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

Ye, F.

Chapter 3

The IOHprofiler Benchmarking Software

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

3.1 Overview

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

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

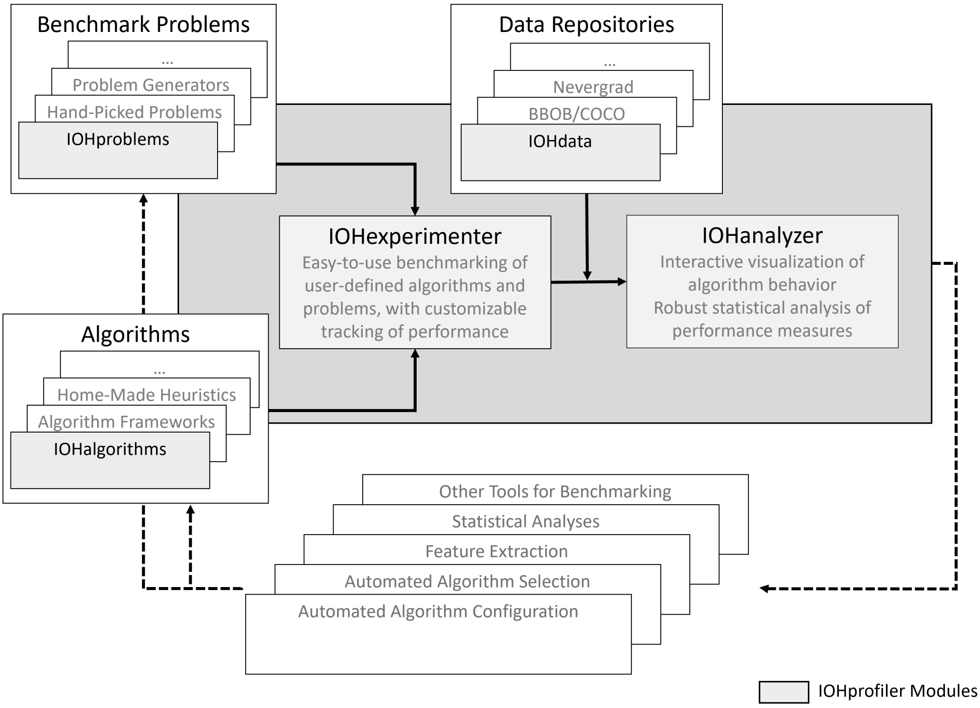


Figure 3.1: Workflow of IOHPROFILER

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

We briefly introduce the modules of IOHPROFILER in the following:

- **IOHproblems:** a collection of benchmark problems. This component currently comprises (1) the PBO suite of pseudo-Boolean optimization problems suggested in [54], (2) the 24 numerical, noise-free BBOB functions from the COCO platform [78], and (3) the W-model problem generator proposed in [160].

- **IOHalgorithms:** a collection of IOHs. For the moment, the algorithms used for the benchmark studies presented in [3, 35, 54] are available. This subsumes textbook algorithms for pseudo-Boolean optimization, an integration to the object-oriented algorithm design framework ParadisEO [24], and the modular algorithm framework for CMA-ES variants originally suggested in [150] and extended in [35]. Further extensions for both combinatorial and numerical solvers are in progress.

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

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

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

3.2 The IOHexperimenter Module

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

3.2.1 Functionalities

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

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

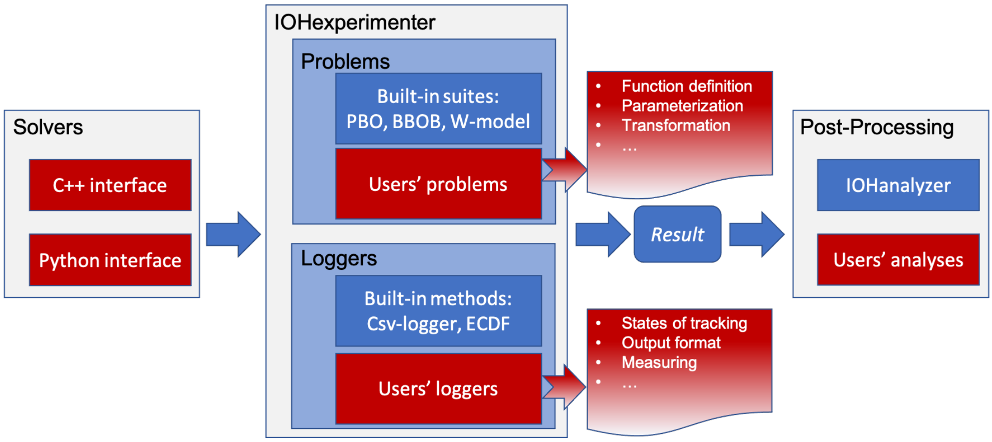


Figure 3.2: Workflow of IOHexperimenter

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

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

3.2.2 Problems

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

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

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

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

3.2.3 Data Logging

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

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

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

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

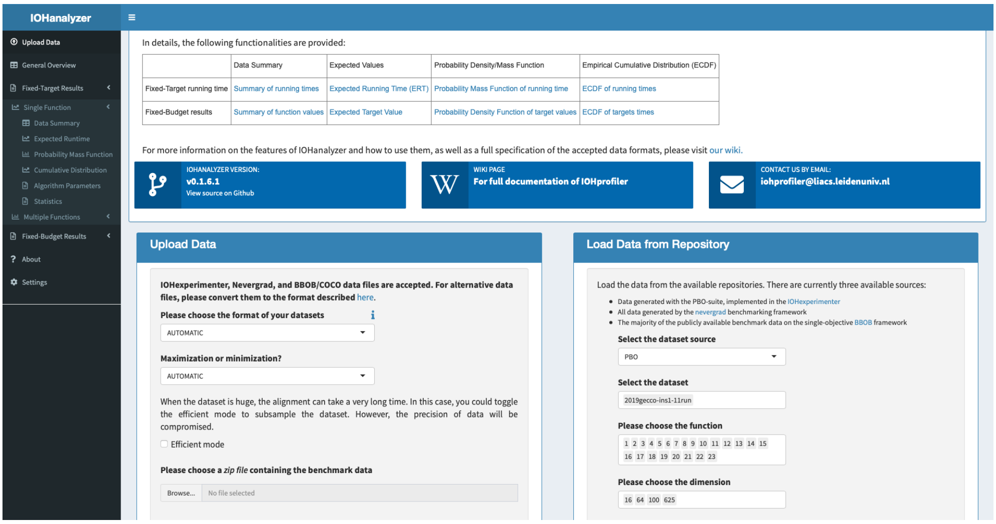


Figure 3.3: The screenshot of the first page of IOHANALYZER.

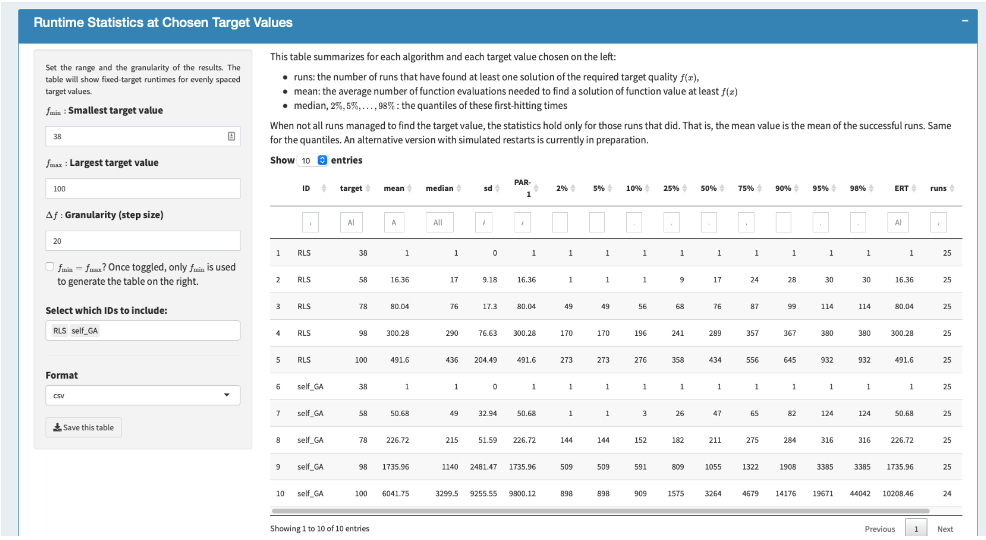
3.2.4 Accessibility

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

3.3 The IOHANALYZER Module

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

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



	ID	target	mean	median	sd	PAR-1	2%	5%	10%	25%	50%	75%	90%	95%	98%	ERT	runs
1	RLS	38	1	1	0	1	1	1	1	1	1	1	1	1	1	1	25
2	RLS	58	16.36	17	9.18	16.36	1	1	1	9	17	24	28	30	30	16.36	25
3	RLS	78	80.04	76	17.3	80.04	49	49	56	68	76	87	99	114	114	80.04	25
4	RLS	98	300.28	290	76.63	300.28	170	170	196	241	289	357	367	380	380	300.28	25
5	RLS	100	491.6	436	204.49	491.6	273	273	276	358	434	556	645	932	932	491.6	25
6	self_GA	38	1	1	0	1	1	1	1	1	1	1	1	1	1	1	25
7	self_GA	58	50.68	49	32.94	50.68	1	1	3	26	47	65	82	124	124	50.68	25
8	self_GA	78	226.72	215	51.59	226.72	144	144	152	182	211	275	284	316	316	226.72	25
9	self_GA	98	1735.96	1140	2481.47	1735.96	509	509	591	809	1055	1322	1908	3385	3385	1735.96	25
10	self_GA	100	6041.75	3299.5	9255.55	9800.12	898	898	909	1575	3264	4679	14176	19671	44042	10208.46	24

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

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

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

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

3.3.1 Accessibility

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

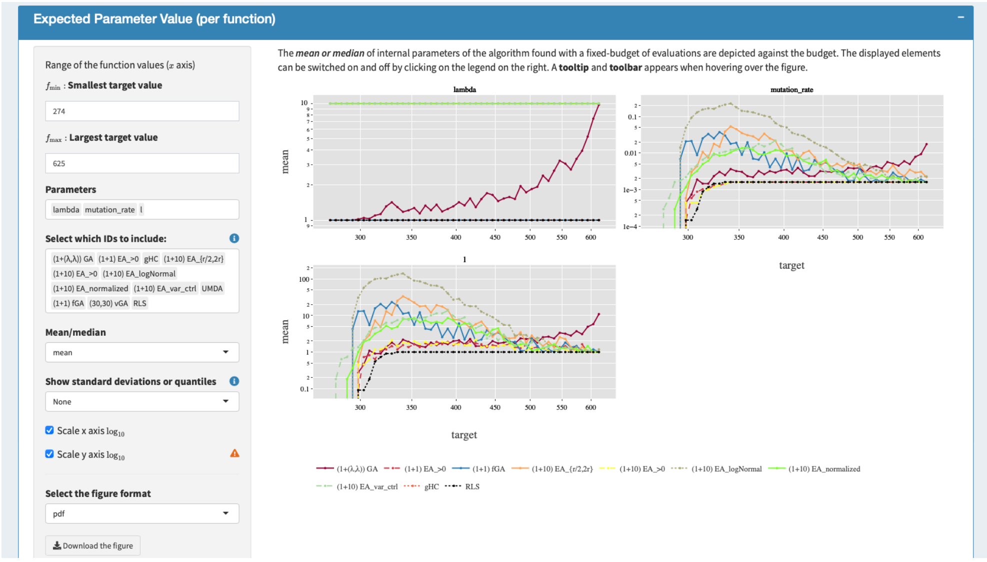


Figure 3.5: The screenshot of the plots of the mean of parameters values for the ten algorithms. The figures plot the mean of internal parameters 'lambda', 'mutation rate', and 'l' of the algorithms for each required target.

details of IOHANALYZER can be found in [156].