Universiteit
Leiden
The Netherlands

IOHanalyzer: detailed performance analyses for iterative optimization heuristics

Wang, H.; Vermetten, D.; Ye, F.; Doerr, C.; Bäck, T.H.W.

Citation
Wang, H., Vermetten, D., Ye, F., Doerr, C., & Bäck, T. H. W. (2022). IOHanalyzer: detailed performance analyses for iterative optimization heuristics. *Acm Transactions On Evolutionary Learning And Optimization*, *2*(1). doi:10.1145/3510426

Version:	Publisher's Version
License:	Leiden University Non-exclusive license
Downloaded from:	https://hdl.handle.net/1887/3513858

Note: To cite this publication please use the final published version (if applicable).

IOHanalyzer: Detailed Performance Analyses for Iterative Optimization Heuristics

HAO WANG, DIEDERICK VERMETTEN, and FURONG YE, LIACS, Leiden University, The Netherlands
CAROLA DOERR, Sorbonne Université, CNRS, LIP6, France
THOMAS BÄCK, LIACS, Leiden University, The Netherlands

Benchmarking and performance analysis play an important role in understanding the behaviour of **iterative optimization heuristics (IOHs)** such as local search algorithms, genetic and evolutionary algorithms, Bayesian optimization algorithms, etc. This task, however, involves manual setup, execution, and analysis of the experiment on an individual basis, which is laborious and can be mitigated by a generic and well-designed platform. For this purpose, we propose IOHanalyzer, a new user-friendly tool for the analysis, comparison, and visualization of performance data of IOHs.

Implemented in R and C++, IOHanalyzer is fully open source. It is available on CRAN and GitHub. IOHanalyzer provides detailed statistics about fixed-target running times and about fixed-budget performance of the benchmarked algorithms with a real-valued codomain, single-objective optimization tasks. Performance aggregation over several benchmark problems is possible, for example in the form of empirical cumulative distribution functions. 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.

IOHanalyzer can directly process performance data from the main benchmarking platforms, including the COCO platform, Nevergrad, the SOS platform, and IOHexperimenter. An R programming interface is provided for users preferring to have a finer control over the implemented functionalities.

CCS Concepts: • **Theory of computation → Random search heuristics**; **Design and analysis of algorithms**; *Bio-inspired optimization;*

Additional Key Words and Phrases: Evolutionary computation, black-box optimization, performance analysis

Our work has been financially supported by the Paris Ile-de-France region and by a public grant as part of the Investissement d'avenir project, reference ANR-11-LABX-0056-LMH, LabEx LMH, in a joint call with the Gaspard Monge Program for optimization, operations research, and their interactions with data sciences. Furong Ye acknowledges financial support from the China Scholarship Council, CSC No. 201706310143. Parts of our work have been inspired by working group 3 of COST Action CA15140 'Improving Applicability of Nature-Inspired Optimisation by Joining Theory and Practice (ImAppNIO)' supported by the European Cooperation in Science and Technology.

Authors' addresses: H. Wang, D. Vermetten, F. Ye, and T. Bäck, LIACS, Leiden University, Niels Bohrweg 1, Leiden, The Netherlands; emails: {h.wang, d.l.vermetten, f.ye, t.h.w.baeck}@liacs.leidenuniv.nl; C. Doerr, Sorbonne Université, CNRS, LIP6, 4 place Jussieu, Paris, France, Carola; email: Doerr@lip6.fr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.

2688-3007/2022/04-ART3 $15.00
https://doi.org/10.1145/3510426

ACM Reference format:
Hao Wang, Diederick Vermetten, Furong Ye, Carola Doerr, and Thomas Bäck. 2022. IOHanalyzer: Detailed Performance Analyses for Iterative Optimization Heuristics. *ACM Trans. Evol. Learn. Optim* 2, 1, Article 3 (April 2022), 29 pages.
https://doi.org/10.1145/3510426

1 INTRODUCTION

Optimization problems not admitting exact solution approaches affect almost all aspects of our daily lives. They appear, for example, in product design, scheduling, data analysis, and machine learning (e.g., hyper-parameter tuning). For instance, it is sometimes important to analyze the optimization procedure when training a neural network, which helps us understand the learning process. The intractability of these problems can have various reasons, e.g., a lack of problem-specific knowledge, limited access to problem data, or the inherent complexity of the underlying problem. **Iterative optimization heuristics (IOHs)** are algorithms designed to search for high-quality solutions of such problems. IOHs are characterized by a sequential structure, which aims to evolve good solutions by iteratively sampling the decision space. The distribution from which the solution candidates are sampled is adjusted after each iteration, to reflect the new information obtained from the last evaluations.

IOHs are often randomized, both with respect to candidate generation and with respect to selecting the information stored from one iteration to the next. The optimization behavior of IOHs is therefore a highly complex system with many dependencies. This makes it very difficult to predict how well a particular IOH performs on a given problem. Existing theoretical results are limited to rather simple algorithms and/or problems, which are typically not representative for the complex strategies used in practice (see [1, 11, 50] for recent surveys of theoretical results). To gather a good understanding of the performance and the search behavior of realistic IOHs and applications, we are therefore often restricted to an empirical evaluation of these solvers, from which we may extrapolate accurate performance predictions. Supporting such empirical evaluations through a systematic experimental design is one of the primary goals of *algorithm benchmarking*. Algorithm benchmarking addresses the selection of problem instances that are most suitable for an accurate performance extrapolation, the experimental setup of the data generation, the choice of the performance indicators and their visualizations, the choice of the statistics used to compare two or more algorithms, etc. In practice, those various aspects of algorithm benchmarking make it laborious and demanding for researchers to handle the details of experimentation, which calls for a standard and easy-to-use software implementation of algorithm benchmarking that would drastically reduce the manual work for practitioners.

1.1 IOHanalyzer: Overview and Availability

In this work, we present IOHanalyzer, a versatile, user-friendly, and highly interactive platform for the assessment, comparison, and visualization of IOH performance data. IOHanalyzer is designed to assess the empirical performance of sampling-based optimization heuristics in an algorithm-agnostic manner. Our **key design principles** are (1) an easy-to-use software interface, (2) interactive performance analysis, and (3) convenient export of reports and illustrations.

IOHanalyzer is developed as the data analysis component of IOHprofiler, a benchmarking platform that aims to integrate various elements of the entire benchmarking pipeline, ranging from problem (instance) generators and modular algorithm frameworks over automated algorithm configuration techniques and feature extraction methods to the actual experimentation, data analysis, and visualization [12]. An illustration of the interplay between these different components is provided in Figure 1. Notably, IOHprofiler already provides the following components:

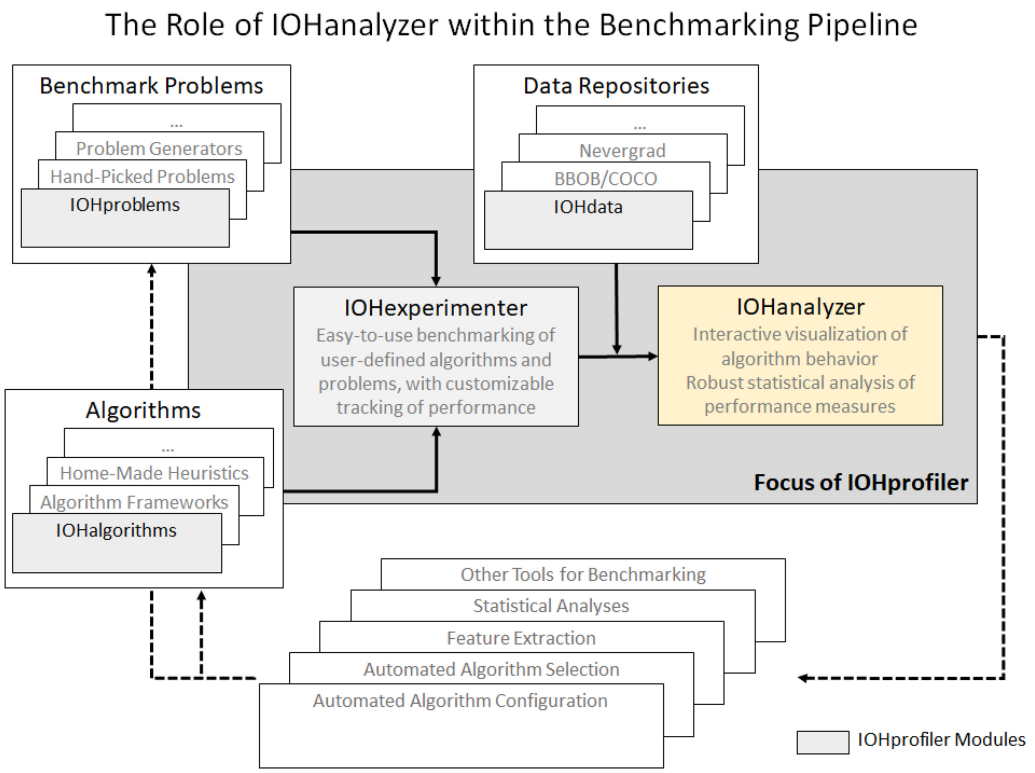


Fig. 1. IOHanalyzer is a module of the IOHprofiler benchmarking environment, which targets different steps of the benchmarking pipeline.

- **IOHproblems:** a collection of benchmark problems. This component currently comprises (1) the PBO suite of pseudo-Boolean optimization problems suggested in [13], (2) the 24 numerical, noise-free BBOB functions from the **COmparing Continuous Optimizers (COCO)** platform [30], and (3) the Wmodel problem generator proposed in [65].
- **IOHalgorithms:** a collection of IOHs. For the moment, the algorithms used for the benchmark studies presented in [2, 9, 13] are available. This subsumes textbook algorithms for pseudo-Boolean optimization, an integration to the object-oriented algorithm design framework ParadisEO [40], and the modular algorithm framework for CMA-ES variants originally suggested in [59] and extended in [9]. 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 [13], a sample data set used in this paper, and some selected data sets from the COCO repository [26]. **IOHdata** also contains performance data from Facebook's Nevergrad benchmarking environment [52], 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 work.

IOHanalyzer takes as input benchmarking data sets, generated, e.g., by IOHexperimenter, through the COCO platform, or through the Nevergrad environment. Of course, users can also use their own experimentation platform (the formatting requirements for the input files are described in Appendix A). IOHanalyzer provides an evaluation platform for these performance traces, which allows users to choose the performance measures, the ranges, and the precision of the displayed data according to their needs. In particular, IOHanalyzer supports both a fixed-target and a fixed-budget perspective, and allows various ways of aggregating performances across different problems (or problem instances). In addition to these performance-oriented analyses, IOHanalyzer also offers statistics about the evolution of non-static algorithmic components, such as, for example, the hyperparameters suggested by a self-adjusting parameter control scheme. These features will be described in more detail in Section 3, where the reader can also find illustrated examples.

The R programming interface of IOHanalyzer offers a fine control on the data and functionalities implemented therein. IOHanalyzer is written in R and C++ and makes use of the two R packages plotly [55] and shiny [8]. The version of the software described in this paper is *v0.1.6.1*, which has been made available on zenodo [62]. This repository also contains all datasets used for the examples illustrated in this paper. For users less experienced with programming in R we offer a web-based **graphical user interface (GUI)**, to which users can load their own data or use data from the **IOHdata** repository.

The stable release of the IOHanalyzer package is distributed through CRAN (https://CRAN.R-project.org/package=IOHanalyzer). It can be easily installed in an R console:

```
R> install.packages("IOHanalyzer")
```

The latest version is hosted on GitHub (https://github.com/IOHprofiler/IOHanalyzer, part of the IOHprofiler project), which can be installed using the devtools library as follows:[1]

```
R> devtools::install_github("IOHprofiler/IOHanalyzer")
```

An up-to-date documentation is maintained on the wiki page, available at https://iohprofiler.github.io/. The web-based GUI of IOHanalyzer is hosted at http://iohprofiler.liacs.nl.

The first use case of IOHanalyzer was the comparison of different variants of the $(1 + \lambda)$ **evolutionary algorithm (EA)** [14]. A number of improvements were made subsequently, and the first study of an important number experiments was reported in [13]. In the meantime, IOHanalyzer has been used in a number of studies, including [6, 35, 67]. It is under constant development. Some of the major ongoing extensions will be discussed in Section 4.

1.2 Related Benchmarking Environments

As argued above, benchmarking IOHs is an essential task towards a better understanding of IOHs. It is therefore not surprising that a large number of different tools have been developed for this purpose. For reasons of space, we can only summarize a few of them and concentrate on those which come closest to IOHanalyzer in terms of functionality and scope.

In evolutionary computation, the arguably best established benchmarking environment is the already mentioned COCO platform [30]. Originally designed to compare derivative-free optimization algorithms operating on numeric optimization problems [29], the tool has seen several extensions in the last years, e.g., towards multi-objective optimization [58], mixed-integer optimization [57], and large-scale optimization [60]. COCO consists of an experimentation part that produces data files with detailed performance traces, and an automated data analysis part in which

[1]The GitHub-page gets updated more frequently with minor changes, while the CRAN-version is generally updated only when major modifications are made.

a fixed number of standardized analyses are automatically generated. The by far most reported performance measures from the COCO framework are ***empirical cumulative distribution function* (ECDF)** curves, see Section 2 for definitions. The COCO software has a strong focus on *fixed-target performances* [27], i.e., on the time needed to find a solution of a certain quality.

COCO has been a major source of inspiration for the development of IOHprofiler. What concerns the performance assessment, the key difference between COCO and our IOHanalyzer is in the interactive interface that allows users of IOHanalyzer to study different performance measures, to change their ranges, and granularity. As mentioned, COCO performance files can be conveniently analyzed by IOHanalyzer.

Another important software environment for benchmarking sampling-based optimization heuristics is the Nevergrad framework [52]. As with COCO, Nevergrad implements functionalities for both experimentation and performance analysis, accommodating continuous, discrete, and mixed-integer problems. It has a strong focus on noisy optimization, but also comprises several noise-free optimization problems. In addition to studying IOHs, Nevergrad has a special suite to compare one-shot optimization techniques, i.e., non-iterative solvers. The current focus of Nevergrad is to be seen on the problem side, as it offers several new benchmark problems, such as the structured optimization problems which are aggregated in their own test suite. Nevergrad also provides interfaces to the following benchmark collections: LSGO [45], YABBOB [46], Pyomo [32], MLDA [24], and MuJoCo [56]. The performance evaluation, however, is much more basic than those of COCO or IOHanalyzer, in that only the quality of the finally recommended point(s) is stored, but no information about the search trajectory. That is, apart from taking a *fixed-budget perspective*, Nevergrad does not store performance traces, but only the final output. IOHanalyzer can interpret and visualize the csv files produced by Nevergrad. An extension of Nevergrad to allow for the same tracking features as IOHanalyzer is currently under construction, in a joint collaborative effort between the Nevergrad and the IOHanalyzer development teams.

Focusing on the algorithm design task, HeuristicLab [63] provides a relatively large collection of various IOHs (e.g., population-based search algorithms) as well as machine learning algorithms (e.g., Support Vector Regression), which are represented as graphs of operators. In HeuristicLab, new algorithms can be constructed by combining existing operators in a graphical user interface, avoiding the laborious coding details. While IOHprofiler mainly targets the black-box optimization problem, HeuristicLab incorporates a very diversified set of benchmark problems, ranging from the symbolic regression to data analysis problems. It implements the parallel execution of algorithms for the ease of benchmarking. In contrast to IOHanalyzer, which is available via a web interface and contains many detailed performance analyses and interactive plots, HeuristicLab is distributed via platform-dependent applications and includes some basic static plots (e.g., box plots) for assessing the empirical performance from a *fixed-budget perspective*.

Several other tools have been developed for displaying performance data and/or the search behavior in decision space. However, all tools that we are aware of allow much less flexibility with respect to the performance measures, the ranges, and the granularity of the analysis or focus on selected aspects of performance analysis only (e.g., [5, 17] study statistical significance, whereas [20, 53] aim to visualize performance with respect to multiple objectives). The ability of IOHanalyzer to link the evolution of algorithms' parameters to the evolution of solutions' quality seems to be unique.

2 BACKGROUND

This section provides the background and motivation for developing IOHanalyzer. In particular, we discuss black-box problems and their optimization and we recall the most relevant performance indicators that will be used in subsequent sections.

ALGORITHM 1: Blueprint of an iterative optimization heuristic (IOH) optimizing a function $f : \mathcal{S} \to \mathbb{R}$.

procedure IOH
  $t \leftarrow 0$ ▷ iteration counter
  $\mathcal{H}(0) \leftarrow \emptyset$ ▷ search history information
  choose a distribution $\Lambda(0)$ on $\mathbb{N}$ ▷ distribution of the number of samples
  while termination criterion not met **do**
    $t \leftarrow t + 1$
    sample $\lambda(t) \sim \Lambda(t-1)$ ▷ nbr. of points to be evaluated
    Based on $\mathcal{H}(t-1)$ choose a distribution $D(t)$ on $\mathcal{S}^{\lambda(t)}$ ▷ choice of sampling distribution
    sample $\left(x^{(t,1)}, \ldots, x^{(t,\lambda(t))}\right) \sim D(t)$ ▷ candidate generation
    evaluate $f\left(x^{(t,1)}\right), \ldots, f\left(x^{(t,\lambda(t))}\right)$ ▷ function evaluation
    choose $\mathcal{H}(t)$ and $\Lambda(t)$ ▷ information update
  end while
end procedure

2.1 Iterative Optimization Heuristics

We study the optimization of problems of the type $f : \mathcal{S} \to \mathbb{R}$, i.e., we assume our problem to be a single-objective, real-valued objective function (i.e., problems for which the quality of possible solutions is rated by real numbers), defined over a search space $\mathcal{S}$. We do not make any assumption on the set $\mathcal{S}$; it can be discrete or continuous, constrained or unconstrained. We do not require that f is explicitly modeled, i.e., f can very well be a *black-box optimization problem*, i.e., a problem for which we are able to evaluate the quality of points $x \in \mathcal{S}$ (e.g., through computer simulations or through physical experiments) but for which we do not know the mapping $x \mapsto f(x)$ without performing such evaluations. Intermediate *gray-box* settings are also possible, i.e., problems for which *some* information about the mapping $x \mapsto f(x)$ is available (see, for example, the discussion in [66], where a setting is analyzed in which users have information about the interaction between different variables). To ease notation, we nevertheless speak of black-box optimization in such cases, i.e., even when some *a priori* information about the problem f is available. We emphasize that the sampling-based optimization algorithms studied in our work can be competitive even when the problem f is explicitly known. The **low auto-correlation binary sequence (LABS)** problem is a good example for such a problem that can be defined in two lines, but for which the best known solvers are sampling-based [51]. The only important feature of the performance traces that can be analyzed by IOHanalyzer is that they rely on the evaluation of possible solution candidates – regardless of how these have been created.

For convenience of presentation, we consider in this document **maximization** as objective. Note, though, that IOHanalyzer automatically detects whether minimization of maximization is considered, and adjusts the plots and statistics accordingly. For example, the COCO and Nevergrad data sets typically consider minimization, whereas the PBO suite of **IOHproblems** studies maximization.

The class of algorithms that we are interested in are ***Iterative Optimization Heuristics* (IOHs)**. IOHs are entirely sampling-based, i.e., they sample the search space $\mathcal{S}$ and use the function values $f(x)$ of the evaluated samples x to guide the search. Algorithm 1 provides a blueprint for IOHs. Classical examples for IOHs are deterministic and stochastic local search algorithms (this class includes Simulated Annealing [42] and Threshold Accepting [16] as two

prominent examples), genetic and evolutionary algorithms [19], Bayesian Optimization and related global optimization algorithms [38], Estimation of Distribution algorithms [43], and Ant Colony Optimization algorithms [15].

2.2 Selected Performance Indicators

Unlike in optimization scenarios in which problem data is accessible without function querying and where solutions are hence typically generated constructively (as opposed to the sampling-based approach taken by IOHs), the most commonly studied performance measures in black-box optimization are based on the number of function evaluations. That is, instead of counting arithmetic operations or CPU time, we measure performance by counting the number of function evaluations that are performed to reach a certain quality threshold (*fixed-target setting*) or we measure the quality of the best found solution that could be recommended after a certain budget of function evaluations has expired (*fixed-budget setting*). Measuring the performance in the number of function evaluations is a classic assumption made in the black-box optimization literature [27]. In contrast to CPU time, this measure is machine-independent and not (or at least much less) sensitive with respect to the actual implementation.

As discussed above, many state-of-the-art IOHs are randomized in nature, therefore yielding random performance traces even when the underlying problem f is deterministic. The performance space is spanned by the number of evaluations, by the quality of the assessed solutions, and by the probability that the algorithm has found within a given budget of function evaluations a solution that is at least as good as a given quality threshold.

Basic Notation. To define the performance measures covered by IOHanalyzer we use the following notation.

- $\mathcal{F}$ denotes the set of problems under consideration. Each problem (or problem instance, depending on the context) $f \in \mathcal{F}$ is assumed to be a function $f\colon \mathcal{S} \to \mathbb{R}$. The *dimension* of $\mathcal{S}$ is denoted by d. We often consider scalable functions that are defined for several or all dimensions $d \in \mathbb{N}$. In such cases, we make the dimension explicit.
- $\mathcal{A} = \{A_1, A_2, \ldots\}$ is the set of algorithms under consideration. $\mathcal{A}$ can be finite or infinite. Often, $\mathcal{A}$ is a configurable meta-algorithmic framework, which allows users to specify parameters such as the degree of parallelism, the intensity of the local perturbations, the memory size, the use (or not) of recombination operators, etc.
- We denote by r the number of independent runs of an algorithm $A \in \mathcal{A}$ on problem $f \in \mathcal{F}$ in dimension d.
- $T(A, f, d, B, v, i) \in \mathbb{N} \cup \{\infty\}$ is a *fixed-target measure.* It denotes the number of function evaluations that algorithm A performed, in its i-th run and when maximizing the d-dimensional variant of problem f, to find a solution x satisfying $f(x) \geq v$. When A did not succeed in finding such a solution within the maximal allocated budget B, $T(A, f, d, B, v, i)$ is set to ∞. Several ways to deal with such failures are considered in the literature, as we shall discuss in the next paragraphs.
- Similarly to the above, $V(A, f, d, t, i) \in \mathbb{R}$ is a *fixed-budget measure.* It denotes the function value of the best solution that algorithm A evaluated within the first t evaluations of its i-th run, when maximizing the d-dimensional variant of problem f.

Descriptive Statistics. We next recall some basic descriptive statistics.

- The average function value given a budget value t is simply

$$\bar{V}(t) = \bar{V}(A, f, d, t) = \frac{1}{r} \sum_{i=1}^{r} V(A, f, d, t, i).$$

As we do with all other measures, we omit explicit mention of A, f, and d when they are clear from the context.

- The ***Penalized Average Runtime*** (**PAR**-c score, where $c \geq 1$ is the penalty factor) for a given target value v is defined as

$$\text{PAR-c}(v) = \text{PAR-c}(A, f, d, B, v) = \frac{1}{r} \sum_{i=1}^{r} \min \{T(A, f, d, B, v, i), cB\}, \tag{1}$$

  i.e., the PAR-c score is identical to the sample mean when all runs successfully identified a solution of quality at least v within the given budget B, whereas non-successful runs are counted as cB. In IOHanalyzer, we typically study the PAR-1 score, which, in abuse of notation, we also refer to as the mean.
- Apart from mean values, we are often interested in quantiles, and in particular in the *sample median* of the r values $\{T(A, f, d, B, v, i)\}_{i=1}^{r}$ and $\{V(A, f, d, t, i)\}_{i=1}^{r}$, respectively. By default, IOHanalyzer calculates the 2%, 5%, 10%, 25%, 50%, 75%, 90%, 95%, and 98% percentiles (denoted as $Q_{2\%}, Q_{5\%}, \ldots, Q_{98\%}$) for both running times and function values.
- We also study the *sample standard deviation* of the running times and function values, respectively.
- The *empirical success rate* is the fraction of runs in which algorithm A reached the given target v within the maximal number B of allowed function evaluations. That is, in the case of a maximization problem,

$$\widehat{p}_s = \widehat{p}_s(A, f, d, B, v) = \frac{1}{r} \sum_{i=1}^{r} \mathbb{1}(V(A, f, d, B, i) > v) = \frac{1}{r} \sum_{i=1}^{r} \mathbb{1}(T(A, f, d, B, v, i) < \infty), \tag{2}$$

  where $\mathbb{1}(\mathcal{E})$ is the characteristic function of the event $\mathcal{E}$.

Expected Running Time. An alternative to the PAR-c score is the **expected running time (ERT)**. ERT assumes independent restarts of the algorithm whenever it did not succeed in finding a solution of quality at least v within the allocated budget B. Practically, this corresponds to sampling indices $i \in \{1, \ldots, r\}$ (i.i.d. uniform sampling with replacement) until hitting an index i with a corresponding value $T(A, f, d, B, v, i) < \infty$. The running time would then have been $mB + T(A, f, d, B, v, i)$, where m is the number of sampled indices of unsuccessful runs. The average running time of such a hypothetically restarted algorithm is then estimated as

$$\begin{aligned} \text{ERT}(A, f, d, B, v) &= \frac{\sum_{i=1}^{r} \min \{T(A, f, d, B, v, i), B\}}{r\widehat{p}_s} \\ &= \frac{\sum_{i=1}^{r} \min \{T(A, f, d, B, v, i), B\}}{\sum_{i=1}^{r} \mathbb{1}(T(A, f, d, B, v, i) < \infty)}. \end{aligned} \tag{3}$$

Note that ERT can take an infinite value when none of the runs was successful in identifying a solution of quality at least v.

Cumulative Distribution Functions. For the fixed-target and fixed-budget analysis, IOHanalyzer estimates probability density (mass) functions and computes **empirical cumulative distribution functions (ECDFs)**. For the fixed-budget function value, its probability density function is estimated via the well-known **Kernel Density Estimation (KDE)** method, which approximates the density function by a superposition of kernel functions (e.g., Gaussian functions with a fixed width) centered at each data point [33]. Intuitively, a set of crowded data points would lead to a very peaky empirical density due to massive superpositions of the kernel, while a set of distant points

can only generate a relatively flat curve. For the fixed-target running time (an integer-valued random variable), we estimate its probability mass function by treating it as a real value and applying the KDE method. For a set $\{T(A, f, d, v, i)\}_{i=1}^{r}$ of fixed-target running times, its ECDF is defined as the fraction of runs which successfully found a solution of quality at least v within a budget of at most t function evaluations. That is,

$$\mathrm{ECDF}(A, f, d, v, t) = \frac{1}{r} \sum_{i=1}^{r} \mathbb{1}(T(A, f, d, v, i) \leq t).$$

ECDF values are most typically used in aggregated form. IOHanalyzer uses the following two aggregations:

- The aggregation over a set $\mathcal{V}$ of *target values*:

$$\mathrm{ECDF}(A, f, d, \mathcal{V}, t) = \frac{1}{r|\mathcal{V}|} \sum_{v \in \mathcal{V}} \sum_{i=1}^{r} \mathbb{1}(T(A, f, d, v, i) \leq t), \tag{4}$$

  i.e., the fraction of (run,target value) pairs (i, v) for which algorithm A has identified a solution of quality at least v within a budget of at most t function evaluations.
- Given a set of functions $\mathcal{F}$ and a mapping $\mathcal{V} : \mathcal{F} \to 2^{\mathbb{R}}$ that specifies the target values to consider for each function, the ECDF can be further aggregated by the following definition:

$$\mathrm{ECDF}(A, \mathcal{F}, d, \mathcal{V}, t) = \frac{1}{r \sum_{f \in \mathcal{F}} |\mathcal{V}(f)|} \sum_{f \in \mathcal{F}} \sum_{v \in \mathcal{V}(f)} \sum_{i=1}^{r} \mathbb{1}(T(A, f, d, v, i) \leq t). \tag{5}$$

The aggregated ECDFs for function values $V(A, f, d, t, i)$ can be defined in the similar manner. By default, IOHanalyzer can generate the targets in a linear or log-linear way, as well as the predefined targets commonly used in the COCO framework. However, all of these targets can be changed by the user.

3 GRAPHICAL USER INTERFACE

The web-based **Graphical User Interface (GUI)** may be the most convenient access to IOHanalyzer for users who are not sufficiently familiar with programming in R, as well as for users who are more interested in comparing (with) data from the existing data sets collected in the performance data repository **IOHdata**. In this and the next section we use an exemplary data set called "sample_data" prepared for this article, which comprises selected performance data from the study presented in [13]. This data set is already available in the web-based GUI and the user can load it from the "Load Data from Repository" box therein (see the bottom right part in Figure 2). More precisely, we have selected from this data set the performance files for two algorithms (**Randomized Local Search (RLS)** and the **Genetic Algorithm (GA)** variant $(1, \lambda)$ GA, see [13] for a detailed description and references) on four problems in two dimensions $d \in \{16, 100\}$. All problems analyzed in [13] are of the type $f : \{0, 1\}^n \to \mathbb{R}$, and both the problem suite as well as the dataset are named **PBO** (for ***pseudo-Boolean optimization***) in **IOHproblems** and **IOHdata**, respectively. To use this data set locally, users need to create a folder named `repository/sample_data` under the home directory (i.e., `~/repository/sample_data`), download the data set from https://github.com/IOHprofiler/IOHdata/blob/master/sample_data.rds, and move the data set to this location.

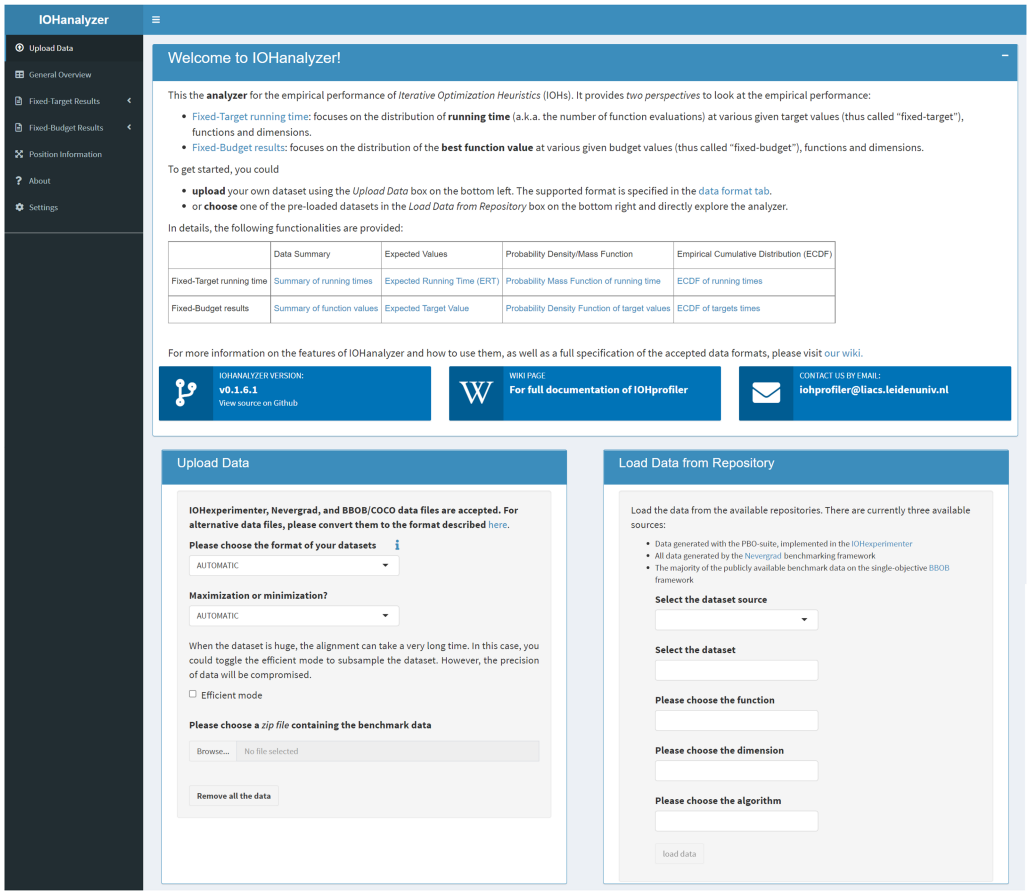


Fig. 2. Screenshot of the GUI immediately after launching the GUI server. Some general information, such as the current version and relevant links are provided at the top. The user can choose a data set from our online data repository in the drop-down menu of right column, or upload local data using the column on the left.

The IOHanalyzer GUI is invoked through the following commands:

```
R> library(IOHanalyzer)
R> runServer()
Loading required package: shiny

Listening on http://127.0.0.1:3943
```

This will start the GUI server on the local machine (hence using IP address 127.0.0.1) and a random port number. The web browser will be launched and connect to this address immediately after starting the server. The screenshot of the "welcome page" is shown in Figure 2. The performance statistics are arranged in four major sections, which can be chosen in the side menu on the left. The side menu is organized as follows.

(1) **Upload Data:** In this section users can upload their own performance data files and/or choose the data from the repository against which the data shall be compared. The format of the data files which can be uploaded is discussed in Appendix A.

(2) **General Overview:** On this tab, we show a summary of algorithms, function/problems, dimensions, the number of runs, and the best reached function values per function and algorithm appearing in the data set loaded by the user.
(3) **Fixed-Target Result:** This section covers the fixed-target performance statistics summarized in Table 1. A detailed description will be given in Section 3.2.
(4) **Fixed-Budget Results:** This section covers the fixed-budget statistics summarized in Table 2.
(5) **Position Information:** A parallel coordinate plot allows the user to display the final point resulting from each run of an algorithm. This can be used for comparing the distribution of the final solutions found across many algorithms.
(6) **About:** A concise description of the IOHanalyzer and installation guide are included here, together with information on the development team, the license, and acknowledgements.
(7) **Settings:** Here, the user can change the color schemes, the font size, and the image size used in plotting and other general settings controlling the calculation of descriptive statistics.

In general, the interactive plotting (enabled by the plotly library) is turned on by default, displaying more details in the plot when the user hovers the mouse over it, e.g., the value of a curve at the mouse cursor. The interactive plotting also allows the user to zoom in/out and to hide/show a curve from some algorithms, which will be helpful when many algorithms are rendered simultaneously. Also, all plots can be downloaded in the following formats: `pdf`, `png`, `eps`, and `svg`. Most data tables can be downloaded in csv format.

3.1 The "Upload Data" Section

The GUI interface to load the experimental data is shown in Figure 2. in which the user is asked to upload a *compressed archive*. The following compression formats are supported: `*.zip`, `*.bz`, `*.tar`, `*.xz`, `*.gz` and `*.rds` (previously processed). Note that, when the user's data set is very large to handle, it is possible to speed up the uploading (and hence plotting) procedure by toggling option `Efficient mode` on, in which the original data set is downsampled uniformly at random. Note that the data-uploading module will automatically detect whether maximization or minimization has been the objective, given the uploaded data set follows the formatting requirements described in Appendix A.

When using the online version of GUI (http://iohprofiler.liacs.nl/), the user can also load the data sets from **IOHdata**, using the "Load Data from Repository" box on the right (see Figure 2).

After loading the data, IOHanalyzer will prompt a summary table of loaded data sets in the "List of Processed Data" box on the bottom of the page (not shown in Figure 2). This allows users to check if the data loading process has been performed correctly.

3.2 The "Fixed-Target Results" Section

In the fixed-target section, the user can analyze the number of function evaluations that the algorithms performed before finding for the first time a solution meeting a certain quality criterion. This section has two main subsections, one for the performance evaluation of a *single function* and one for the evaluation of performance data for *multiple functions*. Table 1 summarizes the main fixed-target performance statistics that IOHanalyzer offers.

3.2.1 The "Single Function" Subsection. The *single function* subsection offers six different types of fixed-targets results, which are grouped as follows: (1) data summary, (2) expected runtime, (3) probability mass function, (4) cumulative distribution, (5) algorithm parameters, and (6) statistics. These groups will be described in the following paragraphs. Note that, in the header of

Table 1. The Functionalities Implemented in the *Fixed-target Results* Section of IOHanalyzer

Section	Group	Functionality	Description
Single Functions	Data Summary	*Data Overview*	The best, worst, mean, median values and success rate of selected algorithms.
		Runtime Statistics	The mean, median, quantiles, success rate and ERT at an evenly spaced sequence of targets controlled by $f_{\min}$, $f_{\max}$ and Δf.
		Runtime Samples	The running time sample at an evenly spaced sequence of targets controlled by $f_{\min}$, $f_{\max}$ and Δf.
	Expected Runtime	*ERT: single function*	The progression of ERT over targets, whose range is controlled by the user.
		Expected Runtime Comparisons	Comparing the ERT values of selected algorithms at pre-computed targets across all problem dimensions on a chosen problem.
	Probability Mass Function	*Histogram*	The histogram of the running time at a target specified by the user on **one** function.
		Probability Mass Function	The probability mass function of the running time at a target specified by the user on **one** function.
	Cumulative Distribution	*ECDF: single target*	On **one** function, the ECDF of the running time at **one** target specified by the user.
		ECDF: single function	On **one** function, ECDFs aggregated over **multiple** targets.
	Algorithm Parameters	*Expected Parameter Value*	The progression of expected value of parameters over **targets**, whose range is controlled by the user.
		Parameter Statistics	The mean, median, quantiles of recorded parameters at an evenly spaced sequence of targets controlled by $f_{\min}$, $f_{\max}$ and Δf.
		Parameter Sample	The sample of recorded parameters at an evenly spaced sequence of targets controlled by $f_{\min}$, $f_{\max}$ and Δf.
	Statistics	*Hypothesis Testing*	The two-sample Kolmogorov-Smirnov test applied on the running time at a target value for each pair of algorithms. A partial order among algorithms is obtained from the test.
Multiple Functions	Data Summary	*Multi-Function Statistics*	Descriptive statistics for all functions at a single target value.
		Multi-Function Hitting Times	Raw hitting times for all functions at a single target value.
	Expected Runtime	*ERT: all functions*	The progress of ERTs are grouped by functions and the range of targets are automatically determined.
		Expected Runtime Comparisons	The ERTs at the best target found on each function (one fixed dimension) is plotted against the function ID for each algorithm.
	Cumulative Distribution	*ECDF: all functions*	On **all** functions, ECDFs aggregated over **multiple** targets.
	Deep Statistics	*Ranking per Function*	Per-function statistical ranking procedure from the Deep Statistical Comparison Tool (DSCTool) [18].
		Omnibus Test	Use the results of the per-function ranking to perform an omnibus test using DSC.
		Posthoc comparison	Use the results of the omnibus test to perform the post-hoc comparison.
	Ranking	*Glicko2-based ranking*	For each pair of algorithms, a running time value at a given target is randomly chosen from all sample points in each round of the comparison. The glicko2-rating is used to determine the overall ranking from all comparisons.
	Portfolio	*Contribution to portfolio (Shapley-values)*	Calculate the approximated Shapley values indicating the contribution of each algorithm to the overall portfolios ECDF.

IOHanalyzer, there are two drop-down menus that allow the user to select the dimension and function, respectively. They are available in the sidebar when data has been loaded.

Group 1: Fixed-Target Results ▶ Single Function ▶ Data Summary: This group provides basic statistics on the distribution of the fixed-target running time, which are grouped in three different tables:

- Table `Data Overview`: A screenshot of this table is given in Figure 3. It simply summarizes the range of *function values* observed in the data set, with the purpose to offer its users a quick overview of the quality of the solutions that were evaluated by the algorithms. In Figure 3, we show the data overview of the "sample_data" data set, where the following values are listed for each triple of function, dimension, and algorithm: (1) the total number of runs, (2) the worst of all function values recorded in any of the runs ("worst recorded"), (3) the worst of the best function values reached in any of the runs ("worst reached"), (4) the

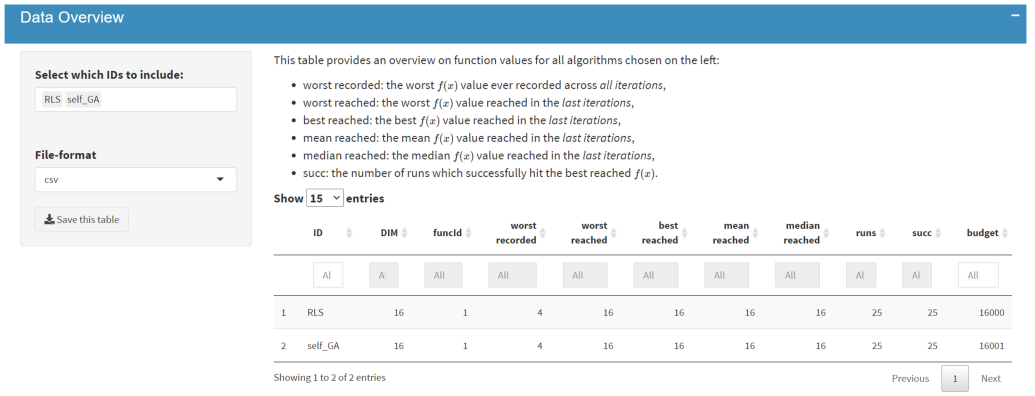


Fig. 3. Screenshot of the overview table of function values reached in the "sample_data" data set on function 1 in 16D.

best function value reached in any of the runs ("best reached"), (5) the mean (with respect to all runs) best function values ("mean reached"), (6) the median (with respect to all runs) best function value ("median reached"), and (7) the number of runs which successfully hit the "best reached" function value ("succ").

- Table `Runtime Statistics at Chosen Target Values`: A screenshot of this table is given in Figure 4. The user can set the range and the granularity of the results in the box on the left. The table shows fixed-target running times for evenly spaced target values.[2] More precisely, the table provides the success rate and the number of successful runs as defined in Equation (2), the sample mean, median, standard deviation, the sample quantiles: $Q_{2\%}, Q_{5\%}, \ldots, Q_{98\%}$, and the ***expected running time*** **(ERT)** as defined in Equation (3). The user can download this table in `csv` format, or as a LaTeX table.
- Table `Original Runtime Samples`: This table uses the same principle as the `Runtime Statistics`, but instead displays the values for each individual run. For this table, the user can choose between a "long" (all sample points are arranged in a column) and a "wide" format (all sample points are arranged in a row).

Group 2: Fixed-Target Results ▶ Single Function ▶ Expected Runtime: An interactive plot illustrates the fixed-target running times. An example of this plot is shown in Figure 5. The interactive plot can be adjusted in the menu on the left as shown in the figure. These options include showing/hiding mean and/or median values along with standard deviations and scaling the axes logarithmically. The user selects the algorithms to be displayed as well as the range of target values within which the curves are drawn. By default, this range is set as $[Q_{25\%}, Q_{75\%}]$ of all function values measured in the data set. The displayed curves can be switched on and off by clicking on the legend on the bottom of the plot.

Group 3: Fixed-Target Results ▶ Single Function ▶ Probability Mass Function: For a selected target value v, the histogram of the running time, as displayed in Figure 6, shows the number of runs i for

[2]These target values are evenly spaced between the user-specified minimum and maximum values (whose default values are set to be the extreme values found in the data) on a linear or log scale, based on the difference in order of magnitude between the extreme values found for the specified function. This same principle is used in all similar tables and plots where both a minimum and maximum target can be chosen by the user. A notable exception is the cumulative distribution functions, where *arbitrary sets* of target values can be chosen by the user.

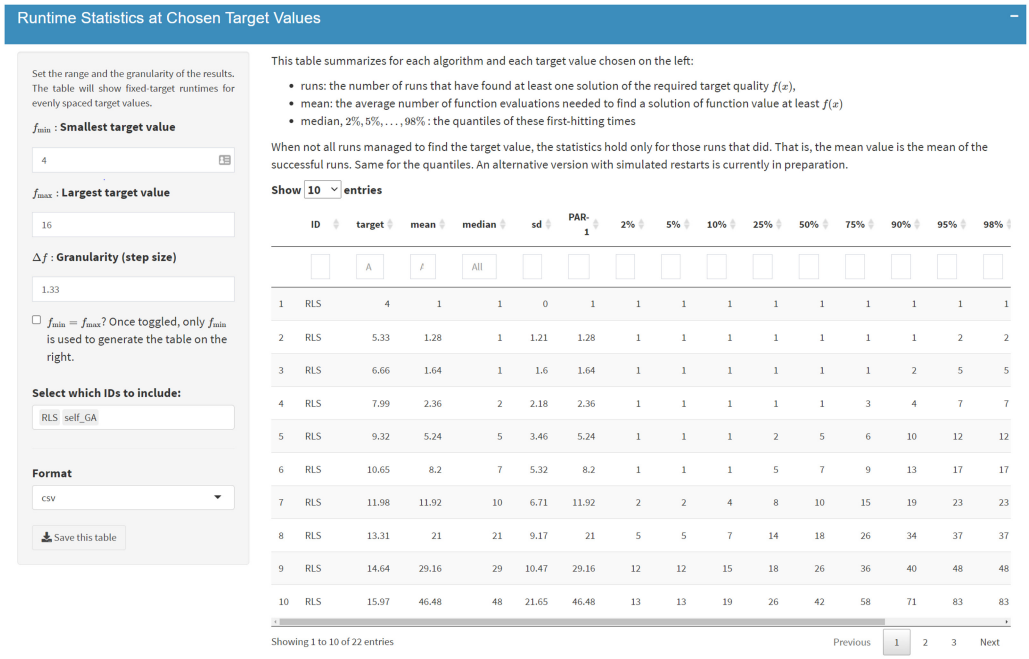


Fig. 4. Screenshot of the data summary table of some descriptive statistics on the running time.

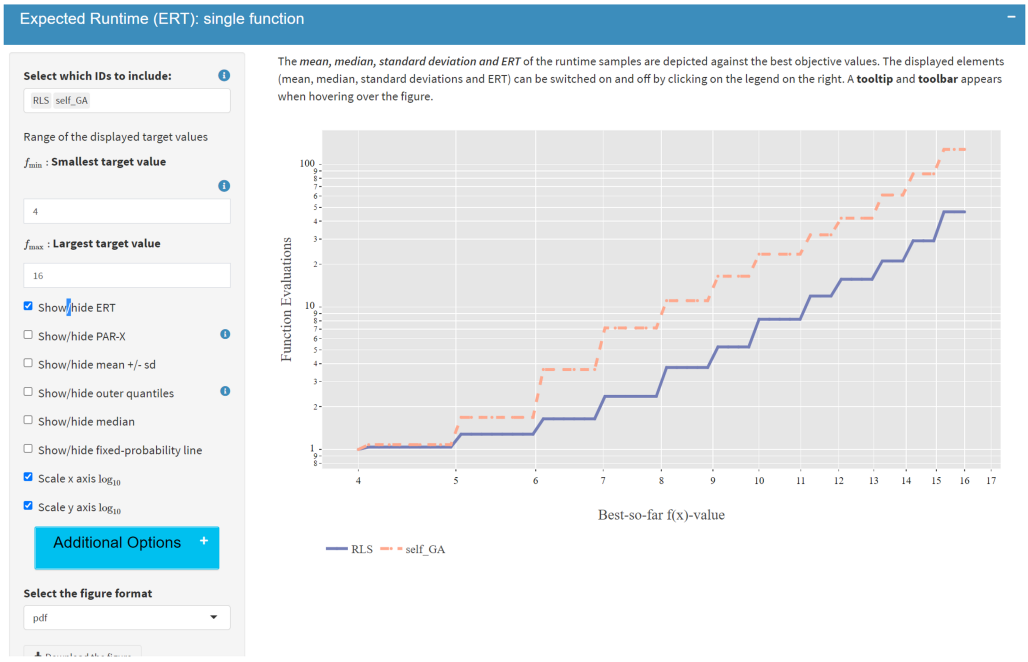


Fig. 5. Screenshot of the expected running time plot.

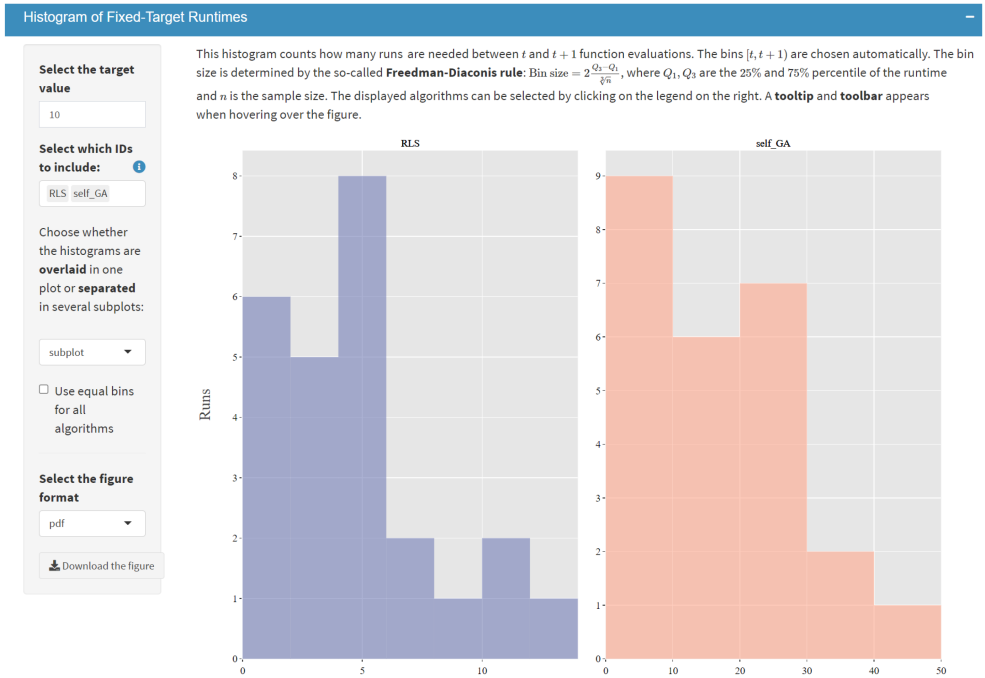


Fig. 6. Screenshot of the histogram of running time (given a fixed target of 10 on Function 1 in dimension 16).

which the running time falls into a given interval $[t, t + \Delta t)$, namely $\sum_{i=1}^{r} \mathbb{1}(t \leq T(A, f, d, v, i) < t + \Delta t)$. The bin size Δt is determined according to the *Freedman-Diaconis* rule [22], which is based on the interquartile range of the sample $\{T(A, f, d, v, i)\}_{i=1}^{r}$. The user has two options: (1) an *overlayed display*, where all algorithms are displayed in the same plot, or (2) a *separated one*, where each algorithm is displayed in an individual sub-plot, as shown in Figure 6.

In addition to the histogram, the probability mass function (Figure 7) might be helpful to get a finer look at the shape of the empirical distribution of T. The user can switch on/off the illustration of all sample points (depicted as dots), or only the empirical probability mass function itself. It is important to point out that the probability mass function is estimated in a "continuous" manner, where running time samples are considered as $\mathbb{R}$-valued and then the ***Kernel Density Estimation* (KDE)** method is taken to estimate the function.[3]

Group 4: Fixed-Target Results ▶ Single Function ▶ Cumulative Distribution. The **empirical cumulative distribution function (ECDF)** of the running time is computed for target values specified by the user. In addition to calculating ECDFs for a single target value, it is recommended to aggregate ECDFs over multiple targets, to obtain an overall performance for solving different targets. For the default target values, the tool takes 10 evenly spaced values in $[Q_{25\%}, Q_{75\%}]$ of all measured function values in a data set. Such a functionality is exemplified in Figure 8: a set of evenly spaced target values can be generated by specifying the range and step-size of the target value.

In this example, with the following setup, $f_{\min} = 4$, $f_{\max} = 16$, and $\Delta f = 1.33$, the target value sequence, $4, 5.33, 6.66, \ldots, 16$ is used to calculate the ECDF. These values are shown in the bar on

[3]Strictly speaking, this method gives imprecise estimations when there are many duplicated values, which can be quite likely in discrete optimization (such as in our examples). Improvements are planned for the future version.

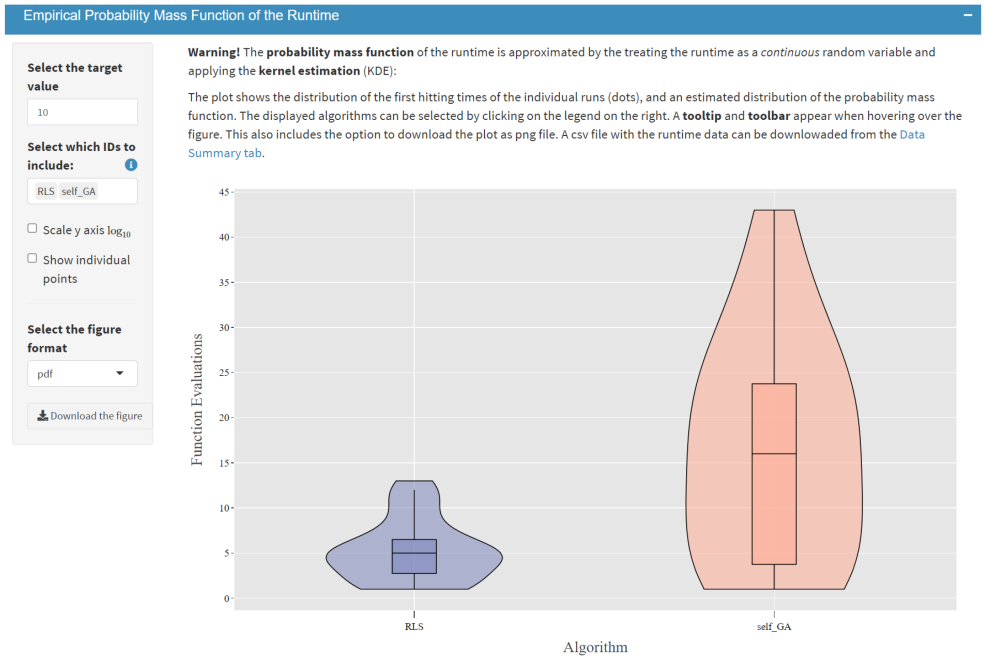


Fig. 7. Screenshot of the empirical probability mass function of running time for target value 10 on Function 1 in dimension 16).

the top of the plot, as can be seen in Figure 8. In the same figure it can be seen for algorithm RLS (blue curve) that within a budget of 24 function evaluations, around 76% of (target, run) pairs have been successful. For algorithm $(1, \lambda)$ GA (purple curve) this value is only 53%.

Group 5: Fixed-Target Results ▶ Single Function ▶ Algorithm Parameters. One of the key motivations to build IOHprofiler was the ability to analyze, in detail, the evolution of control parameters which are adjusted during the search. Such dynamic parameters can be found in most state-of-the-art heuristics. While in numerical optimization a non-static choice of the search radius, for example, is needed to eventually converge to a local optimum, dynamic parameters are also more and more common in discrete and mixed-integer optimization heuristics [10, 39]. In the fifth group of fixed-target results for a single function, the evolution of the parameters is linked to the quality of the best-so-far solutions that have been evaluated. In the experimentation (i.e., data generation) phase, the user selects which parameters are logged along with the evaluated function values. These values are then automatically detected by IOHanalyzer and can be chosen in this group for analysis.

As with the interactive plots on expected running time, the user can choose the range of targets, which parameters and algorithms to plot, and the scale (either logarithmic or linear) of x- and y-axis. We omit the example for parameters as the GUI is similar to the one in Figure 5. As with "Fixed-Target Results ▶ Single Function ▶ Data Summary", this subsection also provides for each parameter tables of descriptive statistics (sample mean, median, standard deviation, and some quantiles) as well as the original parameter values.

Group 6: Fixed-Target Results ▶ Single Function ▶ Statistics. To address the robustness of empirical comparisons, the samples from all algorithm must undergo a proper statistical test procedure [34]. In IOHanalyzer, a standard multiple testing procedure is implemented to compare the fixed-target running time for each pair of algorithms on a single function, for which the

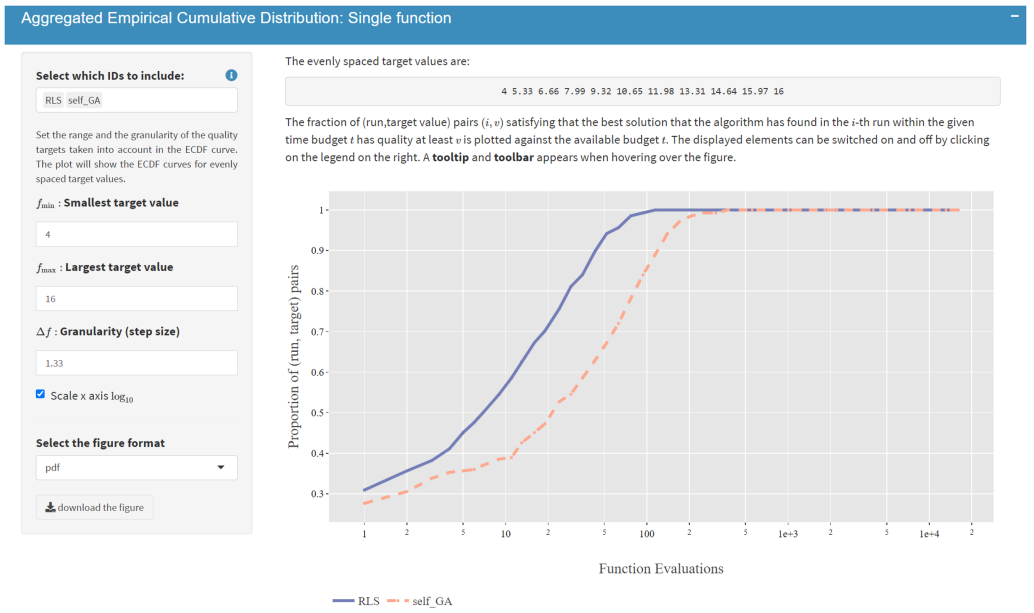


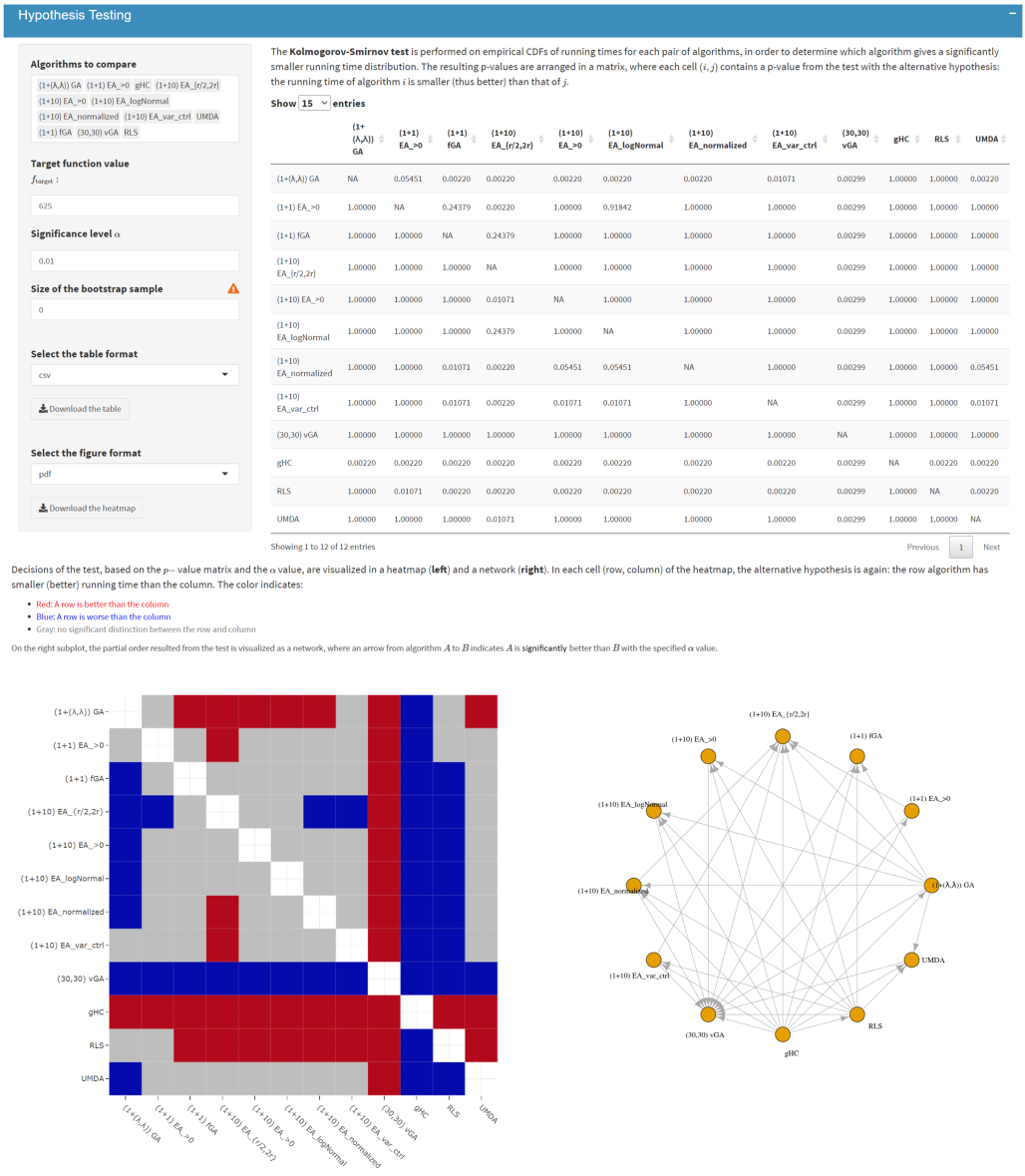
Fig. 8. Screenshot of aggregated ECDF curve over multiple targets.

well-known Kolmogorov-Smirnov test is applied to the ECDFs of running times. Moreover, the Bonferroni procedure is used to correct the p-value in multiple testing. To demonstrate this functionality, we show, in Figure 9, the testing outcome of a data set on 12 reference algorithms.[4] on the PBO problem set from [13], instead of the exemplary two-algorithm data set used previously. Here, the test is conducted across all 12 algorithms on function $f1$ and dimension 64 with a confidence level of 0.01. The result of this procedure is illustrated by a table of pairwise p-values, a color matrix of the statistical decision, and a graph depicting the partial order induced by the test (i.e., an arrow pointing from Algorithm 1 to Algorithm 2 is to be read as Algorithm 1 dominating Algorithm 2 with statistical significance. As with all tables and figures in IOHanalyzer, these can be downloaded in several formats, including `*.tex` and `*.csv` for tables and `*.pdf` and `*.eps` for figures.

3.2.2 The "Multiple Functions" Subsection. This subsection contains three groups of fixed-target results for multiple functions: (1) expected runtime comparison across all functions for one dimension, (2) aggregated Empirical Cumulative Distribution over all functions, and (3) Glicko2-based ranking.

Group 1: Fixed-Target Results ▶ Multiple Functions ▶ Expected Runtime: In this group, the tool depicts the ERT values against multiple functions as a radar-plot, as shown in Figure 10. For each function, the target value used for calculating the ERT is determined by default as follows: firstly, for each algorithm, we obtained the 2% percentile of the best function values reached in multiple runs. Secondly, we took the largest value among all such 2% percentiles as the target value on this function. In this radar-plot, we revert the axis such that the bigger ERT values are further away from the center of the circle compared to smaller ones, indicating that better algorithms will cover a larger area.

[4]This data set is available at https://github.com/IOHprofiler/IOHdata/blob/master/iohprofiler/2019gecco-ins1-11run.rds. It can be loaded to the web-based GUI by selecting the PBO data set in the "upload data" section. The data set comprises the results of the experimental study described in [13].



	(1+ (λ,λ)) GA	(1+1) EA_>0	(1+1) fGA	(1+10) EA_{r/2,2r}	(1+10) EA_>0	(1+10) EA_logNormal	(1+10) EA_normalized	(1+10) EA_var_ctrl	(30,30) vGA	gHC	RLS	UMDA
(1+(λ,λ)) GA	NA	0.05451	0.00220	0.00220	0.00220	0.00220	0.00220	0.01071	0.00299	1.00000	1.00000	0.00220
(1+1) EA_>0	1.00000	NA	0.24379	0.00220	1.00000	0.91842	1.00000	1.00000	0.00299	1.00000	1.00000	1.00000
(1+1) fGA	1.00000	1.00000	NA	0.24379	1.00000	1.00000	1.00000	1.00000	0.00299	1.00000	1.00000	1.00000
(1+10) EA_{r/2,2r}	1.00000	1.00000	1.00000	NA	1.00000	1.00000	1.00000	1.00000	0.00299	1.00000	1.00000	1.00000
(1+10) EA_>0	1.00000	1.00000	1.00000	0.01071	NA	1.00000	1.00000	1.00000	0.00299	1.00000	1.00000	1.00000
(1+10) EA_logNormal	1.00000	1.00000	1.00000	0.24379	1.00000	NA	1.00000	1.00000	0.00299	1.00000	1.00000	1.00000
(1+10) EA_normalized	1.00000	1.00000	1.00000	0.01071	0.00220	0.05451	0.05451	NA	1.00000	0.00299	1.00000	1.00000	0.05451
(1+10) EA_var_ctrl	1.00000	1.00000	1.00000	0.01071	0.00220	0.01071	0.01071	1.00000	NA	0.00299	1.00000	1.00000	0.01071
(30,30) vGA	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	NA	1.00000	1.00000	1.00000
gHC	0.00220	0.00220	0.00220	0.00220	0.00220	0.00220	0.00220	0.00220	0.00299	NA	0.00220	0.00220
RLS	1.00000	0.01071	0.00220	0.00220	0.00220	0.00220	0.00220	0.00220	0.00299	1.00000	NA	0.00220
UMDA	1.00000	1.00000	1.00000	0.01071	1.00000	1.00000	1.00000	1.00000	0.00299	1.00000	1.00000	NA

Fig. 9. Screenshot of the multiple testing procedure applied on all 12 reference algorithms on function $f1$ and dimension 625. The table shows the p-values resulting from the pairwise KS-test between each pair of algorithms. Then, based on the $\alpha = 0.01$, the resulting hypothesis-rejections are shown in both the matrix-plot and the network.

Group 2: Fixed-Target Results ▶ *Multiple Functions* ▶ *Cumulative Distribution:* In this group, ECDFs of running times are aggregated across multiple functions, as defined in Equation (5). This functionality is illustrated in Figure 11: a table of pre-calculated target values are provided for each function (all test functions are included by default). This table of targets can easily be edited directly in the GUI, or by a downloading-editing-uploading procedure (which should, of course, not change the format of the tables, just the values).

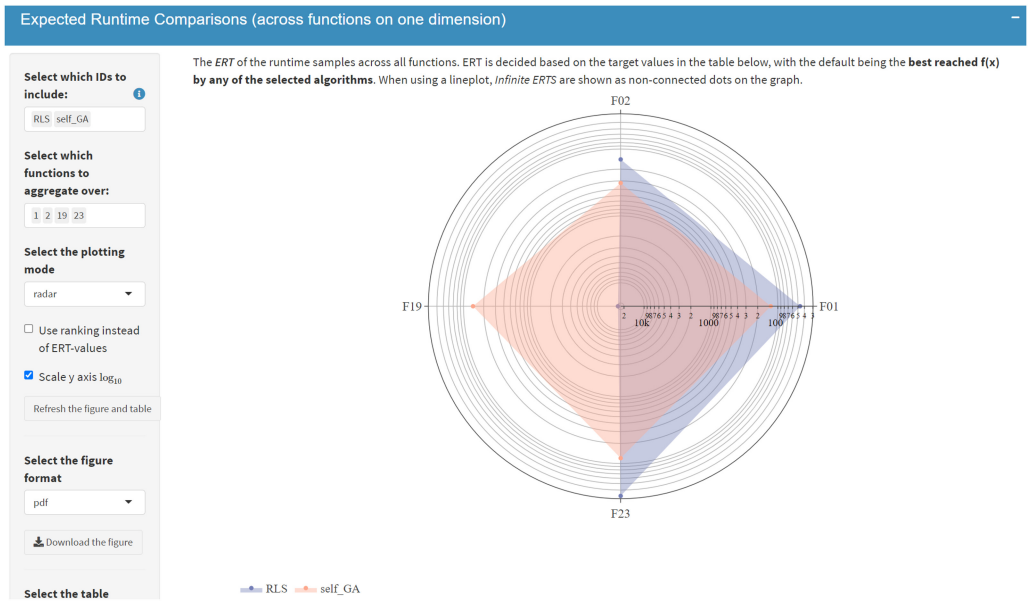


Fig. 10. Screenshot of ERT of RLS and the $(1, \lambda)$ GA on four PBO problems, where the ERT values are shown for each selected problem in a radar plot with inverted axis (values are decreasing when moving away from the center). Loosely speaking, an algorithm with a larger span on the plot is considered better, e.g., RLS dominates $(1, \lambda)$ GA on problem F01, F02, and F23 while the latter is superior on F19.

Note that for these ECDF-figures, the corresponding Area Under the Curve **(AUC)** can also be calculated to get a single value for each algorithm. These AUC-tables are available in the same tab as the ECDF plot.

Group 3: Fixed-Target Results ▶ Multiple Functions ▶ Ranking: This group provides a ranking functionality to compare algorithms across multiple functions and dimensions, in which we employ the Glicko-2 rating system [25] (commonly used in chess games) to rank the algorithms, based on multiple simulated games between them (25 by default). In each game, for every function and dimension, the winner of each pair of algorithms is determined by sampling from the running time values (given a target value) uniformly at random and checking which random sample is better. An overall rating is computed from those games' outcome, which is then used to rank the algorithms. Algorithms with a better rank win more rounds than those with a poor rank, indicating that when sampling a runtime on an arbitrary function, these algorithms tend to perform better. This way of ranking allows us to aggregate performance over an arbitrary number of functions and targets, while inherently managing uncertainty of the performance metrics by doing the repeated rounds and comparing individual values for each 'game'.

3.3 The "Fixed-Budget Results" Section

The fixed-budget section offers performance analysis for the quality of the best solution that the algorithms could identify within a given budget of function evaluations. The results are similar to those presented in the fixed-target section (Section 3.2) except that subsection "Fixed-Budget Results ▶ Multiple Functions ▶ Cumulative Distribution" is still under development and hence not yet available at the time of writing. Table 2 summarizes the main functionalities.

Fig. 11. Screenshot of aggregated ECDF curve across multiple functions and targets.

3.4 The "Position Information" Section

Within this section, the user can visualize the final search points in their decision space in a parallel coordinate plot. In version 0.1.6.1, this functionality is only supported for data generated by the SOS-framework [7]. A processed dataset in this format is available on

Table 2. The Functionalities Implemented in the *Fixed-budget Results* of IOHanalyzer

Section	Group	Functionality	Description
Single Function	Data Summary	*Data Overview*	The minimum and maximum of running times for selected algorithms.
		Target Value Statistics	The mean, median, and quantiles of the function value at a sequence of budgets controlled by $B_{\min}$, $B_{\max}$ and ΔB.
		Target Value Samples	The function value samples at an evenly spaced sequence of budgets controlled by $B_{\min}$, $B_{\max}$ and ΔB.
	Expected Target Value	*Expected Target Value: single function*	The progression of expected function values over budgets, whose range is controlled by the user.
	Probability Density Function	*Histogram*	The histogram of the function value a user-chosen budget.
		Probability Density Function	The probability density function (using the Kernel Density Estimation) of the function value at a user-chosen budget.
	Cumulative Distribution	*ECDF: single budget*	On **one** function, the ECDF of the function value at **one** budget specified by the user.
		ECDF: single function	On **one** function, ECDFs aggregated over **multiple** budgets.
		Area Under the ECDF	On **one** functions, the area under ECDFs of function values that are aggregated over **multiple** budgets.
	Algorithm Parameters	*Expected Parameter Value*	The progression of expected value of parameters over the **budget**, whose range is controlled by the user.
		Parameter Statistics	The mean, median, quantiles of recorded parameters at an evenly spaced sequence of budgets controlled by $B_{\min}$, $B_{\max}$ and ΔB.
		Parameter Sample	The sample of recorded parameters at an evenly spaced sequence of budgets controlled by $B_{\min}$, $B_{\max}$ and ΔB.
	Statistics	*Hypothesis Testing*	The two-sample Kolmogorov-Smirnov test applied on the running time at a target value for each pair of algorithms. A partial order among algorithms is obtained from the test
Multiple Functions	Data Summary	*Multi-Function Statistics*	Descriptive statistics for all functions at a single target value.
		Multi-Function Hitting Times	Raw hitting times for all functions at a single target value.
	Expected Target Value	*Expected Target Value: all functions*	The same as above except that the expected function values are grouped by functions and the range of budgets are automatically determined.
		Expected Target Value: Comparison	The expected function value at the largest budget found on each function is plotted against the function ID for each algorithm.
	Deep Statistics	*Ranking per Function*	Per-function statistical ranking procedure from the Deep Statistical Comparison Tool (DSCTool) [18].
		Omnibus Test	Use the results of the per-function ranking to perform an omnibus test using DSC.
		Posthoc comparison	Use the results of the omnibus test to perform the post-hoc comparison.
	Ranking	*Glicko2-based ranking*	For each pair of algorithms, a function value at a given budget is randomly chosen from all sample points in each round of the comparison. The glicko2-rating is used to determine the overall ranking from all comparisons.

IOHdata.[5] This dataset contains a DE-variant which was generated for the analysis of Structural Bias in DE [61], which can be confirmed visually using the parallel coordinate plot functionality.

Development on extending this position-based functionality to other data sources and more types of analysis is in progress.

3.5 Command-Line Interface

In addition to the web-based graphical interface, we provide a **command-line interface (CLI)** via a feature-rich R-package, which allows for more fine-grained control of various types of analysis and visualization described in this section. All functionality discussed in this paper can also be

[5] https://github.com/IOHprofiler/IOHdata/tree/master/SOS.

accessed through this CLI. A demonstration of the key aspects of this CLI on an example data set is available on the wiki page https://iohprofiler.github.io/IOHanalyzer/R/.

4 DISCUSSION AND OUTLOOK

We have presented IOHanalyzer, a highly versatile environment for evaluating the performance data of iterative optimization heuristics. IOHanalyzer – and, more generally, the whole IOHprofiler project – are under continuous development. Extensions planned for the near future comprise, most notably, an **increased compatibility with the following benchmarking environments and platforms**:

- **General-purpose benchmarking platforms.** As mentioned, IOHanalyzer has already been extended to visualize data sets generated with Facebook's Nevergrad platform [52]. We are now working on various other interfaces, which will allow Nevergrad users to use the logging functionalities of IOHprofiler and to access the problems made available in IOHprofiler. Likewise, we are working towards an interface that allows users of IOHprofiler to more easily access the benchmark problems of Nevergrad.
- **Modular algorithm frameworks and automated configuration tools.** The modular algorithm framework ParadisEO [4] and the modular CMA-ES framework proposed in [59] have already been integrated into IOHprofiler. An integration of other modular algorithm frameworks such as those presented in [21, 23, 49, 54], together with automated algorithm configuration tools such as irace [47], SMAC [36], hyperband [44], and MIP-EGO [64], will pave the way to a more generic research environment for automated configuration of optimization algorithms. For supervised learning approaches, we shall interface IOHprofiler and feature-extraction techniques such as those collected in the R package *flacco* [41].
- **Collections and generators of benchmark problems.** As we are doing for the Nevergrad platform, we are working on easier interfaces with other collections of benchmark problems as well as with generators of these. Already implemented are the 23 discrete problems described in the PBO suite from [13], a (slight variation of the) W-model [65] (see https://iohprofiler.github.io/ for details of our implementation), and the 24 numeric optimization problems from the BBOB suite [31] of the COCO platform [30]. Extensions to other problem types, such as multi-objective or noisy optimization, are also being considered.
- **Other statistical evaluation techniques.** Several interfaces of IOHanalyzer with tools aimed at visualizing or analyzing the performance data are currently under consideration. For example, an integration of the software to efficiently compute *empirical attainment functions* provided by [20] could help to visualize the time-quality-robustness trade-off of IOHs. Building on the initial study [6] we are considering the integration of the rank-based Bayesian inference statistics, which were introduced to the evolutionary computation community via [5]. Other advanced statistical procedures may also be added, e.g., the ***Deep Statistical Comparison tool*** **(DSCtool)** suggested in [17].
- **Performance aggregation and anytime performance measures.** Finally, we are also implementing different ways to aggregate performances over multiple test problems. In this respect we are, among others, looking into so-called performance profiles [48], which is the empirical cumulative distribution of normalized performance values across problems. Related to this, we observe an increasing interest in measuring and/or optimizing for anytime performance metrics [3, 37]. We are carefully observing this development and are considering different ways to extend the statistics of IOHanalyzer with other suggested anytime performance measures.

Computational details. The results in this paper were obtained using R 4.1.1 and version 0.1.6.1 of IOHanalyzer with the following packages, Rcpp 1.0.7, shiny 1.6.0 and plotly 4.9.4.1. For the C/C++ compiler, gcc version 10.3.0 is used (for Rcpp).

APPENDIX

A SUPPORTED DATA FORMAT

IOHanalyzer aims to be as flexible as possible, and to achieve this, it supports data from many different sources. This means that data can be presented in many different formats. At the time of writing, the list of supported formats is as follows:

- IOHprofiler data format, which is motivated by and modified from the COCO data format.
- COCO data format as defined in [28].
- The Nevergrad format from [52].
- The SOS format from [7].
- A "two-column" format is a simplified version of the IOHprofiler format. We describe this format in the upcoming paragraph on "Raw-data".

When loading the data in the programming interface (and in the graphical user interface as well), it is not necessary to specify its format as IOHanalyzer attempts to detect this automatically. For most data formats,[6] data files are organized in the same manner within the file system. The structure of data files is as follows:

```
./
├── IOHprofiler_f1.info
├── data_f1
│   ├── IOHprofiler_f1_DIM64.dat
│   ├── IOHprofiler_f1_DIM64.cdat
│   ├── IOHprofiler_f1_DIM100.dat
│   ├── IOHprofiler_f1_DIM100.cdat
│   └── ...
├── IOHprofiler_f2.info
├── data_f2
│   ├── IOHprofiler_f2_DIM64.dat
│   ├── IOHprofiler_f2_DIM64.cdat
│   ├── IOHprofiler_f2_DIM100.dat
│   ├── IOHprofiler_f2_DIM100.cdat
│   └── ...
├── IOHprofiler_f3.info
└── ...
```

Generally, in the folder (e.g., `./` here) that contains the data set, the following files are mandatory for IOHanalyzer:

- *Meta-data* (`.info`) files summarize the algorithmic performance for each problem instance, with the following naming convention: `IOHprofiler_f1.info` for problem $f1$. Note that one meta-data file can consist of several dimensions. Please see the details below.

[6] The IOHprofiler, COCO and the *two-column* formats have the same basic structure, while Nevergrad uses pure csv files instead, and will thus not be discussed in this section.

- *Raw-data* (`.dat`, `.cdat` etc) are csv-like files that contain performance information indexed by the running time. Raw-data files are named in a similar manner as the meta-data files, for example, `IOHprofiler_f1_DIM100.dat` for problem $f1$ and dimension 100. Raw-data files are organized in sub-folders for each problem. It is important to note that those three data formats only differ in the structure of the raw-data files.

Meta-data. When benchmarking, it is common to specify a number of different dimensions, functions and instances, resulting in a quite large number of data files (e.g., `*.dat` files). It would make the data organization more structured if some meta data are provided. Here, the meta data are implemented in a format that is very similar to that in the well-known `COCO` environment. The meta data are indicated with suffix `.info`. A small example is provided as follows:

```
suite = 'PBO', funcId = 19, DIM = 16, algId = 'self_GA'
%
data_f19/IOHprofiler_f19_DIM16.dat, 1:16001|3.20000e+001, 1:16001|3.20000e+001,
1:16001|3.20000e+001, 1:16001|2.80000e+001, 1:16001|3.20000e+001
suite = 'PBO', funcId = 19, DIM = 100, algId = 'self_GA'
%
data_f19/IOHprofiler_f19_DIM100.dat, 1:100001|1.92000e+002, 1:100001|1.88000e+002,
1:100001|1.80000e+002, 1:100001|1.76000e+002, 1:100001|1.76000e+002
```

Note that the IOHanalyzer relies on the meta-data present in the info-files for its processing of associated data. Thus, it is crucial to ensure that these files are correct, especially when converting data from other formats into IOHprofiler or *two-column* formats. The meta data is structured in the following "three-line" format (two examples of this "three-line" structure are provided in the example above), storing the high-level information on all instances of a tuple of (dimension, function).

- **The first line** stores some meta-information of the experiment as (name, value) pairs. Note that such pairs are separated by commas and three names, `funcId`, `DIM` and `algId` are *case-sensitive* and *mandatory*.
- **The second line** always starts with a single `%`, indicating what follows this symbol should be the general comments from the user on this experiment. By default, it is left empty.
- **The third line** starts with the relative path to the actual data file, followed by the meta-information obtained on each instance, with the following format:

$$\underbrace{1}_{\text{instance ID}} : \underbrace{1953125}_{\text{running time}} \mid \underbrace{5.59000e+02}_{\text{best-so-far f(x)}}$$

  By default, the data files (`*.dat`, `*.cdat`, `*.tdat`, dots) are organized in the group of test functions, which are again stored in sub-folders with naming convention: `data_[function ID]/`, e.g., `data_f10/`. Moreover, when several dimensions are tested, the corresponding information above is written into the meta data one after the other.

Raw-data. Despite the fact that different methods can be used to store data (resulting in four types of data file, which also determines the extension, e.g., `.dat` or `.cdat`), the files take the same format, which is adapted from `csv` format to accommodate multiple runs/instances. An example of the structure of these files is shown below.

```
"function evaluation"  "current f(x)"  "best-so-far f(x)"  "parameter"  ...
                    1    +2.95000e+02       +2.95000e+02    0.000000  ...
                    2    +2.96000e+02       +2.96000e+02    0.001600  ...
                    4    +3.07000e+02       +3.07000e+02    0.219200  ...
                    9    +3.11000e+02       +3.11000e+02    0.006400  ...
                   12    +3.12000e+02       +3.12000e+02    0.001600  ...
                   16    +3.16000e+02       +3.16000e+02    0.006400  ...
                   20    +3.17000e+02       +3.17000e+02    0.001600  ...
                   23    +3.28000e+02       +3.28000e+02    0.027200  ...
                   27    +3.39000e+02       +3.39000e+02    0.059200  ...
"function evaluation"  "current f(x)"  "best-so-far f(x)"  "parameter"  ...
                    1    +3.20000e+02       +3.20000e+02    1.000000  ...
                   24    +3.44000e+02       +3.44000e+02    2.000000  ...
                   60    +3.64000e+02       +3.64000e+02    3.000000  ...
"function evaluation"  "current f(x)"  "best-so-far f(x)"  "parameter"  ...
                  ...             ...                 ...          ...  ...
```

Note that, each *separation line* (line that starts with `"function evaluation"`) serves as a separator among different independent runs of the same algorithm. Therefore, it is clear that the data block between two separation lines corresponds to a single run on a combination of dimension, function, and instance. In addition, a parameter value (named `"parameter"`) is also tracked in this example and recording more parameter value is also facilitated (see below). Columns `"current f(x)"` and `"best-so-far f(x)"` stand for the current function value and the best one found so far, respectively. Here, `"current f(x)"` stands for the function value observed when the corresponding number of function evaluation is performed while `"best-so-far f(x)"` keeps track of the best function value observed since the beginning of one run. Only two columns, `"function evaluation"` and `"best-so-far f(x)"` are **mandatory** in this format. The **two-column** data format mentioned previously is to describe the minimal case where only those two columns are present in the raw data.

In order to emulate the data format generated by IOHexperimenter, it is very important to pay attention to the following principles for the data format:

- The *double quotation* (") in the separation line shall always be kept and it cannot be replaced with single quotation (').
- The numbers in the record can either be written in the plain or scientific notation.
- To separate the columns, *a single space or tab* can be used (only one of them should be used consistently in a single data file).
- If the performance data is tracked in the improvement-based scheme, where a row is written only if the `"best-so-far f(x)"` is improved, the user must make sure that each block of records (as divided by the separation line) ends with the last function evaluation. This allows the used budget to be extracted from the data-file when required.
- Each data line should contain a complete record. Incomplete data lines will be dropped when loading the data into IOHanalyzer.
- The parameter columns, which record the state of (dynamic) internal parameters during the search, are fully customizable. The user can specify which parameter to track when running their algorithm using the IOHexperimenter. For more details on how to setup this parameter tracking in IOHexperimenter, please refer to our wiki page (https://iohprofiler.github.io/IOHexp/Cpp/#using-logger).
- In case the quotation mark is needed in the parameter name, a single quotation (') should be used.

ACKNOWLEDGMENTS

We thank Arina Buzdalova, Maxim Buzdalov, Raphaël Cosson, Johann Dréo, Tome Eftimov, Pietro S. Oliveto, Ofer M. Shir, Markus Wagner, and Thomas Weise for various suggestions that have helped us improve IOHanalyzer and the presentation of this tool. We also thank the COCO team, in particular Anne Auger, Dimo Brockhoff, and Nikolaus Hansen, as well as the Nevergrad team, Jeremy Rapin and Olivier Teyaud, for help with their platforms. We also thank the anonymous reviewers of TELO whose comments have helped us to improve the presentation and reproducibility of this work.

REFERENCES

[1] Anne Auger and Benjamin Doerr. 2011. *Theory of Randomized Search Heuristics*. World Scientific.

[2] Amine Aziz-Alaoui, Carola Doerr, and Johann Dréo. 2021. Towards large scale automated algorithm design by integrating modular benchmarking frameworks. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'21, Companion Material)*. ACM, 1365–1374. https://doi.org/10.1145/3449726.3463155

[3] Jakob Bossek, Pascal Kerschke, and Heike Trautmann. 2020. A multi-objective perspective on performance assessment and automated selection of single-objective optimization algorithms. *Appl. Soft Comput.* 88 (2020), 105901. https://doi.org/10.1016/j.asoc.2019.105901

[4] Sébastien Cahon, Nordine Melab, and El-Ghazali Talbi. 2004. ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. *J. Heuristics* 10, 3 (2004), 357–380. https://doi.org/10.1023/B:HEUR.0000026900.92269.ec

[5] Borja Calvo, Josu Ceberio, and Jose A. Lozano. 2018. Bayesian inference for algorithm ranking analysis. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'18, Companion Material)*. ACM, 324–325.

[6] Borja Calvo, Ofer M. Shir, Josu Ceberio, Carola Doerr, Hao Wang, Thomas Bäck, and Jose A. Lozano. 2019. Bayesian performance analysis for black-box optimization benchmarking. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'19, Companion Material)*. ACM, 1789–1797.

[7] Fabio Caraffini and Giovanni Iacca. 2020. The SOS platform: Designing, tuning and statistically benchmarking optimisation algorithms. *Mathematics* 8, 5 (2020), 785.

[8] Winston Chang, Joe Cheng, JJ Allaire, Yihui Xie, and Jonathan McPherson. 2019. *shiny: Web Application Framework for R*. https://CRAN.R-project.org/package=shiny. R package version 1.3.2.

[9] Jacob de Nobel, Diederick Vermetten, Hao Wang, Carola Doerr, and Thomas Bäck. 2021. 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)*. ACM, 1375–1384. https://doi.org/10.1145/3449726.3463167

[10] Benjamin Doerr and Carola Doerr. 2020. Theory of parameter control for discrete black-box optimization: Provable performance gains through dynamic parameter choices. In *Theory of Evolutionary Computation: Recent Developments in Discrete Optimization*. Springer, 271–321. https://doi.org/10.1007/978-3-030-29414-4_6 Also available online at https://arxiv.org/abs/1804.05650.

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

[12] Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck. 2018. IOHprofiler: A benchmarking and profiling tool for iterative optimization heuristics. *arXiv e-prints*, Article arXiv:1810.05281 (Oct 2018), arXiv:1810.05281. arXiv:cs.NE/1810.05281

[13] Carola Doerr, Furong Ye, Naama Horesh, Hao Wang, Ofer M. Shir, and Thomas Bäck. 2020. Benchmarking discrete optimization heuristics with IOHprofiler. *Appl. Soft Comput.* 88 (2020), 106027. https://doi.org/10.1016/j.asoc.2019.106027

[14] Carola Doerr, Furong Ye, Sander van Rijn, Hao Wang, and Thomas Bäck. 2018. 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)*. ACM, 951–958.

[15] Marco Dorigo and Thomas Stützle. 2004. *Ant Colony Optimization*. MIT Press.

[16] Gunter Dueck and Tobias Scheuer. 1990. Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing. *J. Comput. Phys.* 90 (1990), 161–175.

[17] Tome Eftimov, Peter Korosec, and Barbara Korousic-Seljak. 2017. A novel approach to statistical comparison of meta-heuristic stochastic optimization algorithms using deep statistics. *Inf. Sci.* 417 (2017), 186–215. https://doi.org/10.1016/j.ins.2017.07.015

[18] Tome Eftimov, Gasper Petelin, and Peter Korosec. 2020. DSCTool: A web-service-based framework for statistical comparison of stochastic optimization algorithms. *Appl. Soft Comput.* 87 (2020), 105977. https://doi.org/10.1016/j.asoc.2019.105977

[19] A. E. Eiben and James E. Smith. 2015. *Introduction to Evolutionary Computing, Second Edition*. Springer. https://doi.org/10.1007/978-3-662-44874-8

[20] Carlos M. Fonseca, Andreia P. Guerreiro, Manuel López-Ibáñez, and Luís Paquete. 2011. On the computation of the empirical attainment function. In *Proc. of Evolutionary Multi-Criterion Optimization (EMO'11) (LNCS)*, Vol. 6576. Springer, 106–120.

[21] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research* 13 (2012), 2171–2175.

[22] David Freedman and Persi Diaconis. 1981. On the histogram as a density estimator: L2 theory. *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete* 57, 4 (01 Dec 1981), 453–476.

[23] Christian Gagné and Marc Parizeau. 2006. Genericity in evolutionary computation software tools: Principles and case study. *International Journal on Artificial Intelligence Tools* 15, 2 (April 2006), 173–194.

[24] Marcus Gallagher and Sobia Saleem. 2018. Exploratory landscape analysis of the MLDA problem set. In *Proceedings of PPSN18 Workshops*.

[25] Mark E. Glickman. 2012. Example of the Glicko-2 system. *Boston University* (2012), 1–6.

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

[27] Nikolaus Hansen, Anne Auger, Dimo Brockhoff, Dejan Tusar, and Tea Tusar. 2016. COCO: Performance assessment. *CoRR* abs/1605.03560 (2016). arXiv:1605.03560 http://arxiv.org/abs/1605.03560.

[28] Nikolaus Hansen, Anne Auger, Steffen Finck, and Raymond Ros. 2009. *Real-Parameter Black-Box Optimization Benchmarking 2009: Experimental Setup*. Research Report RR-6828. INRIA. https://hal.inria.fr/inria-00362649.

[29] Nikolaus Hansen, Anne Auger, Raymond Ros, Steffen Finck, and Petr Posík. 2010. Comparing results of 31 algorithms from the black-box optimization benchmarking BBOB-2009. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'10, Companion Material)*. ACM, 1689–1696.

[30] Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tušar, and Dimo Brockhoff. 2020. COCO: A platform for comparing continuous optimizers in a black-box setting. *Optimization Methods and Software* (2020), 1–31. https://doi.org/10.1080/10556788.2020.1808977 arXiv:https://doi.org/10.1080/10556788.2020.1808977

[31] Nikolaus Hansen, Steffen Finck, Raymond Ros, and Anne Auger. 2009. *Real-Parameter Black-Box Optimization Benchmarking 2009: Noisy Functions Definitions*. Research Report RR-6869. INRIA.

[32] William E. Hart, Carl D. Laird, Jean-Paul Watson, David L. Woodruff, Gabriel A. Hackebeil, Bethany L. Nicholson, and John D. Siirola. 2017. *Pyomo-optimization Modeling in Python*. Vol. 67. Springer.

[33] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer. https://doi.org/10.1007/978-0-387-84858-7

[34] Myles Hollander, Douglas A. Wolfe, and Eric Chicken. 2013. *Nonparametric Statistical Methods*. Vol. 751. John Wiley & Sons.

[35] Naama Horesh, Thomas Bäck, and Ofer M. Shir. 2019. Predict or screen your expensive assay: DoE vs. surrogates in experimental combinatorial optimization. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'19)*. ACM, 274–284.

[36] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *Proc. of Learning and Intelligent Optimization (LION'11) (LNCS)*, Carlos A. Coello Coello (Ed.), Vol. 6683. Springer, 507–523. https://doi.org/10.1007/978-3-642-25566-3_40

[37] Alexandre D. Jesus, Arnaud Liefooghe, Bilel Derbel, and Luís Paquete. 2020. Algorithm selection of anytime algorithms. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'20)*. ACM, 850–858. https://doi.org/10.1145/3377930.3390185

[38] Donald R. Jones. 2001. A taxonomy of global optimization methods based on response surfaces. *Journal of Global Optimization* 21, 4 (01 Dec 2001), 345–383.

[39] Giorgos Karafotias, Mark Hoogendoorn, and A. E. Eiben. 2015. Parameter control in evolutionary algorithms: Trends and challenges. *IEEE Transactions on Evolutionary Computation* 19 (2015), 167–187.

[40] Maarten Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. 2002. Evolving objects: A general purpose evolutionary computation library. *Artificial Evolution* 2310 (2002), 829–888. http://www.lri.fr/~marc/EO/EO-EA01.ps.gz. Latest release available on https://nojhan.github.io/paradiseo/.

[41] Pascal Kerschke and Heike Trautmann. 2016. The r-package FLACCO for exploratory landscape analysis with applications to multi-objective optimization problems. In *Proc. of IEEE Congress on Evolutionary Computation (CEC'16)*. IEEE, 5262–5269. https://doi.org/10.1109/CEC.2016.7748359 Flacco is available at http://kerschke.github.io/flacco/.

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

[43] Pedro Larrañaga and José Antonio Lozano (Eds.). 2002. *Estimation of Distribution Algorithms*. Springer. https://doi.org/10.1007/978-1-4615-1539-5

[44] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.* 18 (2017), 185:1–185:52. http://jmlr.org/papers/v18/16-558.html.

[45] Xiaodong Li, Ke Tang, Mohammmad Nabi Omidvar, Zhenyu Yang, and Kai Qin. 2013. Benchmark functions for the CEC'2013 special session and competition on large-scale global optimization. (01 2013).

[46] Jialin Liu, Antoine Moreau, Mike Preuss, Jeremy Rapin, Baptiste Roziere, Fabien Teytaud, and Olivier Teytaud. 2020. Versatile black-box optimization. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 620–628. https://doi.org/10.1145/3377930.3389838

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

[48] Jorge J. Moré and Stefan M. Wild. 2009. Benchmarking derivative-free optimization algorithms. *SIAM Journal on Optimization* 20, 1 (2009), 172–191.

[49] Antonio J. Nebro, Juan J. Durillo, and Matthieu Vergne. 2015. Redesigning the JMetal multi-objective optimization framework. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'15, Companion Material)*. ACM, 1093–1100. https://doi.org/10.1145/2739482.2768462

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

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

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

[53] Lennart Schäpermeier, Christian Grimme, and Pascal Kerschke. 2021. To boldly show what no one has seen before: A dashboard for visualizing multi-objective landscapes. In *Proc. of Evolutionary Multi-Criterion Optimization (EMO'21) (LNCS)*, Vol. 12654. Springer, 632–644. https://doi.org/10.1007/978-3-030-72062-9_50

[54] Eric O. Scott and Sean Luke. 2019. ECJ at 20: Toward a general metaheuristics toolkit. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'19, Companion Material)*. ACM, 1391–1398. https://doi.org/10.1145/3319619.3326865

[55] Carson Sievert. 2018. *Plotly for R*. https://plotly-r.com.

[56] Emanuel Todorov, Tom Erez, and Yuval Tassa. 2012. MuJoCo: A physics engine for model-based control. *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 5026–5033. https://doi.org/10.1109/IROS.2012.6386109

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

[58] Tea Tusar, Dimo Brockhoff, Nikolaus Hansen, and Anne Auger. 2016. COCO: The bi-objective black box optimization benchmarking (bbob-biobj) test suite. *CoRR* abs/1604.00359 (2016). arXiv:1604.00359 http://arxiv.org/abs/1604.00359.

[59] Sander van Rijn, Hao Wang, Matthijs van Leeuwen, and Thomas Bäck. 2016. Evolving the structure of evolution strategies. In *Proc. of IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 1–8. https://doi.org/10.1109/SSCI.2016.7850138

[60] Konstantinos Varelas, Ouassim Ait ElHara, Dimo Brockhoff, Nikolaus Hansen, Duc Manh Nguyen, Tea Tusar, and Anne Auger. 2020. Benchmarking large-scale continuous optimizers: The BBOB-largescale testbed, a COCO software guide and beyond. *Appl. Soft Comput.* 97, Part (2020), 106737. https://doi.org/10.1016/j.asoc.2020.106737

[61] Diederick Vermetten, Anna V. Kononova, Fabio Caraffini, Hao Wang, and Thomas Bäck. 2021. Is there anisotropy in structural bias? *CoRR* abs/2105.04480 (2021). arXiv:2105.04480 https://arxiv.org/abs/2105.04480.

[62] Diederick Vermetten, Hao Wang, Furong Ye, Carola Doerr, and Thomas Bäck. 2021. IOHanalyzer version 0.1.6.1 + example datasets. (Oct 2021). https://doi.org/10.5281/zenodo.5599719

[63] Stefan Wagner, Gabriel Kronberger, Andreas Beham, Michael Kommenda, Andreas Scheibenpflug, Erik Pitzer, Stefan Vonolfen, Monika Kofler, Stephan Winkler, Viktoria Dorfer, and Michael Affenzeller. 2014. *Advanced Methods and Applications in Computational Intelligence*. Topics in Intelligent Engineering and Informatics, Vol. 6. Springer, Chapter Architecture and Design of the HeuristicLab Optimization Environment, 197–261. http://link.springer.com/chapter/10.1007/978-3-319-01436-4_10.

[64] Hao Wang, Bas van Stein, Michael Emmerich, and Thomas Bäck. 2017. A new acquisition function for Bayesian optimization based on the moment-generating function. In *2017 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2017, Banff, AB, Canada, October 5-8, 2017*. IEEE, 507–512. https://doi.org/10.1109/SMC.2017.8122656

[65] Thomas Weise and Zijun Wu. 2018. 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)*. ACM, 1769–1776.

[66] L. Darrell Whitley, Francisco Chicano, and Brian W. Goldman. 2016. Gray box optimization for Mk landscapes (NK landscapes and MAX-kSAT). *Evol. Comput.* 24, 3 (2016), 491–519. https://doi.org/10.1162/EVCO_a_00184

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

Received June 2021; revised January 2022; accepted January 2022