak geconfronteerd met moeilijke optimalisatie-problemen. Om deze problemen op te lossen zijn er veel verschillende algoritmes ontworpen, maar we hebben meestal geen duidelijke conclusies over hoe effectief deze algoritmes zijn voor specifieke type problemen. Gelukkig kunnen benchmark studies ons helpen om de effectiviteit van deze algoritmes op een onbevoordeelde manier te vergelijken.

In deze thesis introduceren we de IOHprofiler, een benchmark omgeving die de transitie van de implementatie van algoritmes naar het analyseren en visualiseren van hun effectiviteit mogelijk maakt. Deze software bestaat uit twee componenten: IOHexperimenter, een gebruiksvriendelijke en makkelijk aanpasbaare module voor het uitvoeren van de benchmark experimenten en het genereren van de bijbehorende data; en IOHanalyzer, een module voor het verwerken van deze data en het genereren van gedetaileerde statistiche analyses.

We maken gebruik van de verscheidene functionaliteiten van IOHprofiler om de effectiviteit van evolutionaire algoritmes voor het optimaliseren van discrete problemen te analyseren. We bestuderen specifiek de impact van de mutatie-graaden de grootte van de populatie in $(1+\lambda)$ EAs en de impact van de crossover kans binnen $(\mu+\lambda)$ EAs op OneMax en LeadingOnes. Verder vergelijken we twaalf heuristieken en verschillende versies van genetische algoritmes op vijfentwintig pseudo-Booleaanse problemen. Geinspireerd door deze resultaten introduceren we de gestandardiseerde genormaliseerde bit mutatie voor EAs en stellen we voor om niet-asymptotische looptijd-analyse (d.w.z., genzen die houden voor een specifieke dimensie van een probleem in plaats van grote-O-notatie) te gebruiken voor theoretische studies over het gedrag van EAs.

Vervolgens gebruiken we verschillende algoritme-configuratie methoden (irace, MIP-EGO en MIES) om de parameters van een genetisch algoritme af te stemmen. De experimentele resultaten geven ons inzicht in veelbelovende configuraties van het

genetische algoritme voor verschillende type problemen. Ook analyseren we de impact van de kost-metriek op het configuratie proces. Onze resultaten suggereren dat zelfs wanneer het doel is om de verwachte optimalisatie-tijd te minimaliseren het vaak wenselijk is om andere metrieken, die gebruik maken van de effectiviteit op meerdere punten in de optimalisatie (bijvoorbeeld oppervlakte onder de curve) te gebruiken tijdens het configureren.

Tot slot maken we gebruik van de verzamelde benchmark-data voor dynamische selectie van optimalisatie-algoritmes, waarvan de resultaten laten zien dat het wissellen van een configuration van het genetisch algoritme naar een andere configuratie tijdens het optimalisatie proces een verbeterde effectiviteit geeft vergeleken met de statische configuraties.

Summary

Many hard optimization problems need to be solved in our daily life and research work, and various algorithms are proposed to solve different problems. Meanwhile, we lack a clear conclusion about the performance of these algorithms across different types of problems. Fortunately, benchmarking studies can help us obtain unbiased assessments of algorithms' performance.

This thesis introduces the IOHprofiler benchmarking software, which allows for an easy transition from the implementation of algorithms to the analysis and comparison of performance data. The software consists of two components: IOHexperimenter, an easy-to-use and customizable module for processing the actual experiments and generating the performance data, and IOHanalyzer, a post-processing module for compiling detailed statistical evaluations.

Benefiting from the functionalities of IOHprofiler, we can systematically perform our study of benchmarking evolutionary algorithms on discrete optimization problems. In practice, we investigated the impact of the population size and the mutation rate for the $(1+\lambda)$ EAs and the impact of the crossover probability for the $(\mu+\lambda)$ GA on OneMax and LeadingOnes. Moreover, we compared twelve heuristics and variants of a family of genetic algorithms on the twenty-five pseudo-Boolean problems. Inspired by the benchmarking results, the standard normalized bit mutation is proposed for the EAs, and non-asymptotic runtime analysis (i.e., bounds that hold for a fixed dimension rather than in big-Oh notation) is suggested for theoretical study of understanding the behavior of EAs.

Moreover, we apply the algorithm configuration methods Irace, MIP-EGO, and MIES to tune the parameters of a family of genetic algorithms. The experimental results provide insights into promising configurations of the genetic algorithm for different types of problems. In addition, we analyze the impact of the cost metric for the configuration tasks. Our results suggest that even when interested in expected run-

ning time performance (i.e., ERT), it can be preferable to use the anytime performance measure (i.e., AUC) for the configuration task.

Finally, we leverage our benchmarking data for dynamic algorithm selection, of which results show improvements obtained by switching from a genetic algorithm configuration to another one during the optimization process when compared to the static configurations.

Acknowledgements

This thesis may never be finished without support from all my friends. I would like to thank everyone who has helped me in any way during the past four years.

Foremost, I would like to express gratitude to my promotor Thomas Bäck. Thank you for offering me the opportunity to join the NACO group. Even though I made all those mistakes, you always trusted me and encouraged me to be ambitious. When I needed your help, you always made time for me despite your busy schedule. I also would like to thank my supervisors, Carola Doerr and Hao Wang. Thank you, Carola. You have offered so much patience and kindness to me. Our weekly meetings guided me through the mist in the research forest many times. Your kind words encouraged me to move towards this achievement. Thank you, Hao, for all your guidance and help. You taught me many skills with professional advice and detailed examples. Your spirit inspired me to grow up as an independent researcher. I am sincerely honoured to have been a student of all of you.

It was my pleasure to work in NACO and LIACS. I got much professional help and enjoyed the international working environment in the group. I would like to say thank you to everyone in NACO and LIACS. Special thanks to Diederick and Jacob for the collaboration of IOHprofiler and for becoming close friends.

I was lucky to have collaboration and talks with many excellent researchers. I would like to thank: Ofer M. Shir and Naama Horesh for the collaboration on the ASOC paper; Thomas Weise for the discussions on W-model and for pointing out a mistake in our definition of Ising-models; Johann Dréo for the contributions to IOHprofiler.

As an international student, I am blessed to have my friends. Friendship helped me get through all tough periods, especially the days in the hospital. Particularly, I would like to thank my dearest friend, Yueqi. We have supported each other for twelve years. I felt so happy when you decided to come and stay in the Netherlands. I also would like to say thanks to my friends: Danyi, Jiao, Yuchao, and Kelvin, all for the gathering and

food; Koen, for asking me to the foobar when I just arrived Leiden, having coffee after working hours, and offering me the template for this thesis; Marie, for all drinks and honest conversations; Theodoros, for your concern and bringing happiness; Hugo, for international cultural discussions and inviting me to your parties; Charles, for cooking and offering help anytime; Sander and Lieuwe, for our work and study meetings; António and Alexa, for inviting me to your apartment; Kaifeng, Yash, and Fay, for being TAs together; Hui Wang and Wei, for sharing research and career plans; Hui Feng and Cong, for working in the same office for the first year; Anne, for hosting the PhD Seminar together; Marios, for the research discussions; Mano, for the guitar moments; Shengxiang, for sharing your place when I could not find accommodation; Yali and Yuchen, for tennis games; Zhongwei, for medical advice.

I also thank the Chinese Scholarship Council for supporting my PhD study financially.

At last, I must say thanks to my parents. I am so blessed to have such wonderful parents. Though the only thing you want from me is to be happy, I would like to present this thesis as a gift to you. Hope this achievement makes you proud.

Curriculum Vitae

Furong Ye was born on the 22nd July 1993 in Liupanshui, China. He had lived in this southwestern mountain city until he graduated from high school in 2010. After that, he moved to the southeastern ocean city Xiamen in China to study at Xiamen University. There, he obtained two Bachelor's degrees in Computer Science and Mathematical Statistics (in Economics) in 2014 and a Master's degree in Computer Engineering in 2017. His master thesis, titled "Research on the Vehicle Routing Problem with Three-dimensional Loading and Time Window Constraints," was supervised by Defu Zhang. During his study at Xiamen University, he was awarded the National Scholarship of China, the Xiamen University President Scholarship, and the Social Practice Scholarship of Fujian Province.

In September 2017, he started his PhD in Computer Science and joined the Natural Computing Group at Leiden University, supported by the Chinese Scholarship Council (CSC. No. 201706310143). His PhD study is under the supervision of Thomas Bäck, Carola Doerr (Sorbonne Université, France), and Hao Wang.