IOHprofiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics

Carola Doerr[1], Hao Wang[2], Furong Ye[2], Sander van Rijn[2], Thomas Bäck[2]

[1]Sorbonne Université, CNRS, Laboratoire d'informatique de Paris 6 (LIP6), Paris, France
[2] LIACS, Leiden University, Niels Bohrweg 1, 2333CA Leiden, The Netherlands

October 15, 2018

Abstract

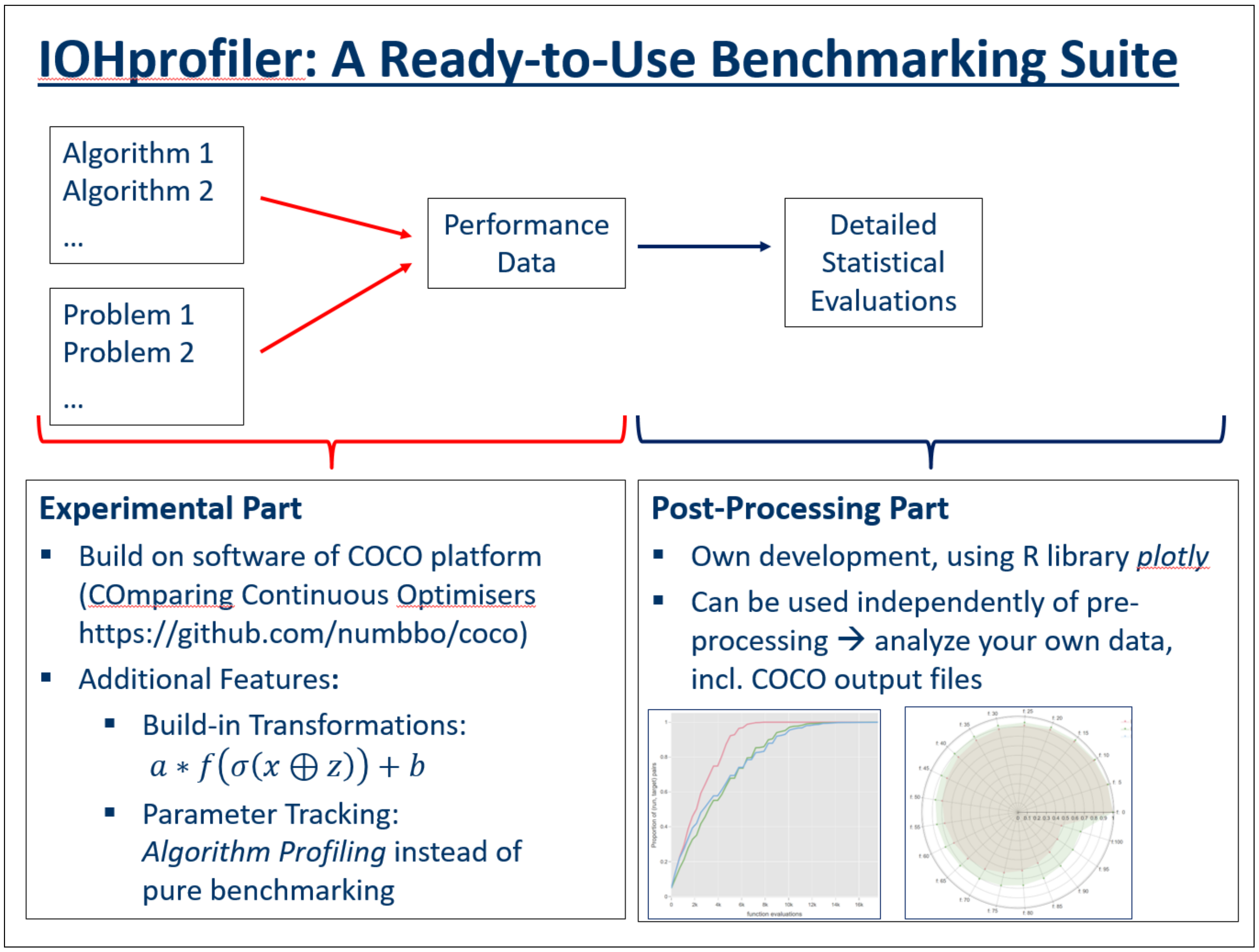
IOHprofiler is a new tool for analyzing and comparing iterative optimization heuristics. Given as input algorithms and problems written in C or Python, it provides as output a statistical evaluation of the algorithms' performance by means of the distribution on the fixed-target running time and the fixed-budget function values. In addition, IOHprofiler also allows to track the evolution of algorithm parameters, making our tool particularly useful for the analysis, comparison, and design of (self-)adaptive algorithms.

IOHprofiler is a ready-to-use software. It consists of two parts: an experimental part, which generates the running time data, and a post-processing part, which produces the summarizing comparisons and statistical evaluations. The experimental part is build on the COCO software, which has been adjusted to cope with optimization problems that are formulated as functions $f : \mathcal{S}^n \to \mathbb{R}$ with $\mathcal{S}$ being a discrete alphabet of integers. The post-processing part is our own work. It can be used as a stand-alone tool for the evaluation of running time data of arbitrary benchmark problems. It accepts as input files not only the output files of IOHprofiler, but also original COCO data files. The post-processing tool is designed for an interactive evaluation, allowing the user to chose the ranges and the precision of the displayed data according to his/her needs.

IOHprofiler is available on GitHub at `https://github.com/IOHprofiler`.

Keywords: Benchmarking, Black-Box Optimization, Discrete Optimization, Evolutionary Computation, Algorithm Profiling

Contents



IOHprofiler: A Ready-to-Use Benchmarking Suite

Algorithm 1
Algorithm 2
…

Problem 1
Problem 2
…

Performance Data

Detailed Statistical Evaluations

Experimental Part

- Build on software of COCO platform (COmparing Continuous Optimisers https://github.com/numbbo/coco)
- Additional Features:
  - Build-in Transformations: $a * f\big(\sigma(x \oplus z)\big) + b$
  - Parameter Tracking: *Algorithm Profiling* instead of pure benchmarking

Post-Processing Part

- Own development, using R library *plotly*
- Can be used independently of pre-processing → analyze your own data, incl. COCO output files

Figure 1: General Layout of IOHprofiler. The post-processing part can be used independently of the experimental part and supports the analysis of COCO output files.

1 Introduction

The ultimate goal of research on optimization problems is the design of efficient problem solvers, which determine high-quality solutions at low cost. Thousands of new algorithms are suggested every year, and the questions of how their performances compare across different optimization problems, and of how far the underlying design ideas can be used to solve different types of optimization problems impose themselves. *Benchmarking* addresses these questions in a principled way, by providing an empirical performance evaluation across different types of optimization problems. The task of designing suitable benchmarking environments is highly non-trivial, and comprises the following questions:

Q1 Which type of optimization algorithms shall be compared?

Q2 Which benchmark problems are most suitable for the comparison?

Q3 Which performance measures should be used?

Q4 Apart from performance, which additional properties of the algorithms should be compared?

In addition to these questions, a number of technical difficulties, such as the various programming languages in which the algorithms and problems are written or the platform on which the benchmark suite is executed also need to be addressed.

The answer to any of the questions Q1-Q4 is quite subjective, and we cannot expect to reach consensus among the scholars and users of different optimization methods. When restricting to certain classes of optimization algorithms, however, it is possible to distill a number of common design principles. In the following, we briefly explain the choices and assumptions made by IOHprofiler.

1.1 Iterative Optimization Heuristics

With respect to Q1, we focus in this work on *iterative optimization heuristics* (IOH). As IOH we classify all algorithms which aim to find optimal solutions by an iterative search. That is, to optimize a problem $f : \mathcal{S} \to \mathbb{R}$, these algorithms proceed in rounds. In each round, the objective values of one or more solution candidates (*search points*) $s^1, \ldots, s^\lambda \in \mathcal{S}$ are evaluated. Their function values $f(s^1), \ldots, f(s^\lambda)$ are used to update the strategy by which the search points for the next round are generated. The search continues until a stopping criterion has been met, e.g., when a solution of a desired quality has been found, a time budget has been reached, or no significant progress could be observed in the last iterations.

The class of IOH subsumes *local* search variants (including first/steepest ascent, variable neighborhood search, Simulated Annealing, Metropolis algorithms, etc.) and *global* search heuristics such as evolutionary algorithms, (Quasi-)Monte Carlo algorithms, swarm intelligence, differential evolution, estimation of distribution algorithms, efficient global optimization, Bayesian optimization, etc.

IOH are particular useful for the optimization of complex, high-dimensional, and large-scale optimization problems. They are—in par with mathematical programming—among the most frequently applied optimization routines in industrial and academic optimization.

1.2 Real-Valued Optimization

Providing an answer to Q2 is arguably the most subjective part of the decision process. We have chosen to take this question aside and to present a very general benchmarking environment which allows an in-depth comparison of IOH for arbitrary real-valued optimization problems.[1] That is, we do not intend in this work to discuss which problems are particularly suitable for the comparison of different IOH. Rather do we offer a tool that can be used to compare performance across functions of the user's choice.

Since the algorithms need to know which search space they should operate on, we focus in the experimental part on problems defined over a discrete alphabet; i.e., we allow functions of the type $f : \mathcal{S}^n \to \mathbb{R}$ with $\mathcal{S} \subset \mathbb{Z}$ being a discrete alphabet of integers. Note that this class comprises in particular the (very broad) class of *pseudo-Boolean optimization problems*, i.e., functions of the type $f : \{0,1\}^n \to \mathbb{R}$.

The post-processing part does not make any assumptions on the type of problem. It can also be used to compare performance across arbitrary optimization problems $f : \mathbb{R}^n \to \mathbb{R}$. It notably accepts in particular data files produced by the original COCO software.

The experimental part of IOHprofiler assume *maximization* as objective. The post-processing part automatically detects from the best-so-far values whether minimization or maximization was the objective of the corresponding experiment.

1.3 Performance Measures

With respect to Q3 we make the important assumption that the running time of the algorithms is dominated to a significant extend by the evaluation of solution candidates, so that all performance measures are based on the *number of function evaluations.* Originally inspired by so-called *black-box optimization* (which, in intuitive terms, assumes that the objective function is not accessed by the algorithm other than through the evaluation of solution candidates), these measures are today established performance indicators also in situations where algorithms *do* have access to (and do make use of) instance data. In several streams of Computer Science the evaluation of a function value is considered a *query* (with the idea that the objective value is queried from an oracle), and the number of function evaluations referred to as *query complexity.* The advantage of evaluation-based performance measures is that they are independent of the machine on which they are executed. However, the user should keep in mind that query complexity can only give an accurate picture of CPU time when the latter is indeed determined to a large extend by the number of evaluated samples.

As standard performance measures IOHprofiler provides information about the distribution of *fixed-target* running times and *fixed-budget* function values. These results include average values and quantiles, but also empirical cumulative distribution function (ECDF) curves, histograms, and empirical probability mass functions. All results are presented in an interactive format that allows the user to specify the granularity, the ranges, and the precision at which the results are displayed. All plots can be stored as png files, the data tables as csv files.

[1]At the moment, we assume only that the problems are static (i.e., f does not change while being optimized) and noise-free (i.e., f is a deterministic function). An extension to dynamic, noisy, and multi-objective optimization problems is under consideration. Users interested in such cases are invited to contact the authors to discuss how to modify IOHprofiler to cover such optimization problems.

1.4 Tracking Additional Information

Addressing Q4, IOHprofiler can also be used to analyze the evolution of various algorithm parameters. The parameters to be tracked are specified by the user. For any of these parameters the user can obtain the same type of statistics as for the running time data. That is, the standard output of IOHprofiler includes in particular statistics (average, median, quantiles,...) about the parameter value at a given point in time (fixed-budget perspective) and at a given function value (fixed-target perspective).

This profiling aspect can be of independent interest, as such information can be very useful for the design of suitable optimization heuristics.

1.5 GitHub page of IOHprofiler

IOHprofiler can be downloaded from the GitHub page `https://github.com/IOHprofiler`.

1.6 Mailing List

Users interested in receiving important updates about IOHprofiler can subscribe to a mailing list at `http://eepurl.com/dBahWb`.

1.7 User Support

The development team of IOHprofiler can be reached by e-mail at Carola.Doerr@mpi-inf.mpg.de[2]

Users can use this contact address to ask for support with the setup of IOHprofiler, but also to suggest new functionalities, different evaluation statistics, etc., or to provide feedback.

1.8 License and Main References

The experimental part of IOHprofiler is build on the COCO software [HAM+16], available at `https://github.com/numbbo/coco`, and includes various modifications to adjust this tool to discrete optimization, to allow for the transformations described in Section 2.2, to choose the granularity by which the data is stored, and to track algorithm parameters.

The post-processing part is original work; the visualization of the results uses the R library *plotly* from `https://plot.ly/`.

IOHprofiler is governed by the BSD 3-Clause license.

1.9 Structure of the Documentation

In the following sections, we summarize the algorithm and problem requirements of IOHprofiler (Section 2), discuss the various options to set the precision of the performance evaluation (Section 3), provide an overview over the standard outputs generated by IOHprofiler (Section 4), and conclude with a number of extensions currently in preparation and planned for future releases of IOHprofiler (Section 5).

[2]This address will be updated with a more generic one in the next version, but for now, please use this address.

Algorithm 1: Blueprint of an iterative optimization heuristic for the optimization of a function $f : \mathcal{S} \to \mathbb{R}$.

```
1 Initialization:
2     $t \leftarrow 0$//iteration counter;
3     $\mathcal{H}(1) = \emptyset$//search history;
4 Optimization: while termination criterion not met do
5     $t \leftarrow t + 1$;
6     Based upon the search history $\mathcal{H}(t)$, choose a probability distribution on $\mathbb{N}$ and sample
        from it $\lambda(t)$; //number of samples to be queried in $t$-th iteration
7     Based upon $\mathcal{H}(t)$ and $\lambda(t)$, choose a probability distribution $D(t)$ on $\mathcal{S}^{\lambda(t)}$; //strategy from
        which next solution candidates are generated
8     From $D(t)$ sample $x^{(t,1)}, \ldots, x^{(t,\lambda(t))} \in \mathcal{S}$ and evaluate their function values
        $f(x^{(t,1)}), \ldots, f(x^{(t,\lambda(t))})$;
9     Build $\mathcal{H}(t+1)$ by selecting which of the samples $(x^{(t,1)}, f(x^{(t,1)})), \ldots, (x^{(t,\lambda(t))}, f(x^{(t,\lambda(t))}))$
        and which of the samples from $\mathcal{H}(t)$ to keep in the search history;
```

A step-by-step manual for the two experimental part of IOHprofiler is provided in Section A of the appendix. The manual for installing and running the post-processing part of IOHprofiler can be found on the aforementioned GitHub page `https://github.com/IOHprofiler`.

2 Summary of Algorithm and Problem Requirements

In this section we summarize the class of algorithms for which IOHprofiler can compare performance and discuss the type of optimization problems which are admissible.

2.1 Iterative Optimization Heuristics

The focus of IOHprofiler is on the performance analysis of *iterative optimization heuristics* (IOH), the class of all algorithms that follow the structure of Algorithm 1. As mentioned in the introduction, this class comprises all sorts of randomized search heuristics, ranging from simple local hill-climbers to complex global search heuristics. The only important feature is that these heuristics do (not only) directly manipulate the problem data, but rather work in a trial-and-error fashion, in which the the function evaluation of search points $s \in \mathcal{S}$ is an integral part of the optimization routine.

Note that in Algorithm 1 all randomized decisions can be replaced by deterministic ones, so that the class of IOH also subsumes deterministic optimizers.

Counting function evaluations. As mentioned above, an important assumption that we make about the algorithms is that their running time is determined (to a large extend) by the time needed to evaluate the solution candidates. We therefore regard in IOHprofiler only performance measures that are based on counting the number of function evaluations; either in a fixed-target or a fixed-budget sense. While the former answers the question how many evaluations are needed to identify a solution of a certain quality., the fixed-budget perspective addresses the complementary questions asking for the quality of the best solutions that can be identified within a given budget of function evaluations.

General-purpose vs. problem-aware algorithms. Our original interest is in comparing iterative optimization heuristic that do not have any a priori knowledge about the (type of) optimization problem that they are facing. That is, we classically work in the aforementioned black-box setting, in which we assume that the algorithm only knows that the problem is a function $f : \mathcal{S}^n \to \mathbb{R}$; i.e., it "knows" in particular the domain (*search space*) $\mathcal{S}^n$ and the co-domain $\mathbb{R}$ (and possibly some bounds on the co-domain). In the classic black-box optimization scenario, the only way to acquire knowledge about the problem instance f is through the evaluation of potential solutions $s \in \mathcal{S}$. However, despite this initial motivation, IOHprofiler is nevertheless also suitable for the comparison and profiling of *problem-aware heuristics,* which have been designed for a particular type of optimization problem.

As we shall explain in the next subsection, IOHprofiler offers to test several *problem instances* that are obtained from a given base problem through transformations of the search points and/or the function values. This allows the user to analyze, for example, if an algorithm is invariant with respect to problem representation and with respect to absolute function values.

2.2 Admissible Benchmark Problems

As mentioned above, the experimental part of IOHprofiler assume maximization as objective, and allows arbitrary functions $f : \mathcal{S}^n \to \mathbb{R}$ with $\mathcal{S}$ being a discrete alphabet of integers. In contrast, the post-processing part of IOHprofiler does not make any further assumption about the type of problems for which the statistics are generated; all real-valued problems are admissible, and the objective can be either minimization or maximmization.

Both parts assume that the optimization problem is *static,* i.e., it does not change over time. We also assume that the evaluations are *noise-free.*

Problem instances. Instead of testing one particular problem f only, the user can choose to run experiments on several problem instances that are obtained from f through a set of transformations. In its most general form, IOHprofiler currently offers to return to the algorithm the values $af(\sigma(x \oplus z)) + b$, where

- a is a **multiplicative shift** of the function value,
- b is a **additive shift** of the function value,
- $\oplus z : \{0,1\}^n \to \{0,1\}^n, (x_1, \ldots, x_n) \mapsto (x_1 + z_1 \mod 2, \ldots, x_n + z_n \mod 2)$ is an **XOR-shift** of the search point,
- $\sigma : \mathcal{S}^n \to \mathcal{S}^n, (x_1, \ldots, x_n) \mapsto (x_{\sigma(1)}, \ldots, x_{\sigma(n)})$ is a **permutation** of the search point. Note here that, in abuse of notation, we identify the permutation $\sigma : [1..n] \to [1..n]$ with the here-defined re-ordering of the bit string.

Note that the "$\oplus z$" transformation is defined only for pseudo-Boolean problems $f : \{0,1\}^n \to \mathbb{R}$, but it can easily be extended to search spaces of the form $\{k_1, \ldots, k_1 + r_1\} \times \ldots \times \{k_n, \ldots, k_n + r_n\}$.[3]

The transformations defined above can be used to test if an algorithm behaves invariant under the proposed modifications. This is an often desired feature of iterative optimization heuristic.

[3]Users interested in such an extension are invited to contact the authors to discuss a possible integration of such a transformation to IOHprofiler.

The user can choose if and which of the transformations are applied to his/her problem, cf. Section A.5 for details. If no transformation is selected, the original $f(x)$-values are returned to the algorithm.

We mention here already that, for convenience of the data analysis, the output files store the following four values:

- $f(\sigma(x \oplus z))$, the non-shifted function value of the search point evaluated in the corresponding iteration,
- the best-so-far $f(\sigma(x \oplus z))$ value,
- $af(\sigma(x \oplus z)) + b$, the shifted function value of the current search point; this is the value that the algorithm has access to, and
- the best-so-far $af(\sigma(x \oplus z)) + b$ value.

3 Precision of the Analysis and Data Format

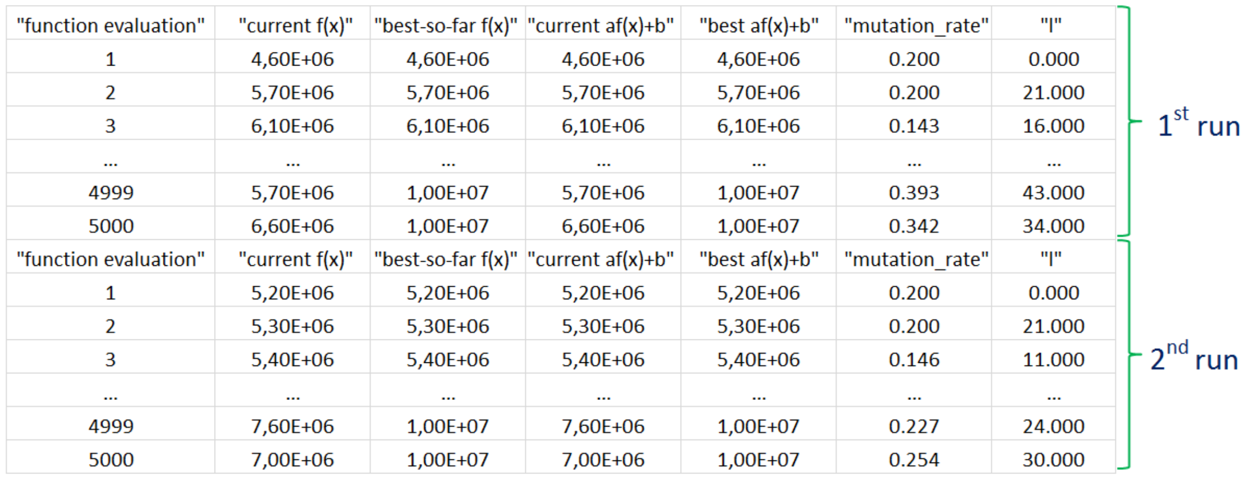
A sound statistical comparison of algorithms requires a substantial amount of data. For the standard output of IOHprofiler, we track for selected evaluations the number of search points evaluated up to this iteration, the (transformed and the original) function value of the solution under evaluation, and the (transformed and original) function value of the best-so-far solution. Apart from this information, IOHprofiler can store additional data, such as the parameters that determine the exact structure of the algorithm. Typical examples for such parameters are the radius at which new solution candidates are sampled (e.g., the *mutation rate*), the number of offspring evaluated in the present iteration, and parameters that determine the selection of the points to keep in the memory. The user selects which algorithm parameters are tracked, cf. Section A for details.

An example for a standard output file be seen in Figure 2.

The interval at which data is stored is chosen by the user. IOHprofiler allows for the following options. A detailed description how to select this granularity is provided in Section A.

- **Complete tracking (*.cdat files):** This data file provides the highest granularity, by storing the above-described information for each function evaluation.
- **Interval tracking (*.idat files):** The user specifies a step size τ. Data is stored for every τ-th function evaluation.
- **Target-based tracking (*.dat files):** These data files store data for each iteration in which the best-so-far function value improves.
- **Time-based tracking (*.tdat files):** In this data file, records are written when the user-specified running time budgets are reached. These running time budgets are evenly spaced in the log-10 scale, taking the form $v10^i \mid i = 0, 1, 2, \ldots$ or $10^{i/t} \mid i = 0, 1, 2, \ldots$. Here, v and t can be set by the user.

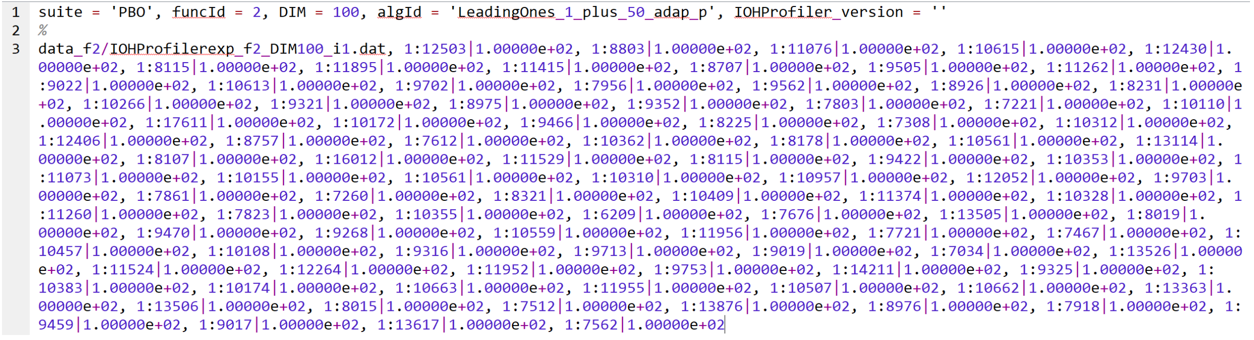
With the current experimental setup, the *.dat data format is always generated, the other three are optional.



"function evaluation"	"current f(x)"	"best-so-far f(x)"	"current af(x)+b"	"best af(x)+b"	"mutation_rate"	"l"
1	4,60E+06	4,60E+06	4,60E+06	4,60E+06	0.200	0.000
2	5,70E+06	5,70E+06	5,70E+06	5,70E+06	0.200	21.000
3	6,10E+06	6,10E+06	6,10E+06	6,10E+06	0.143	16.000
...	...	...	...	...	...	...
4999	5,70E+06	1,00E+07	5,70E+06	1,00E+07	0.393	43.000
5000	6,60E+06	1,00E+07	6,60E+06	1,00E+07	0.342	34.000
"function evaluation"	"current f(x)"	"best-so-far f(x)"	"current af(x)+b"	"best af(x)+b"	"mutation_rate"	"l"
1	5,20E+06	5,20E+06	5,20E+06	5,20E+06	0.200	0.000
2	5,30E+06	5,30E+06	5,30E+06	5,30E+06	0.200	21.000
3	5,40E+06	5,40E+06	5,40E+06	5,40E+06	0.146	11.000
...	...	...	...	...	...	...
4999	7,60E+06	1,00E+07	7,60E+06	1,00E+07	0.227	24.000
5000	7,00E+06	1,00E+07	7,00E+06	1,00E+07	0.254	30.000

Figure 2: The output of the experimental part of IOHprofiler are log-files with information about the current and best-so-far function values and possibly additional algorithm parameters, the mutation rate and parameter "l" in this case. These files are the input for the post-processing part of IOHprofiler.

The structure of the output files follows very closely that of the COCO environment: For each tested algorithm, a separate folder *Algorithm1.zip* is created; the name of this folder can be chosen by the user, see Section A for details. In this folder we find for each tested benchmark function a ".info" file, e.g., `IOHprofiler_f2_i1.info` (where the part "_f2" indicates the tested function and the part "_i1" the smallest index of the tested instances of this benchmark problem).



```
suite = 'PBO', funcId = 2, DIM = 100, algId = 'LeadingOnes_1_plus_50_adap_p', IOHProfiler_version = ''
%
data_f2/IOHProfilerexp_f2_DIM100_i1.dat, 1:12503|1.00000e+02, 1:8803|1.00000e+02, 1:11076|1.00000e+02, 1:10615|1.00000e+02, 1:12430|1.
00000e+02, 1:8115|1.00000e+02, 1:11895|1.00000e+02, 1:11415|1.00000e+02, 1:8707|1.00000e+02, 1:9505|1.00000e+02, 1:11262|1.00000e+02, 1
:9022|1.00000e+02, 1:10613|1.00000e+02, 1:9702|1.00000e+02, 1:7956|1.00000e+02, 1:9562|1.00000e+02, 1:8926|1.00000e+02, 1:8231|1.00000e
+02, 1:10266|1.00000e+02, 1:9321|1.00000e+02, 1:8975|1.00000e+02, 1:9352|1.00000e+02, 1:7803|1.00000e+02, 1:7221|1.00000e+02, 1:10110|1
.00000e+02, 1:17611|1.00000e+02, 1:10172|1.00000e+02, 1:9466|1.00000e+02, 1:8225|1.00000e+02, 1:7308|1.00000e+02, 1:10312|1.00000e+02,
1:12406|1.00000e+02, 1:8757|1.00000e+02, 1:7612|1.00000e+02, 1:10362|1.00000e+02, 1:8178|1.00000e+02, 1:10561|1.00000e+02, 1:13114|1.
00000e+02, 1:8107|1.00000e+02, 1:16012|1.00000e+02, 1:11529|1.00000e+02, 1:8115|1.00000e+02, 1:9422|1.00000e+02, 1:10353|1.00000e+02, 1
:11073|1.00000e+02, 1:10155|1.00000e+02, 1:10561|1.00000e+02, 1:10310|1.00000e+02, 1:10957|1.00000e+02, 1:12052|1.00000e+02, 1:9703|1.
00000e+02, 1:7861|1.00000e+02, 1:7260|1.00000e+02, 1:8321|1.00000e+02, 1:10409|1.00000e+02, 1:11374|1.00000e+02, 1:10328|1.00000e+02, 1
:11260|1.00000e+02, 1:7823|1.00000e+02, 1:10355|1.00000e+02, 1:6209|1.00000e+02, 1:7676|1.00000e+02, 1:13505|1.00000e+02, 1:8019|1.
00000e+02, 1:9470|1.00000e+02, 1:9268|1.00000e+02, 1:10559|1.00000e+02, 1:11956|1.00000e+02, 1:7721|1.00000e+02, 1:7467|1.00000e+02, 1:
10457|1.00000e+02, 1:10108|1.00000e+02, 1:9316|1.00000e+02, 1:9713|1.00000e+02, 1:9019|1.00000e+02, 1:7034|1.00000e+02, 1:13526|1.00000
e+02, 1:11524|1.00000e+02, 1:12264|1.00000e+02, 1:11952|1.00000e+02, 1:9753|1.00000e+02, 1:14211|1.00000e+02, 1:9325|1.00000e+02, 1:
10383|1.00000e+02, 1:10174|1.00000e+02, 1:10663|1.00000e+02, 1:11955|1.00000e+02, 1:10507|1.00000e+02, 1:10662|1.00000e+02, 1:13363|1.
00000e+02, 1:13506|1.00000e+02, 1:8015|1.00000e+02, 1:7512|1.00000e+02, 1:13876|1.00000e+02, 1:8976|1.00000e+02, 1:7918|1.00000e+02, 1:
9459|1.00000e+02, 1:9017|1.00000e+02, 1:13617|1.00000e+02, 1:7562|1.00000e+02
```

This file contains the following information:

- in the first line we store the name of the benchmark suite (a *suite* is a collection of benchmark functions), the ID of the benchmark function, the dimension for which experiments have been conducted, the name of the algorithm, and information about the version of the IOHprofiler.
- the second line is an empty line containing only the symbol %. The user can use this line to record some information about the algorithm or the experiment. This information can be specified in the configuration file.

- the subsequent lines specify the path where the actual runtime data is located (in the example of the screenshot above, this is the file `IOHprofilerexp_f2_DIM100_i1.dat` in folder *data_f2*. Thereafter, it is recorded for each run (100 in the example) how many lines of data points have been stored, along with the final best-so-far value of the respective run. In the example above, 12 503 data points have been stored for the first run, and the best found solution had a function value of 100.

When several dimensions have been tested, the corresponding information above is written into the `IOHprofiler_f2_i1.info` one below the other.

The performance data is stored in sub-folders; one subfolder for each tested benchmark function. In these sub-folders the different data files specified above can be found, generic names for these files are `IOHprofiler_f2_DIM1000_i1.dat` for a *.dat file containing performance data from an experiment on the 1 000-dimensional function f2. Results for different dimensions are stored in the same folder.

4 Supported Performance Analyses

We recall that the objectives of IOHprofiler tool are two-fold. On the one hand, it aims to contribute to a statistically sound comparison of iterative heuristics for pseudo-Boolean optimization problems. This is the **benchmarking aspect** of IOHprofiler. An important motivation for algorithm benchmarking is the desire to generate insights that can be used for the design of efficient optimizers. To this end, it is not only important to understand well how the algorithms perform on different types of optimization problems; not less important is to analyze how the states of the algorithm itself evolve over time. To address this question, IOHprofiler allows to track the evolution of the key parameters that determine the algorithm. The evaluation of these parameters covers the **profiling aspect** of IOHprofiler.

We describe in this section the standard outputs that IOHprofiler generates. The results are grouped into three categories:

(a) **Fixed-Target Results,** described in Section 4.3: This section provides summarizing statistics covering the fixed-target perspective of performance evaluation. That is, the results in this section mainly address the question how much "time" (i.e., function evaluations) is needed to obtain a solution of a desired target quality.

(b) **Fixed-Budget Results,** Section 4.4: Covering the fixed-budget perspective, these outputs present statistics for the quality of the search points obtained within a given budget of function evaluations. That is, the results in this section mainly address the question how good the search points are that a user can expect to see within a given time frame (where "time" refers again to the number of evaluations).

(c) **Algorithm Parameters,** Section 4.5: This section provides details about the evolution of the algorithm parameters that the user specified to be tracked during the experimental part.

Performance Measure in Preparation: IOHprofiler currently does not perform statistical tests, nor comparing results over various problem dimensions, nor performance aggregation over several benchmark problems. These measures are currently in preparation, and will be made available shortly. Note, however, that in addition to the summarizing statistics detailed in the next subsections, IOHprofiler provides for each section the option to store sorted raw data, which may be convenient for computing additional performance measures, statistical tests, etc. Users interested in a discussion which additional performance measures to include as a standard output, are asked to get in touch with the IOHprofiler developers.

4.1 Notation and Basic Terminology

Before we present the various outputs, we briefly discuss the terminology used in the remainder of this section. We recall that we assume *maximization* as objective.

For every algorithm A and every function f, we denote by

- $T(A, f, v, i)$ the number of function evaluations that have been performed in run i until and including the first evaluation of a search point x satisfying $f(x) \geq v$; i.e., the "time" needed by algorithm A in run i to reach for function f a solution x with *target value* at least v.
- $V(A, f, t, i) := \max\{f(x^{(j)}) \mid j \in \{1, \ldots, t\}\}$, the function value of the best among the first t evaluated solution candidates in run i. The variable t is referred to as *budget*.

The values $T(A, f, v, i)$ and $V(A, f, t, i)$ are aggregated over the r independent runs to *fixed-target running times* and *fixed-budget function values*, respectively. Note that $T(A, f, v, \cdot)$ and $V(A, f, t, \cdot)$ are random variables, and $T(A, f, v, i)$ and $V(A, f, t, i)$ samples thereof.

Among the most classic performance measures are the *mean* values $\mathbb{E}[V(A, f, t)]$ and $\mathbb{E}[T(A, f, v)]$ of the distributions $V(A, f, t, \cdot)$ and $T(A, f, v, \cdot)$. We approximate these expected values by the empirical averages over all r independent runs, and abbreviate:

- $\widehat{\mathbb{E}}[T(A, f, v)] := \sum_{i=1}^{r} T(A, f, v, i)/r$, the average budget needed to find a solution of quality at least v.
- $\widehat{\mathbb{E}}[V(A, f, t)] := \sum_{i=1}^{r} V(A, f, t, i)/r$, the average quality of the best solution found within a budget of t function evaluations.

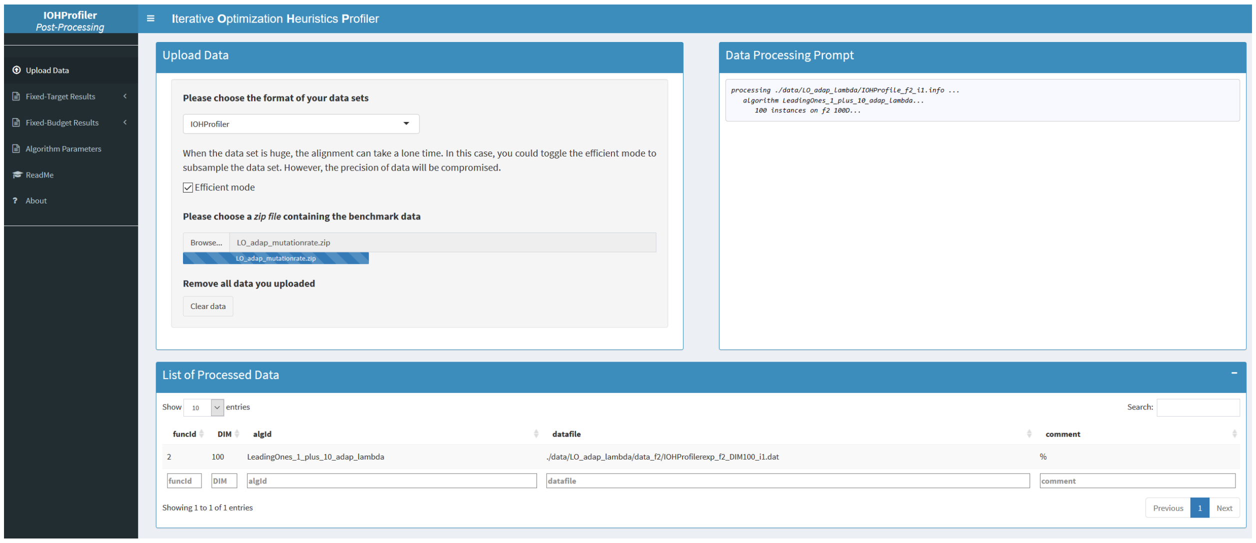
When the variables $T(A, f, v, \cdot)$ and $V(A, f, t, , \cdot)$ are not concentrated and/or not symmetric, average values can be misleading. IOHprofiler therefore also computes different quantiles of these distributions. To this end, the values $V(A, f, t, 1), \ldots, V(A, f, t, r)$ [and $T(A, f, v, 1), \ldots, T(A, f, v, r)$, respectively] are sorted in non-decreasing order. We denote by $V(A, f, t, i : r)$ [and $T(A, f, v, i : r)$, resp.] the i-th element of the resulting sequence. For any $1 \leq p \leq 100$, the *p-th percentile* of the distributions are estimated as

$$\widehat{\mathbb{Q}}[T(A, f, v), p] := T(A, f, v, \lfloor pr/100 \rfloor), \text{ and } \widehat{\mathbb{Q}}[V(A, f, t), p] := V(A, f, t, \lfloor pr/100 \rfloor),$$

respectively.

In addition to these values, it is also interesting to accumulate the running time data into ECDF curves. ECDF stands for *empirical cumulative distribution function.* Again we have to distinguish between the fixed-target and the fixed-budget perspective:

- In the fixed-target perspective, an ECDF curve requires to select a set $\{v_1, \ldots, v_d\}$ of target values. The corresponding ECDF curve shows for each budget t the fraction $\{(i, v_j) \mid 1 \leq i \leq r, 1 \leq j \leq d\}/(rd)$ of the (run, target value) pairs (i, v_j) that satisfy that $V(A, f, t, i) \geq v_j$. That is, $\widehat{F}(t) = \frac{1}{dr} \sum_{j=1}^{d} \sum_{i=1}^{r} \mathbf{1}_{V(A,f,t,i) \geq v_j}$, where $\mathbf{1}_{\mathcal{C}}$ denotes the indicator variable, which is one when the condition $\mathcal{C}$ is satisfied.



Upload Data

Please choose the format of your data sets

IOHProfiler

When the data set is huge, the alignment can take a lone time. In this case, you could toggle the efficient mode to subsample the data set. However, the precision of data will be compromised.

[✓] Efficient mode

Please choose a *zip file* containing the benchmark data

Browse... LO_adap_mutationrate.zip

Remove all data you uploaded

Clear data

Data Processing Prompt

```
processing ./data/LO_adap_lambda/IOHProfile_f2_i1.info ...
   algorithm LeadingOnes_1_plus_10_adap_lambda...
      100 instances on f2 100D...
```

List of Processed Data

Show 10 entries Search:

funcId	DIM	algId	datafile	comment
2	100	LeadingOnes_1_plus_10_adap_lambda	./data/LO_adap_lambda/data_f2/IOHProfilerexp_f2_DIM100_i1.dat	%

Showing 1 to 1 of 1 entries

Previous 1 Next

Figure 3: In the *upload* tab, the user provides the links the performance data.

- Likewise, in the fixed-budget perspective, the user selects a set $\{t_1, \ldots, t_d\}$ of budgets. The corresponding ECDF curve shows for each target value v the fraction $\{(i, t_j) \mid 1 \leq i \leq r, 1 \leq j \leq d\}/(rd)$ of the (run, budget) pairs that satisfy that $T(A, f, v, i) \leq t_j$.

4.2 Linking the Data Files

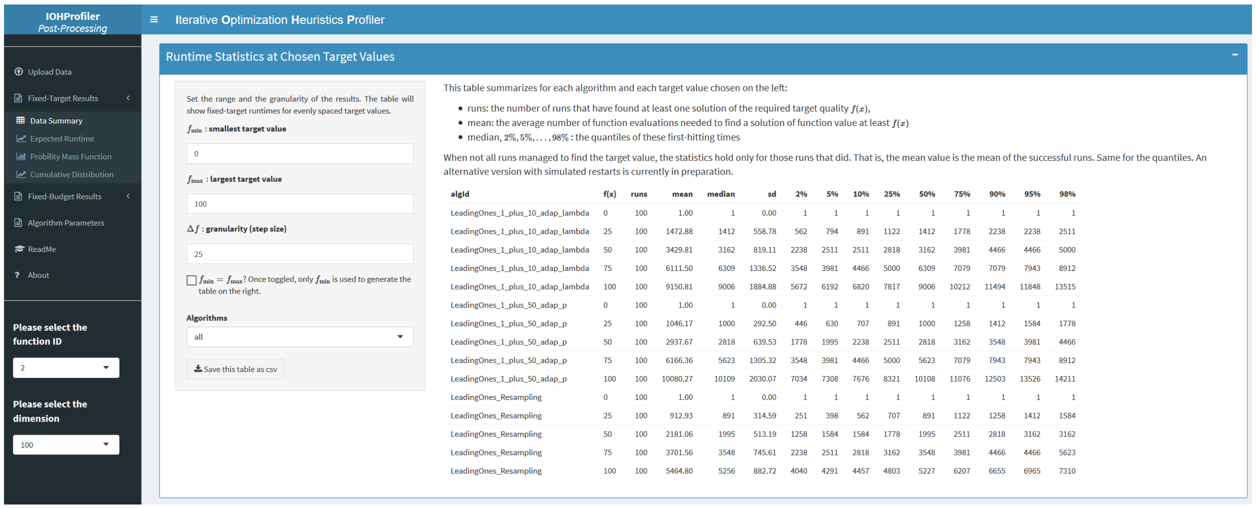
In the **upload** tab of the post-processing part, the user provides the links to the folders containing the performance data that shall be analyzed. Figure 3 shows this tab. The user can select whether his/her data is in the format of the IOHPROFILER experimentation part or in the COCO format. Toggling the efficient mode results in a faster computation of the results, at the cost of precision. After choosing the data file to be uploaded to the tool, the *Data Processing Promt* on the right records which data has been identified; in the example 100 runs for the 100-dimensional version of function *f2*. The list of processed data at the bottom summarizes this information in table format.

4.3 Fixed-Target Results

The fixed-target section has four different subsections ("*tabs*"):

- 'Data Summary': this tab provides tables with the fixed-target running time statistics, as well as tables with the sorted raw values $T(A, f, v, i)$ of the individual runs. See Section 4.3.1 for details.

- 'Expected Runtime': an interactive plot illustrates the fixed-target running times. The user can choose to display mean and/or median values along with the standard deviations. The user also selects the algorithms which are displayed, the range for which the fixed-target statistics are computed, and whether or not the axes are scaled logarithmically. Confer Section 4.3.2 for details.



Please select the function ID: 2

Please select the dimension: 100

Runtime Statistics at Chosen Target Values

Set the range and the granularity of the results. The table will show fixed-target runtimes for evenly spaced target values.

$f_{\min}$: smallest target value: 0

$f_{\max}$: largest target value: 100

Δf : granularity (step size): 25

☐ $f_{\min} = f_{\max}$? Once toggled, only $f_{\min}$ is used to generate the table on the right.

Algorithms: all

Save this table as csv

This table summarizes for each algorithm and each target value chosen on the left:

- runs: the number of runs that have found at least one solution of the required target quality $f(x)$,
- mean: the average number of function evaluations needed to find a solution of function value at least $f(x)$
- median, $2\%, 5\%, \ldots, 98\%$: the quantiles of these first-hitting times

When not all runs managed to find the target value, the statistics hold only for those runs that did. That is, the mean value is the mean of the successful runs. Same for the quantiles. An alternative version with simulated restarts is currently in preparation.

algId	f(x)	runs	mean	median	sd	2%	5%	10%	25%	50%	75%	90%	95%	98%
LeadingOnes_1_plus_10_adap_lambda	0	100	1.00	1	0.00	1	1	1	1	1	1	1	1	1
LeadingOnes_1_plus_10_adap_lambda	25	100	1472.88	1412	558.78	562	794	891	1122	1412	1778	2238	2238	2511
LeadingOnes_1_plus_10_adap_lambda	50	100	3429.81	3162	819.11	2238	2511	2511	2818	3162	3981	4466	4466	5000
LeadingOnes_1_plus_10_adap_lambda	75	100	6111.50	6309	1336.52	3548	3981	4466	5000	6309	7079	7079	7943	8912
LeadingOnes_1_plus_10_adap_lambda	100	100	9150.81	9006	1884.88	5672	6192	6820	7817	9006	10212	11494	11848	13515
LeadingOnes_1_plus_50_adap_p	0	100	1.00	1	0.00	1	1	1	1	1	1	1	1	1
LeadingOnes_1_plus_50_adap_p	25	100	1046.17	1000	292.50	446	630	707	891	1000	1258	1412	1584	1778
LeadingOnes_1_plus_50_adap_p	50	100	2937.67	2818	639.53	1778	1995	2238	2511	2818	3162	3548	3981	4466
LeadingOnes_1_plus_50_adap_p	75	100	6166.36	5623	1305.32	3548	3981	4466	5000	5623	7079	7943	7943	8912
LeadingOnes_1_plus_50_adap_p	100	100	10080.27	10109	2030.07	7034	7308	7676	8321	10108	11076	12503	13526	14211
LeadingOnes_Resampling	0	100	1.00	1	0.00	1	1	1	1	1	1	1	1	1
LeadingOnes_Resampling	25	100	912.93	891	314.59	251	398	562	707	891	1122	1258	1412	1584
LeadingOnes_Resampling	50	100	2181.06	1995	513.19	1258	1584	1584	1778	1995	2511	2818	3162	3162
LeadingOnes_Resampling	75	100	3701.56	3548	745.61	2238	2511	2818	3162	3548	3981	4466	4466	5623
LeadingOnes_Resampling	100	100	5464.80	5256	882.72	4040	4291	4457	4803	5227	6207	6655	6965	7310

Figure 4: ‘Fixed-Target Results: Data Summary’: running time statistics at chosen target values.

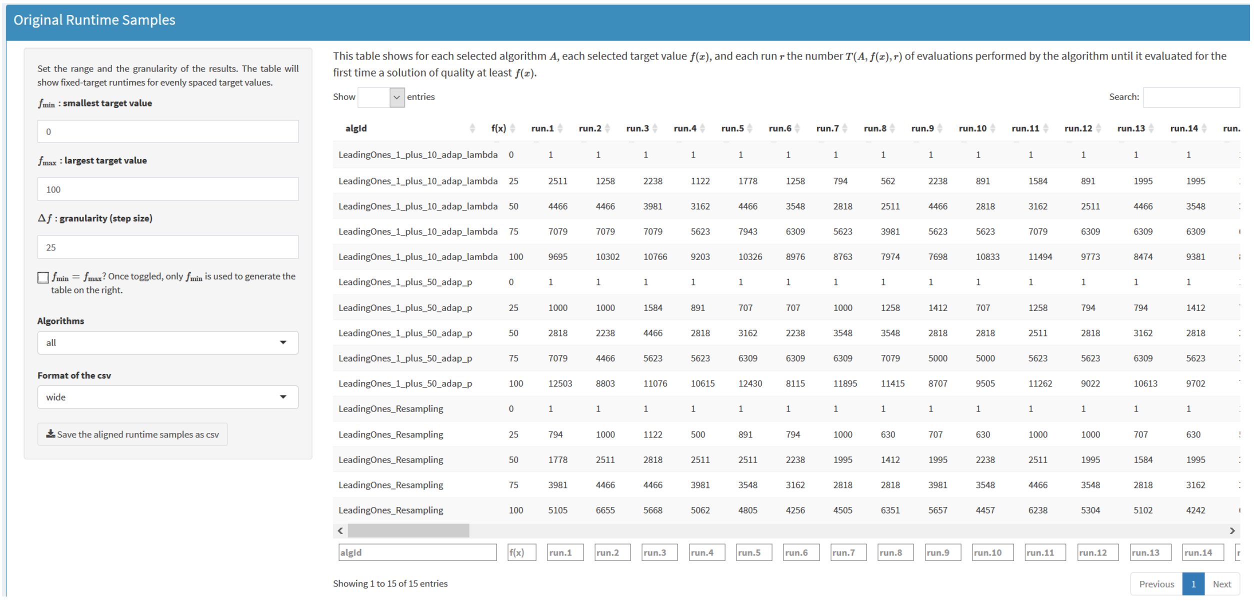
- ‘Probability Mass Function’: interactive histograms show the distribution of the values $T(A, f, v, i)$ for target values v selected by the user. Furthermore, an approximation for the empirical probability mass function is provided in this tab, cf. Section 4.3.3.
- ‘Cumulative Distribution’: ECDF curves are computed for target values specified by the user. A spider-plot shows the area under the ECDF curves for different target values. In addition, ECDF curves for individual target values can be shown, cf. Section 4.3.4.

4.3.1 Fixed-Target: ‘Data Summary’

Figure 4 shows the upper part of the ‘Data Summary’ tab. 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. More precisely, for each (algorithm A, target value v) pair the table provides

- runs: the number of runs of algorithm A in which at least one solution x satisfying $f(x) > v$ has been found,
- mean: $\widehat{\mathbb{E}}[T(A, f, v)]$, the average number of function evaluations needed to find a solution of function value at least v,
- median, 2%, 5%, …: the quantiles $\widehat{\mathbb{Q}}[T(A, f, v), p]$ of these first-hitting times.

The sorted raw data used to compute the summarizing statistics can be downloaded from the `Original Runtime Samples` section on the bottom of the ‘Data Summary’ tab. For each target value v selected in the options box on the left, the table shows, for each algorithm A and each run i, the number $T(A, f, v, i)$ of evaluations performed by the algorithm until it evaluated for the first time a solution x of quality at least v. The user can choose between a vertical and an horizontal alignment of the data; Figure 5 shows the wide variant. These tables can be stored as csv files.


Original Runtime Samples

Set the range and the granularity of the results. The table will show fixed-target runtimes for evenly spaced target values.

$f_{\min}$: smallest target value: 0

$f_{\max}$: largest target value: 100

Δf : granularity (step size): 25

☐ $f_{\min} = f_{\max}$? Once toggled, only $f_{\min}$ is used to generate the table on the right.

Algorithms: all

Format of the csv: wide

Save the aligned runtime samples as csv

This table shows for each selected algorithm A, each selected target value $f(x)$, and each run r the number $T(A, f(x), r)$ of evaluations performed by the algorithm until it evaluated for the first time a solution of quality at least $f(x)$.

Show entries

Search:

algId	f(x)	run.1	run.2	run.3	run.4	run.5	run.6	run.7	run.8	run.9	run.10	run.11	run.12	run.13	run.14	run.
LeadingOnes_1_plus_10_adap_lambda	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
LeadingOnes_1_plus_10_adap_lambda	25	2511	1258	2238	1122	1778	1258	794	562	2238	891	1584	891	1995	1995	
LeadingOnes_1_plus_10_adap_lambda	50	4466	4466	3981	3162	4466	3548	2818	2511	4466	2818	3162	2511	4466	3548	
LeadingOnes_1_plus_10_adap_lambda	75	7079	7079	7079	5623	7943	6309	5623	3981	5623	5623	7079	6309	6309	6309	
LeadingOnes_1_plus_10_adap_lambda	100	9695	10302	10766	9203	10326	8976	8763	7974	7698	10833	11494	9773	8474	9381	
LeadingOnes_1_plus_50_adap_p	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
LeadingOnes_1_plus_50_adap_p	25	1000	1000	1584	891	707	707	1000	1258	1412	707	1258	794	794	1412	
LeadingOnes_1_plus_50_adap_p	50	2818	2238	4466	2818	3162	2238	3548	3548	2818	2818	2511	2818	3162	2818	
LeadingOnes_1_plus_50_adap_p	75	7079	4466	5623	5623	6309	6309	6309	7079	5000	5000	5623	5623	6309	5623	
LeadingOnes_1_plus_50_adap_p	100	12503	8803	11076	10615	12430	8115	11895	11415	8707	9505	11262	9022	10613	9702	
LeadingOnes_Resampling	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
LeadingOnes_Resampling	25	794	1000	1122	500	891	794	1000	630	707	630	1000	1000	707	630	
LeadingOnes_Resampling	50	1778	2511	2818	2511	2511	2238	1995	1412	1995	2238	2511	1995	1584	1995	
LeadingOnes_Resampling	75	3981	4466	4466	3981	3548	3162	2818	2818	3981	3548	4466	3548	2818	3162	
LeadingOnes_Resampling	100	5105	6655	5668	5062	4805	4256	4505	6351	5657	4457	6238	5304	5102	4242	

Showing 1 to 15 of 15 entries

Previous 1 Next

Figure 5: 'Fixed-Target Results: Data Summary': sorted first hitting times $T(A, f, v, i)$.

4.3.2 Fixed-Target: 'Expected Runtime'

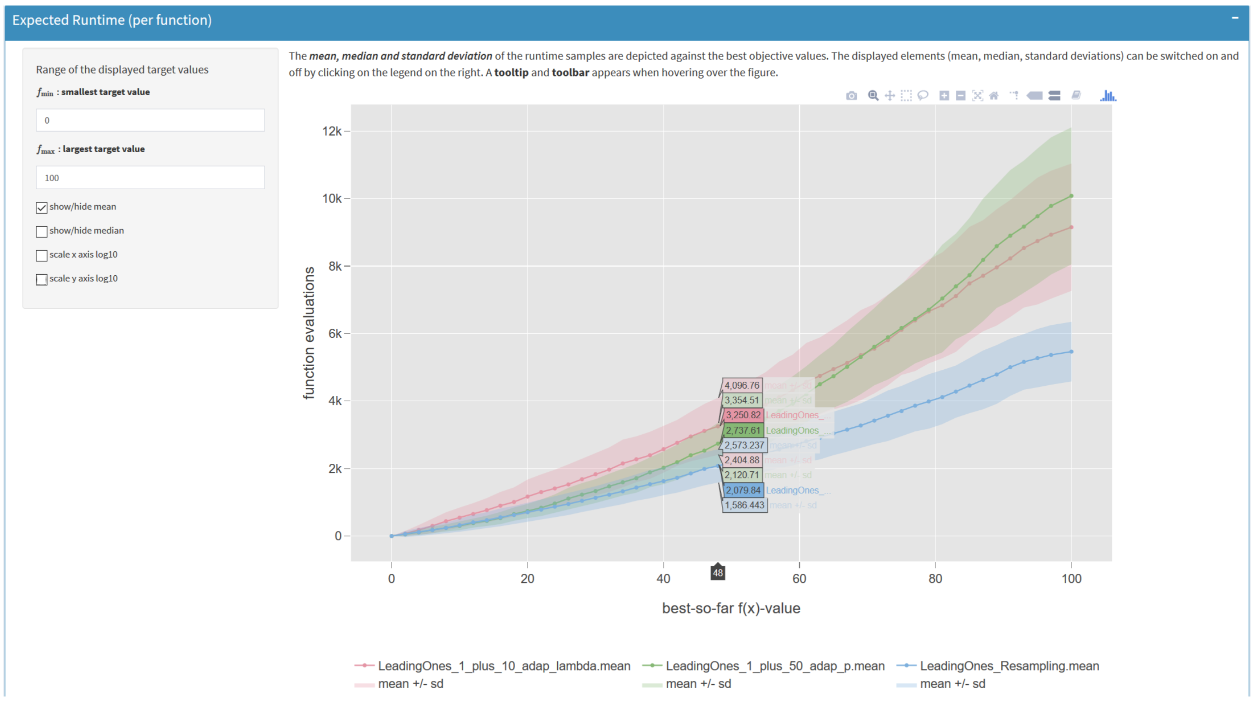
The average, median, and standard deviations of the running time samples are depicted against the best-so-far objective values. The displayed elements can be switched on and off by clicking on the legend on the right. This also allows the user to select the algorithms for which the results are shown. Some display options, including the option to store the picture as a png file, appear when moving the mouse over the picture. Detailed numbers appear when hovering the mouse over the curves, cf. Figure 6.

4.3.3 Fixed-Target: 'Probability Mass Function'

The third tab of the fixed-target section provides, for a target value selected by the user, histograms of the running time samples and an approximation of the probability mass function.

For a selected target value v the histogram, displayed in Figure 7, shows for each range $[t, t+1)$ the number of runs i satisfying $t \leq T(A, f, v, i) < t+1$. The bin sizes $[t, t+1)$ are chosen automatically according to the so-called Freedman–Diaconis rule, by which the bin size is set to $(\widehat{\mathbb{Q}}[T(A, f, v), 75] - \widehat{\mathbb{Q}}[T(A, f, v), 25]) / \sqrt[3]{r}$. Note that the displayed algorithms can be selected again by clicking on the legend on the right. The user has two options: an overlayed display, where all algorithms are displayed in the same plot, or a separated one, in which each algorithm is displayed in an individual chart.

Finally, the estimation of the probability mass function (cf. Figure 8) may be useful to get a better idea of how the values $T(A, f, v, i)$ are distributed for a given target value v. The user can opt to show all individual values $T(A, f, v, 1), \ldots, T(A, f, v, r)$, or only the approximated probability mass function. Note, however, that the latter is just an approximation, which estimates the


Expected Runtime (per function)

Range of the displayed target values

$f_{\min}$: **smallest target value**

0

$f_{\max}$: **largest target value**

100

[✓] show/hide mean

[ ] show/hide median

[ ] scale x axis log10

[ ] scale y axis log10

The ***mean, median and standard deviation*** of the runtime samples are depicted against the best objective values. The displayed elements (mean, median, standard deviations) can be switched on and off by clicking on the legend on the right. A **tooltip** and **toolbar** appears when hovering over the figure.

Figure 6: 'Fixed-Target Results: Expected Runtime': Average and median fixed-target running times.

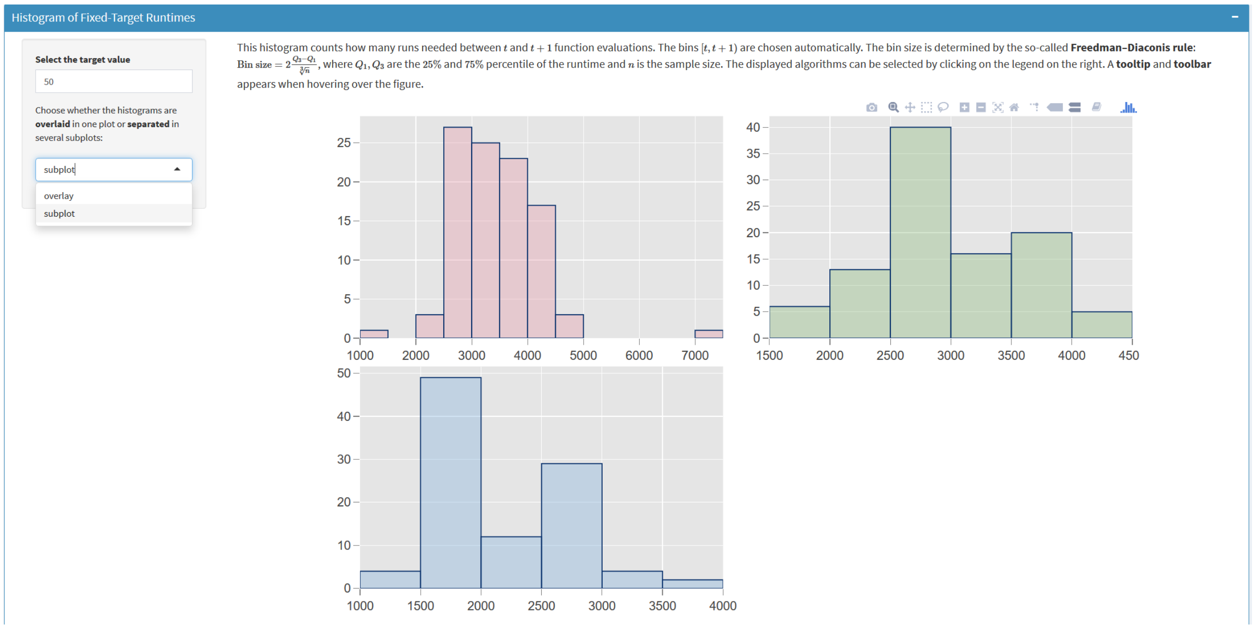
probability mass function by treating the running times as continuous variables.[4] Note also that in the example of Figure 8 many data points seem aligned, this might be caused by turning (in the upload tab) the "efficient mode" on, in which the raw data set is trimmed.

4.3.4 Fixed-Target: 'Cumulative Distribution'

This tab provides ECDF curves and information about the area under the ECDF curves. For the aggregated ECDF curves, the user selects a range of the target values and the steps at which the data is displayed. Selecting as $f_{\min} = 0$, $f_{\max} = 100$, and $\Delta = 10$ as in the example of Figure 9, the ECDF curves for target values $0, 10, 20, 30, \ldots, 100$ are computed. When r independent runs have been performed, the ECDF curves thus show the fraction of all $11r$ (run, target value) pairs $\{(i, 10j) \mid 1 \leq i \leq r, 0 \leq j \leq 10\}$ that satisfy for a given t that $T(A, f, 10j, i) \leq t$. In the example of Figure 9, for Algorithm *LeadingOnes_resampling* (blue curve) this is the case for around 75% of the pairs after $t = 4\,000$ function evaluations. For Algorithm *LeadingOnes_1_plus_50_adap_p* (green curve) the fraction is 58%.

An ideal algorithm would sample the maximal function value $f_{\max}$ in the first step. This algorithm would have a 100% score for all budgets t. In practice, such an algorithm does not exist, but it serves as a theoretical upper bound and we use the area under its curve to normalize the areas under the curves of the tested algorithms. The radar-like plot in the 'Area under the ECDF'

[4]Strictly speaking, this method gives imprecise estimations when there are many duplicated values. Improvements are planned for the future version.


Histogram of Fixed-Target Runtimes

Select the target value

50

Choose whether the histograms are **overlaid** in one plot or **separated** in several subplots:

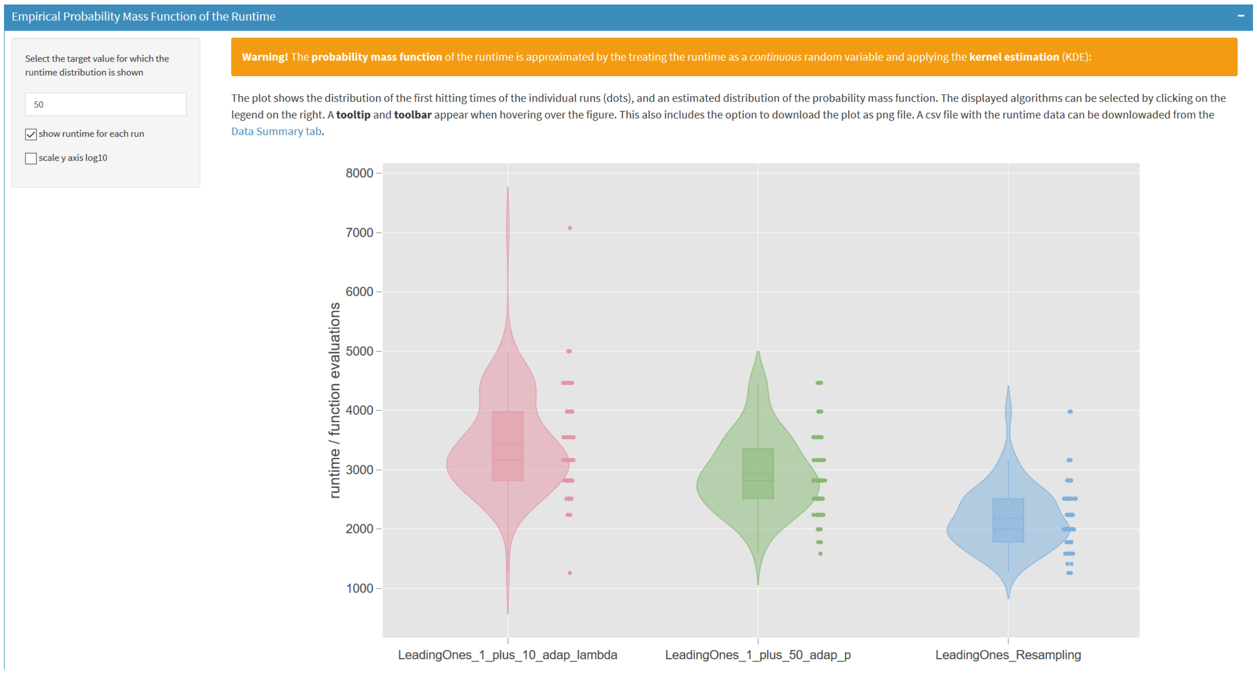
subplot

overlay

subplot

This histogram counts how many runs needed between t and $t+1$ function evaluations. The bins $[t, t+1)$ are chosen automatically. The bin size is determined by the so-called **Freedman–Diaconis rule**: $\text{Bin size} = 2\frac{Q_3 - Q_1}{\sqrt[3]{n}}$, where Q_1, Q_3 are the 25% and 75% percentile of the runtime and n is the sample size. The displayed algorithms can be selected by clicking on the legend on the right. A **tooltip** and **toolbar** appears when hovering over the figure.

Figure 7: 'Fixed-Target Results: Probability Mass Function': Histograms for the fixed-target running times $T(A, f, v, i)$.


Empirical Probability Mass Function of the Runtime

Select the target value for which the runtime distribution is shown

50

[✓] show runtime for each run

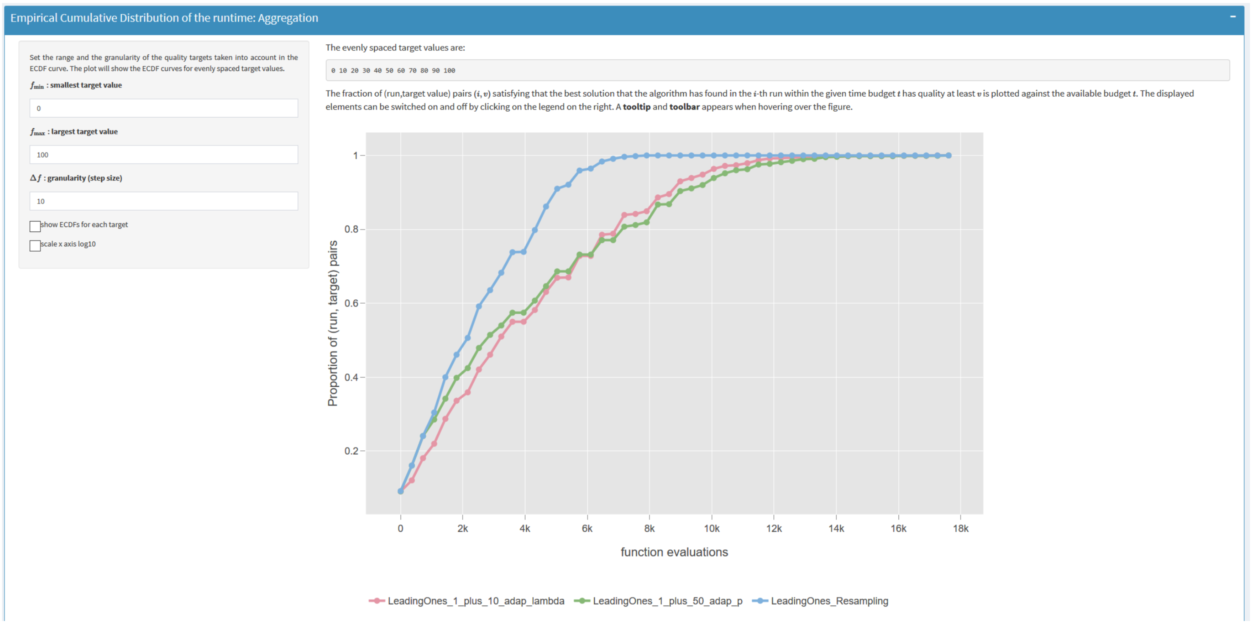
☐ scale y axis log10

Warning! The **probability mass function** of the runtime is approximated by the treating the runtime as a *continuous* random variable and applying the **kernel estimation** (KDE):

The plot shows the distribution of the first hitting times of the individual runs (dots), and an estimated distribution of the probability mass function. The displayed algorithms can be selected by clicking on the legend on the right. A **tooltip** and **toolbar** appear when hovering over the figure. This also includes the option to download the plot as png file. A csv file with the runtime data can be downloaded from the Data Summary tab.

runtime / function evaluations
8000
7000
6000
5000
4000
3000
2000
1000
LeadingOnes_1_plus_10_adap_lambda
LeadingOnes_1_plus_50_adap_p
LeadingOnes_Resampling

Figure 8: 'Fixed-Target Results: Probability Mass Function': approximated probability mass function for the fixed-target running times $T(A, f, v, i)$.


Empirical Cumulative Distribution of the runtime: Aggregation

Set the range and the granularity of the quality targets taken into account in the ECDF curve. The plot will show the ECDF curves for evenly spaced target values.

$f_{\min}$: **smallest target value**

0

$f_{\max}$: **largest target value**

100

Δf : **granularity (step size)**

10

☐ show ECDFs for each target

☐ scale x axis log10

The evenly spaced target values are:

```
0 10 20 30 40 50 60 70 80 90 100
```

The fraction of (run,target value) pairs (i, v) satisfying that the best solution that the algorithm has found in the i-th run within the given time budget t has quality at least v is plotted against the available budget t. The displayed elements can be switched on and off by clicking on the legend on the right. A **tooltip** and **toolbar** appears when hovering over the figure.

Proportion of (run, target) pairs
1
0.8
0.6
0.4
0.2
0
2k
4k
6k
8k
10k
12k
14k
16k
18k
function evaluations
LeadingOnes_1_plus_10_adap_lambda
LeadingOnes_1_plus_50_adap_p
LeadingOnes_Resampling

Figure 9: 'Fixed-Target Results: Cumulative Distribution': Aggregated ECDF curves for selected target values.

part of the 'Cumulative Distribution' tab displays these normalized values for the (equally-spaced) target values chosen by the user, cf. Figure 10.

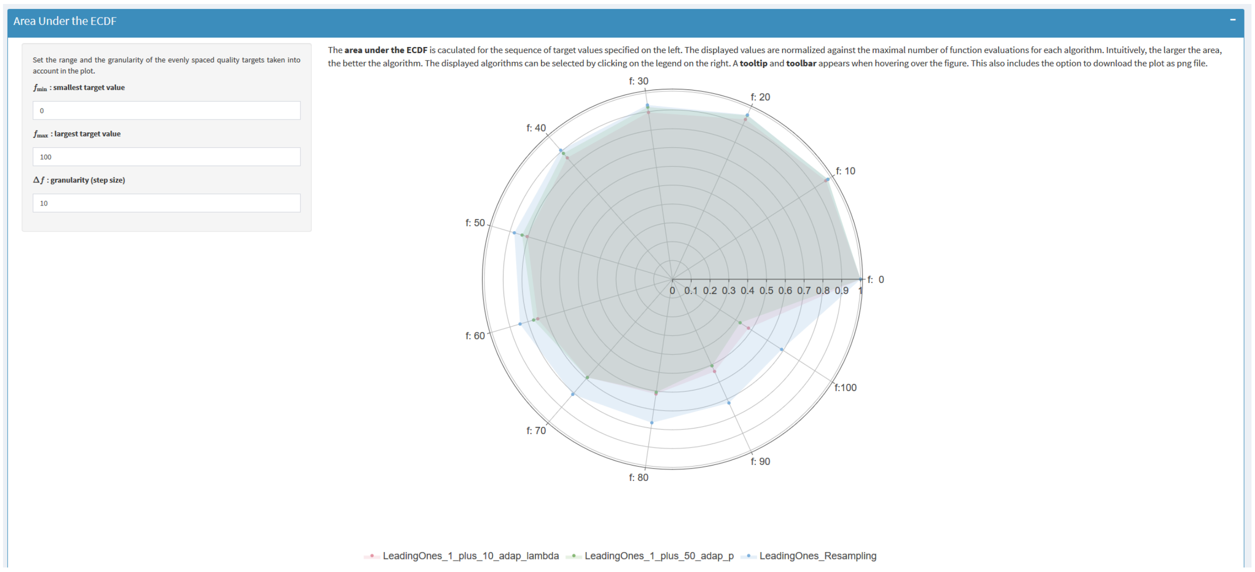
ECDF curves for individual targets are available in the 'Single Target' section of the 'Cumulative Distribution' tab. An example is shown in Figure 11.

4.4 Fixed-Budget Results

The fixed-budget section has the same four tabs as the fixed-target section:

- 'Data Summary': tables with fixed-budget running time statistics and sorted raw values $V(A, f, t, i)$ of the individual runs,
- 'Expected Target Values': interactive plot illustrating the best-so-far functions values as a function of the budget, in particular the averages $\widehat{\mathbb{E}}[V(A, f, t)]$, the median $\widehat{\mathbb{Q}}[V(A, f, t), 50]$, and standard deviations,
- 'Probability Mass Function': interactive histograms of the $V(A, f, t, i)$ values for budgets t selected by the user and an approximation for an empirical probability mass function for $V(A, f, t)$, and
- 'Cumulative Distribution': ECDF curves and normalized values for the area under the ECDF curve for budgets specified by the user.

The plots are similar to those presented in Section 4.3, we omit a detailed description.


Area Under the ECDF

Set the range and the granularity of the evenly spaced quality targets taken into account in the plot.

$f_{\min}$: **smallest target value**

0

$f_{\max}$: **largest target value**

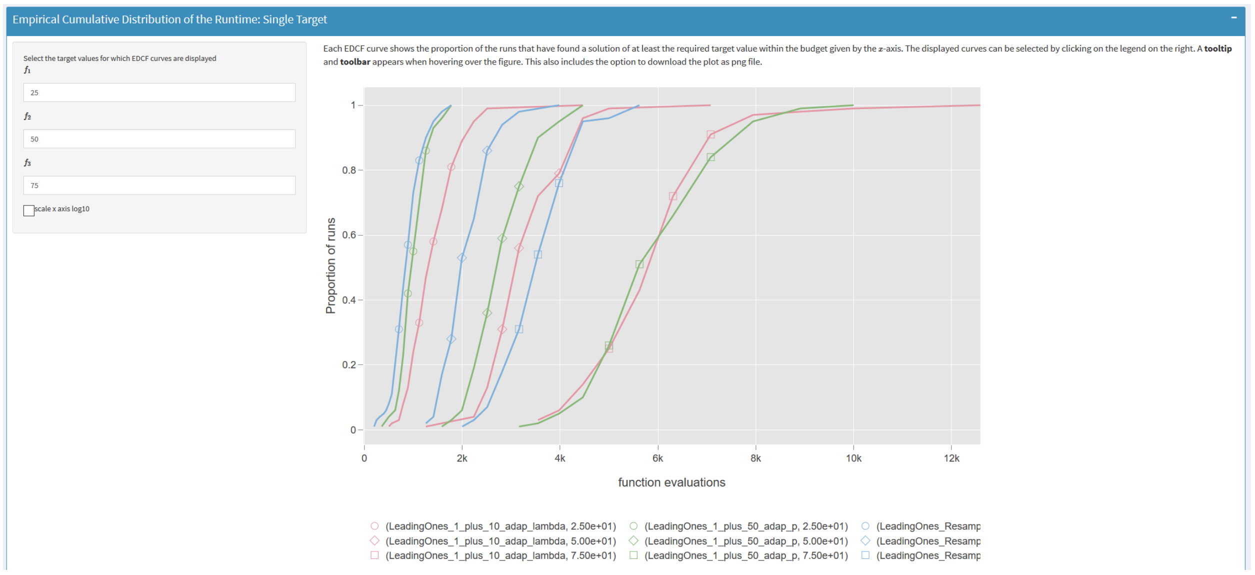
100

Δf : **granularity (step size)**

10

The **area under the ECDF** is caculated for the sequence of target values specified on the left. The displayed values are normalized against the maximal number of function evaluations for each algorithm. Intuitively, the larger the area, the better the algorithm. The displayed algorithms can be selected by clicking on the legend on the right. A **tooltip** and **toolbar** appears when hovering over the figure. This also includes the option to download the plot as png file.

Figure 10: ‘Fixed-Target Results: Cumulative Distribution’: Area under the ECDF curves for selected target values, normalized by the area of the theoretically optimal algorithm evaluating a point with function value $\geq f_{\max}$ in the first iteration.


Empirical Cumulative Distribution of the Runtime: Single Target

Select the target values for which EDCF curves are displayed

f_1

25

f_2

50

f_3

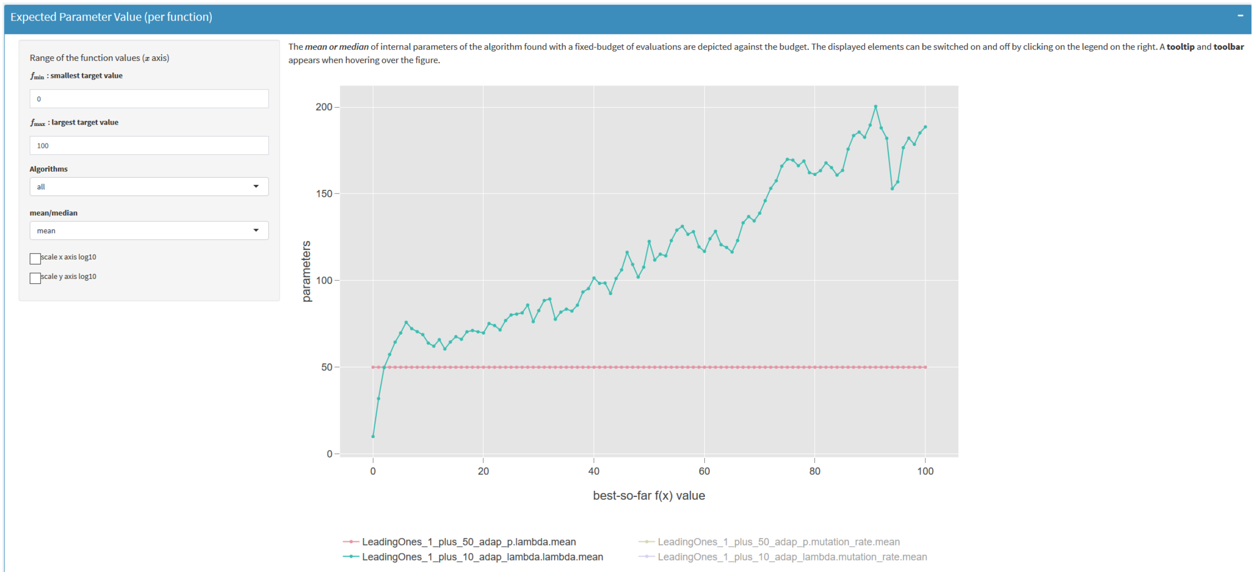
75

☐ scale x axis log10

Each EDCF curve shows the proportion of the runs that have found a solution of at least the required target value within the budget given by the x-axis. The displayed curves can be selected by clicking on the legend on the right. A **tooltip** and **toolbar** appears when hovering over the figure. This also includes the option to download the plot as png file.

1
0.8
0.6
0.4
0.2
0
Proportion of runs
0
2k
4k
6k
8k
10k
12k
function evaluations
(LeadingOnes_1_plus_10_adap_lambda, 2.50e+01)
(LeadingOnes_1_plus_10_adap_lambda, 5.00e+01)
(LeadingOnes_1_plus_10_adap_lambda, 7.50e+01)
(LeadingOnes_1_plus_50_adap_p, 2.50e+01)
(LeadingOnes_1_plus_50_adap_p, 5.00e+01)
(LeadingOnes_1_plus_50_adap_p, 7.50e+01)
(LeadingOnes_Resamp
(LeadingOnes_Resamp
(LeadingOnes_Resamp

Figure 11: ‘Fixed-Target Results: Cumulative Distribution’: ECDF curves for individual target values.


Expected Parameter Value (per function)

Range of the function values (x axis)

$f_{\min}$: **smallest target value**

0

$f_{\max}$: **largest target value**

100

Algorithms

all

mean/median

mean

☐ scale x axis log10

☐ scale y axis log10

The ***mean or median*** of internal parameters of the algorithm found with a fixed-budget of evaluations are depicted against the budget. The displayed elements can be switched on and off by clicking on the legend on the right. A **tooltip** and **toolbar** appears when hovering over the figure.

200
150
100
50
0
parameters
0
20
40
60
80
100
best-so-far f(x) value
LeadingOnes_1_plus_50_adap_p.lambda.mean
LeadingOnes_1_plus_10_adap_lambda.lambda.mean
LeadingOnes_1_plus_50_adap_p.mutation_rate.mean
LeadingOnes_1_plus_10_adap_lambda.mutation_rate.mean

Figure 12: ‘Algorithm Parameters’: Average parameter values at given target values.

4.5 Parameter Evolution

In this section the user can track the evolution of the parameters (cf. Section A.6.2 for an example explaining how to record this data in the experimental part of IOHPROFILER). In the example of Figure 12, we see that the Algorithm *LeadingOnes_1_plus_50_adap_p* (red curve) used a static population size of 50, while Algorithm *LeadingOnes_1_plus_10_adap_lambda* (green curve) uses a dynamic population size. Starting from solutions of function value 80, the average population size of this algorithm was around 163. A table containing average values as well as quantiles and standard deviations can be downloaded/stored on the bottom of this tab.

The corresponding fixed-budget results will be made available shortly.

5 Conclusions and Possible Extensions

An important aspect of benchmarking, which we have taken aside in this present work, is the **selection of suitable benchmark problems.** The user can apply IOHPROFILER to any set of optimization problems that can be formulated as maximization of a function $f : \mathcal{S} \rightarrow \mathbb{R}$. In ongoing collaborations with various colleagues, most notably working group 3 from COST action CA15140, we aim to present to the community a suggestion of benchmark functions that should be included in a standardized benchmark set. We recall that the continuous counterpart COCO [HAM$^+$16], on which IOHPROFILER is built, compares in the single-objective, static, and noise-free case performance across 24 functions, which are grouped into 5 sets, according to whether or not they are separable, uni- or multi-modal, well- or ill-conditioned, and according to whether or not they exhibit a global structure, cf. [HFRA09] for details. For the discrete benchmarking, we suggest to start the discussion which problems to include in the benchmark environment by the question which problem features should be represented, and across which of them performance should be aggregated.

All performance indicators provided by IOHprofiler are based on counting function evaluations. In the long run, it will be desirable to allow for a comparison between iterative and non-iterative optimization methods such as Mathematical Programming. To this end, the outputs of IOHprofiler will have to be adjusted to **time-based performance indicators.** A major challenge posed by the latter is the question if or how to provide system-independent performance measures, i.e., results that do not depend on the hardware on which the algorithms are run.

An important aspect that we plan to address in future work is the extension of IOHprofiler to allow for comparisons of noisy, dynamic, constrained, or multi-objective optimization problems. Finally, we also consider to extend IOHprofiler to other search domains, e.g., permutation-based problems.

As a short-term perspective, we will include additional performance measures, in particular a comparison across different dimensions and some standard statistical tests. Concerning the experimental part, we are most notably working on simplifying the creation of different problem suits.

Acknowledgments

We thank our colleagues Anne Auger, Dimo Brockhoff, Arina Buzdalova, Maxim Buzdalov, Johann Dréo, Nikolaus Hansen, Pietro S. Oliveto, Ofer Shir, Markus Wagner, and Thomas Weise for various discussions around the benchmarking of iterative optimization heuristics.

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.

Our work has been supported 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.

References

[DKLW13] Benjamin Doerr, Timo Kötzing, Johannes Lengler, and Carola Winzen. Black-box complexities of combinatorial problems. *Theoretical Computer Science*, 471:84–106, 2013.

[HAM$^+$16] N. Hansen, A. Auger, O. Mersmann, T. Tušar, and D. Brockhoff. COCO: A platform for comparing continuous optimizers in a black-box setting. *ArXiv e-prints*, arXiv:1603.08785, 2016.

[HFRA09] Nikolaus Hansen, Steffen Finck, Raymond Ros, and Anne Auger. Real-Parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions. Research Report RR-6829, INRIA, 2009.

[LW12] Per Kristian Lehre and Carsten Witt. Black-box search by unbiased variation. *Algorithmica*, 64:623–642, 2012.

[RV11] Jonathan Rowe and Michael Vose. Unbiased black box search algorithms. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'11)*, pages 2035–2042. ACM, 2011.

Appendix

A Manual for the Experimental Part: Data Generation

In this section we describe the experimental part of IOHprofiler and provide an illustrated manual, which enables the user to conduct experiments of their choice. We recall that this part is build upon the COCO (COmparing Continuous Optimisers) platform [HAM+16], from which the data structure and some tool functions are inherited.

A.1 Preparation

Running IOHprofiler requires a working python environment and a C compiler. The benchmark problems as well as the algorithms can be provided in either python or in C. IOHprofiler has currently been tested with python 2.7.12 and with gcc 5.4.1. A version accepting algorithms and problems in Java is in preparation.

Both the experimental and the post-processing parts of IOHprofiler can be downloaded from the GitHub page `https://github.com/IOHprofiler`. After downloading the zipped files of the experimental part, the archive needs to be extracted.

A.2 Overview of the Main Steps

The files of the C [python] interface are located at the path "/code-experiments/build/c" ["/code-experiments/build/python"].

To run an experiment, the following steps need to be executed.

- **1. Benchmark Selection and Configuration:** The user needs to select the set of problems (hereafter called "the suite") for which benchmark data shall be generated. The definition of the suite is done in the configuration file `configuration.ini`, which is also used to select the granularity at which performance data is stored, the location at which the results are stored, etc. The configuration file and the suite definition are described in Section A.3.
- **2. Algorithm Setup:** The algorithm for which the performance data is generated needs to be defined in the file `user_algorithm.c` [`user_algorithm.py`]. To this end, the content of the function "user_algorithm" (which includes as example the code for pure random search) is replaced by the new algorithm.
- **3. Data Generation**: After ensuring that the working path is "code-experiments/build/c" ["code-experiments/build/python"], the execution of the statement "python ../../../do.py run-c" ["python ../../../do.py run-python"] generates the performance data, which is saved in the current path.

A.3 The Configuration File

IOHprofiler applies the INI file format for the configuration file ”configuration.ini”. All variables are grouped into three sections: [suite], [observer], and [triggers].

The section [**suite**] contains information about the benchmark problems for which performance data is generated. There are four keys in this section:

- suite_name is the name of the suite. The suite is defined in the file `suite_PBO.c` at ”/code-experiments/scr/”. As an example, the suite PBO, which contains the benchmark problems OneMax, LeadingOnes, a linear function with random weights between 0 and 5, and a jump function with gap size $k = 1$ is pre-defined as an example. For the time being, we recommend that users keep using the PBO suite, and add their benchmark functions to it. The design of a new suite is possible, but not recommended.
- functions_id: IDs of the selected benchmark problems. An en-dash '-'is allowed to present the range of problem IDs, for example, ”1-4”. Alternatively, the problems can be listed by comma, for example, ”1,2,3,4”. An explanation of how to add new benchmark problems and how to assign the function IDs will be given in Section A.4. In the distributed version the following benchmark problems are already defined:

  1 OneMax

  2 LeadingOnes

  3 Jump with jump size $k = 1$

  4 A linear function with fixed, but randomly chosen weights between 0 and 5.
- instances_id: IDs of the problem instances, cf. Section A.5. Instances can be selected using commas and en-dashes, e.g., ”1-25,75,80-100”.
- dimensions: the selection of the problem dimensions, e.g., ”100,500,1000” will create experimental data for the three different problem dimensions.

The [**observer**] section contains information concerning the output files. There are five keys in this section:

- observer_name: the name of observer. We suggest to use PBO for the time being, and updated description will be made available when the functionality to create new suits has been improved.
- result_folder: the name of folder where the results will be stored. If the folder does not exist, it will be created automatically.
- algorithm_name: a name for the algorithm. This information will be stored in the output files.
- algorithm_info: users can write here additional information about the algorithm here, this information will be stored in the .info file of the output, cf. Section 3.

- parameters_name: a list of parameters to be stored. If no algorithm parameters are to be stored, the users leaves this as "". If several parameters are to be stored, they are separated by a comma, e.g., "p1, p2, p3".

In the **[triggers]** section the user decides the granularity of the performance data. According to the user's choice, up to four different files will be created, cf. Section 3 for a description of the available output files.

There are four keys in the [triggers] section.

- complete_triggers: Set as "true" to output *.cdat files.
- number_interval_triggers: The step size for *.idat files. For example, selecting number_interval_triggers = 50 will store results of every 50-th function evaluation. If the user does not wish to generate *.idat files, he/she selects number_interval_triggers = 0.
- number_target_triggers: the budget t of storing information for every $[10^i, 10^{i+1}]$ in *.tdat files. If the user does not wish to generate *.tdat files, he/she selects number_target_triggers = 0.
- base_evaluation_triggers: A set of parameter v for *.tdat files, for example, "1,2,5" means storing information every $1 * 10^i$-th, $2 * 10^i$-th and $5 * 10^i$-th function evaluation in *.tdat files. If the user does not wish to generate *.tdat files, he/she selects base_evaluation_triggers = 0.

A.4 Adding New Benchmark Problems

IOHprofiler allows to add new user-defined benchmark problems. To do so, the user needs to create a problem file, which contains the definition of the function, and needs to include the problem into a suite.

The benchmark problems are defined in `f_*.c` files. The probably easiest way to create a new benchmark problem is to copy the OneMaxexample contained in file `f_one_max.c`, and to adjust it to the new function.

The actual definition of the OneMaxfunction is contained in the function f_one_max_raw. The user should replace the content of this function by his/her problem. All occurrences of "one_max" need to be replaced by the name of the new problem (we use "new_problem" in the following). If different problem instances are desired, the user creates these in the function f_new_problem_IOHProfiler_problem_allocate, cf. Section A.5 for details.

To be used by IOHprofiler, the new problem needs to be added to a problem suite. For example, the pre-installed suite "PBO" is defined in the file `suite_PBO.c`. To include the new problem in this suite, the problem file `f_new_problem.c` is added to the header of the file `suite_PBO.c`. Then, the f_new_problem_IOHProfiler_problem_allocate function of the new problem needs to be called in the function PBO_get_problem. Finally, the problem numbers need to be modified in the function suite_PBO_initialize.

A.5 Problem Instances

Many standard IOHs are representation-invariant, in the sense that their performance is identical on each fitness landscape, regardless of how it is embedded. That is, the performance is oblivious of rotations and shifts. Furthermore, it is sometimes argued that performance should also be oblivious with respect to a scaling of the function values—algorithms respecting this invariant are often referred to as "comparison-based". Users interested in testing how sensitive their algorithms are with respect to search space and/or fitness landscape transformations can make use of build-in transformations: To this end—similar to the COCO framework—IOHprofiler offers to test performance on various instances of the same problem. Precisely, the following four transformations are available. They can be combined with each other, cf. also Section 2.2 of the main document. As mentioned above, these transformations are chosen by the user in function f_new_problem_IOHProfiler_problem_allocate in the file `f_new_problem.c`.

- **transform_obj_scale:** multiplicative shift of the function values, i.e., instead of $f(x)$ the transformed function values $af(x)$ are returned to the algorithm.
- **transform_obj_shift:** additive shift of the function values, i.e., instead of $f(x)$ the algorithms receive the transformed function values $f(x) + b$.
- **transform_vars_xor:** an XOR of the search point, i.e., a shift in the search space. Instead of evaluating $f(x)$, the function values $f(x \oplus z)$ are computed and returned to the algorithm. Note that in this case the instance $f(\cdot \oplus z)$ has a fitness landscape that is isomorphic to that of the original function f. Algorithms that are *unbiased* in the sense of [LW12, RV11, DKLW13] show the same performance on any of these instances.
- **transform_vars_sigma:** a permutation of the search point, i.e., instead of computing function values $f(x_1, x_2, ..., x_n)$, the algorithms receive the function values $f(\sigma(x))$ of the permuted search points $\sigma(x) = (x_{\sigma(1)}, x_{\sigma(2)}, ..., x_{\sigma(n)}$, where σ is a permutation of the set $\{1, 2, \ldots, n\}$.

Using OneMaxas an example, we demonstrate how to define the different problem instances. As a general rule, we recommend to reserve instance "1" for the original benchmark problem, i.e., the one that does not call any of the four transformations.

We first demonstrate to convert the original OneMaxfunction f so that instead of $f(x)$ values the algorithm receives the function values $af(x \oplus z) + b$. To this end, we first assign to "problem" the original OneMaxinstance (cf. line 1 in the example below). After that, the instance $f(\cdot)$ is transformed to $f(\cdot \oplus z)$ (line 6), where z is a pseudo-randomized binary vector chosen in line 2. In lines 7 and 8, we then transform the instance $f(\cdot \oplus z)$ to $af(\cdot \oplus z)$ and to $af(\cdot \oplus z) + b$, respectively. The multiplicative and additive shifts "a" and "b" are again two pseudo-random numbers, which are chosen in lines 3-5. The ranges for a and b are $[1/5, 5]$ and $[-1000, 1000]$, respectively.

```
1 problem = f_one_max_allocate(dimension);
2 IOHprofiler_compute_xopt(z,rseed,dimension);
3 a = IOHprofiler_compute_fopt(function,instance + 100);
4 a = fabs(a) / 1000 * 4.8 + 0.2;
5 b = IOHprofiler_compute_fopt(function,instance);
6 problem = transform_vars_xor(problem,z,0);
7 problem = transform_obj_scale(problem,a);
8 problem = transform_obj_shift(problem,b);
```

The following code shows how to transform the original instance $f(\cdot)$ to $af(\sigma(\cdot)) + b$, where we recall that we denote by $\sigma(x)$ the permuted string $(x_{\sigma(1)}, \ldots, x_{\sigma(n)})$. After being allocated with the original OneMaxinstance $f(\cdot)$, "problem" is transformed to $f(\sigma(\cdot))$ in line 13, where σ is a pseudo-random permutation chosen in lines 3 to 9. Then, following the same procedure as in the example above, "problem" is transformed to $af(\sigma(\cdot))$ in line 14 and, finally, to $af(\sigma(x)) + b$ in line 15, where a and b are again random numbers, chosen in lines 10 to 12.

```
1  problem = f_one_max_allocate(dimension);
2  IOHProfiler_compute_xopt_double(xins,rseed,dimension);
3  for(i = 0; i < dimension; i++){
4      sigma[i] = i;
5  }
6  for(i = 0; i < dimension; i++){
7      t = (int)(xins[i] * dimension);
8      temp = sigma[0];sigma[0] = sigma[t];sigma[t] = temp;
9  }
10 a = IOHProfiler_compute_fopt(function,instance + 100);
11 a = fabs(a) / 1000 * 4.8 + 0.2;
12 b = IOHProfiler_compute_fopt(function, instance);
13 problem = transform_vars_sigma(problem, sigma, 0);
14 problem = transform_obj_scale(problem,a);
15 problem = transform_obj_shift(problem,b);
```

A.6 Examples

The user can find two examples in the git folder `/example/`:

- `/example/example1/` includes the code and the results of pure random search (hereafter called "random search"), while
- `example/example2/` includes the code and results of a $(1 + \lambda)$ evolutionary algorithm.

We describe the configuration of these examples in Sections A.6.1 and A.6.2.

A.6.1 Example 1: Pure Random Search

The example of the "random search" method is located at the path `/example/example1/`.

The "configuration.ini" file is set as follows.

```
[suite]
suite_name = PBO
functions_id = 1-4
instances_id = 1-100
dimensions = 100
[observer]
observer_name = PBO
result_folder = EXP
algorithm_name = RANDOM_SEARCH
algorithm_info = RANDOM_SERACH
```

parameters_name = evaluation
[**triggers**]
complete_triggers = true
number_interval_triggers = 10
number_target_triggers = 3
base_evaluation_triggers = 1,2,5

Based on this configuration file, performance data is collected for the algorithm optimizing the 100-dimensional variants of the benchmark problems 1, 2, 3, and 4. For each problem, the instances from 1 to 100 are used. For each instance the number of independent runs performed in the experimental part is specified in the variable "INDEPENDENT_RESTARTS" in the file containing the algorithm, cf. example below. If, for example, the user wishes to run each of the instances 1-100 instances twice, he/she sets "instance_id = 1-100" in the configuration file, and sets "INDEPENDENT_RESTARTS = 2" in the algorithm file.

The results of this example experiment will be stored in the folder `./EXP/`. The name of the parameter to be stored is set as "evaluation"; this will be the header of the respective column in the output files.

In this example four different output files will be created:
- *.cdat storing data of each iteration,
- *.idat storing data from every 10-th iteration,
- *.tdat storing data from every $10^{n/3}$-th, every $1 * 10^n$-th, $2 * 10^n$-th, and $5 * 10^n$-th function evaluation,
- *.dat storing data for each iteration in which an improvement has been found, where * is of the form IOHprofiler_f1_DIM100_i1, as explained in Section 3.

The user_algorithm that implements the pure random search is implemented in file `/example/example1/c/user_algorithm.c` as follows:

```c
static const size_t BUDGET_MULTIPLIER = 50;
static const size_t INDEPENDENT_RESTARTS = 1;
void User_Algorithm() {
  size_t number_of_parameters = 1;
  int *x = IOHProfiler_allocate_int_vector(dimension);
  double *y = IOHProfiler_allocate_vector(number_of_objectives);
  double *p = IOHProfiler_allocate_vector(number_of_parameters);
  size_t i, j;

  for (i = 0; i < max_budget; ++i) {
    for (j = 0; j < dimension; ++j) {
      x[j] = (int)(IOHProfiler_random_uniform(random_generator) * 2);
    }
    p[0] = i + 1;
    set_parameters(number_of_parameters,p);
    evaluate(x, y);
  }

  IOHProfiler_free_memory(x);
  IOHProfiler_free_memory(y);
}
```

The user needs to set two parameters in the `user_algorithm.c` [`user_algorithm.py`].

- BUDGET_MULTIPLIER: This parameter controls the maximal budget of function evaluations, which is set to BUDGET_MULTIPLIER times the dimension of the problem.
- INDEPENDENT_RESTARTS: The number of independent runs of the algorithm for each instance. That is, setting "INDEPENDENT_RESTARTS=100" and "instance_id=1-3" will result in an overall number of 300 runs—100 independent runs for each of the first three instances.

With the code above, the maximal number of evaluations for each run is 50*dimension, and the algorithm will not restart within one run.

For each iteration, a new individual x is generated randomly (line 12), and the fitness is evaluated by line 16. Note here that y is a vector that stores the fitness of x. Also, a parameter p (evaluation step) will be logged in output files (line 15), and its logging name is defined in the configuration file as "evaluation". We will see in the next section an example where more than one parameter are stored.

A.6.2 Example 2: A $(1 + \lambda)$ EA

This example of a $(1 + \lambda)$ EA is located at the path `/example/example2/`. The configuration file is as follows.

```
[suite]
suite_name = PBO
functions_id = 1-4
instances_id = 1
dimensions = 100,500,1000
[observer]
observer_name = PBO
result_folder = EXP
algorithm_name = ONE_PLUS_LAMDA_EA
algorithm_info = ONE_PLUS_LAMDA_EA
parameters_name = mutation_rate,l
[triggers]
complete_triggers = true
number_interval_triggers = 50
number_target_triggers = 0
base_evaluation_triggers = 0
```

Based on this configuration file, running time data is generated for the benchmark problems with function IDs 1 to 4. For each function, the algorithm will be run for dimension 100, 500, and 1000. Only the first instance (without transformation) is tested. All results will be stored in the folder `./EXP/`, and the names of the two parameters which are tracked are set as "mutation_rate" and "l" (the number of bits in which parent and offspring differ).

The three data files *.idat (storing data after every 50-th evaluation), *.cdat, and *.dat will be generated.

The user_algorithm that implements the $(1 + \lambda)$ EA is as follows:

```python
import numpy as np
import random as rd
import math

independent_restart = 10
budget = 50

def mutation(ind,mutation_rate,dim):
    l = 0
    while l == 0:
        l = np.random.binomial(dim,mutation_rate)
    flip = rd.sample(range(0,dim),l)

    for index in flip:
        ind[index] = (ind[index] + 1) % 2

    return l

def user_algorithm(fun,lbounds,ubounds,budget):
    lbounds, ubounds = np.array(lbounds), np.array(ubounds)
    dim = fun.dimension
    parent = lbounds + (ubounds - lbounds + 1) * np.random.rand(dim)
    parent = parent.astype(int)
    best = parent.copy()
    budget -= 1
    lamb = 1
    mutation_rate = 1.0/dim
    para = np.array([mutation_rate])
    fun.set_parameters(para)
    best_value = fun(parent)
    while budget > 0:
        for i in range(0,lamb):
            offspring = parent.copy()
            l = mutation(offspring,mutation_rate,dim)
            para = np.array([mutation_rate,l])
            fun.set_parameters(para)
            v = fun(offspring)
            if v > best_value :
                best_value = v
                best = offspring.copy()
            budget -= 1
            if(budget == 0):
```

```
43                 break
44         parent = best.copy()
45         mutation_rate = 1.0 / (1 + (1 - mutation_rate) / mutation_rate
              * math.exp(0.22 * np.random.normal()))
46         mutation_rate = min(max(mutation_rate,1.0/dim),0.5)
47     return best_value
```

With the code above, the maximal number of function evaluations for each run is 50*dimension, and the algorithm will do ten independent runs for each selected instance (as discussed above, only instance 1 has been selected in the configuration file).

For each generation, λ offspring are created by mutating the parent individual (line 34). Their fitness is evaluated in line 36. The function fun(offspring) returns the fitness of the offspring.

In addition to the information about the fitness values, a vector of parameters ('para') will be stored in the output files (line 36). The vector stores the mutation_rate and the number of flipped bits (line 35). The names attributed to these parameters are chosen as "mutation_rate, l" in the `configuration.ini` file.

A.7 Overview of the Different Files

The following lists summarize the folders and the main files that can be found in the experimental part of IOHprofiler. The files that need to be edited by the user are formatted in bold font.

Folder **/code-experiments/**:

- /src/ : a folder of source files
- /build/ : a folder of C and Python interfaces
- /tools/ : some common tools for the project

Folder **/example/**:

- /example1/ : examples of random search method, cf. Section A.6.1
- /example2/ : examples of $(1 + \lambda)$ EA, cf. Section A.6.2

Folder **/src/**:

- f_binary.c : Implementation of the binary function and problem
- f_jump.c : Implementation of the jump function and problem
- f_leading_ones.c : Implementation of the leading ones function and problem
- f_linear.c : Implementation of the linear function and problem
- f_one_max.c : Implementation of the one_max function and problem
- IOHProfiler.h : Header file for all public IOHProfiler functions and variables
- IOHProfiler_internal.h : Definitions of internal IOHProfiler structures and typedefs

- IOHProfiler_observer.c : Definitions of functions regarding IOHProfiler observers
- IOHProfiler_platform.h : Automatic platform-dependent configuration of the IOHProfiler framework
- IOHProfiler_problem.c : Definitions of functions regarding IOHProfiler problems
- IOHProfiler_random.c : Definitions of functions regarding IOHProfiler random numbers
- IOHProfiler_runtime_c.c : Generic IOHProfiler runtime implementation for the C language
- IOHProfiler_string.c : Definitions of functions that manipulate strings
- IOHProfiler_suite.c : Definitions of functions regarding IOHProfiler suites
- IOHProfiler_utilities.c : Definitions of miscellaneous functions used throughout the IOHProfiler framework
- suite_PBO_legacy_code.c : Methods for generating pseudo random numbers
- logger_PBO.c : Implementation of the PBO logger
- observer_PBO.c : Implementation of the PBO observer
- **suite_PBO.c** : Selection of functions to be included in the PBO suite
- transform_obj_shift.c : Implementation of shifting the objective value by the given offset
- transform_obj_scale.c : Implementation of scaling the objective value by the given offset
- transform_vars_shift.c : Implementation of shifting all decision values by an offset
- transform_vars_xor.c : Implementation of the xor of all decision values by an offset
- transform_vars_sigma.c : Implementation of re-ordering all decision values by permuting the string of decision values

Folder **/build/c/**:

- Makefile : Makefile to build the C program
- user_experiment.c : The interface to invoke user algorithm
- **user_algorithm.c** : The file where the user defines his/her algorithm
- **configuration.ini** : the configuration file, cf. Section A.3
- ...

Folder **/build/python/**:

- user_experiment.py : The interface to invoke user algorithm
- **user_algorithm.py** : The file where the user defines his/her algorithm
- **configuration.ini** : the configuration file, cf. Section A.3
- ...