From benchmarking optimization heuristics to dynamic algorithm configuration

Vermetten, D.L.

Chapter 3

Benchmarking Optimization Algorithms

With the ever-increasing number of available optimization heuristics, insights into which algorithm effectively tackles which kinds of problem characteristics can be hard to come by. Theoretical analysis is often limited to relatively small sets of problems and algorithms, so researchers are forced to rely on empirical results to judge their algorithm's effectiveness. *Benchmarking* aims to bridge the gap between theory and practice by collecting sets of problems with known characteristics, such that an algorithm's performance on these problems can be interpreted and compared to other algorithms.

Benchmarking optimization algorithms in a robust and reproducible manner is a key challenge in the field. In any given study, design choices regarding the used problems (including instances of the same problem), number of runs, evaluation budget, performance criteria etc. have to be made, since each of these variables influences the types of comparisons which are available, as well as how they can be interpreted.

To ease the barrier to robust, yet flexible benchmarking, we propose the IOHprofiler framework in Section 3.1 and illustrate how this tool can contribute to multiple parts of the benchmarking pipeline. Since rigorous benchmarking can result in large amounts of performance data, we also discuss a project related to the creation of an optimization ontology, OPTION, in Section 3.1.4. The core of this section is based on [255, 53, 136], while the problem sets were presented in [43, 177, 242].

As one of the most critical parts of the benchmarking pipeline, we pay particular

attention to the choice of optimization problems used in benchmarking algorithms for continuous optimization: the BBOB suite. In Section 3.2, we discuss an in-depth analysis of the BBOB functions and the corresponding instance generation process, originally published in [150].

Finally, in Section 3.3 we note that benchmarking is not limited to the performance-oriented viewpoint by summarizing several papers on behavior-based benchmarking in the form of structural bias detection [244, 236].

3.1 IOHprofiler

In this section, we introduce IOHprofiler, a tool developed as a collaboration between Leiden University and Sorbonne University (France), with additional input and design help from Tel-Hai College (Isreal). IOHprofiler is a benchmarking platform that aims to integrate elements of the entire benchmarking pipeline, ranging from problem (instance) generators and modular algorithm frameworks over automated algorithm configuration techniques and feature extraction methods to the actual experimentation, data analysis, and visualization [63, 255, 53]. An illustration of the interplay between these different components is provided in Figure 3.1. Notably, IOHprofiler provides the following components:

- **IOHproblems:** a collection of benchmark problems. This comprises pseudo-Boolean, discrete, and continuous problem suites [63, 95], as well as several parameterized problem generators [256]. Section 3.1.3 discusses this component in more detail.

- **IOHalgorithms:** a collection of IOHs. For the moment, the algorithms used for the benchmark studies presented in [63, 7, 51] are available. This subsumes textbook algorithms for pseudo-Boolean optimization, an integration to the object-oriented algorithm design framework ParadisEO [121], and the modular algorithm framework for CMA-ES variants originally suggested in [233] and extended in [51] (discussed further in Chapter 7). 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 several benchmarking studies [63, 134, 208], a sample data set used to illustrate IOHanalyzer functionality in Section 3.1.2, and all compatible data sets from the single-objective,

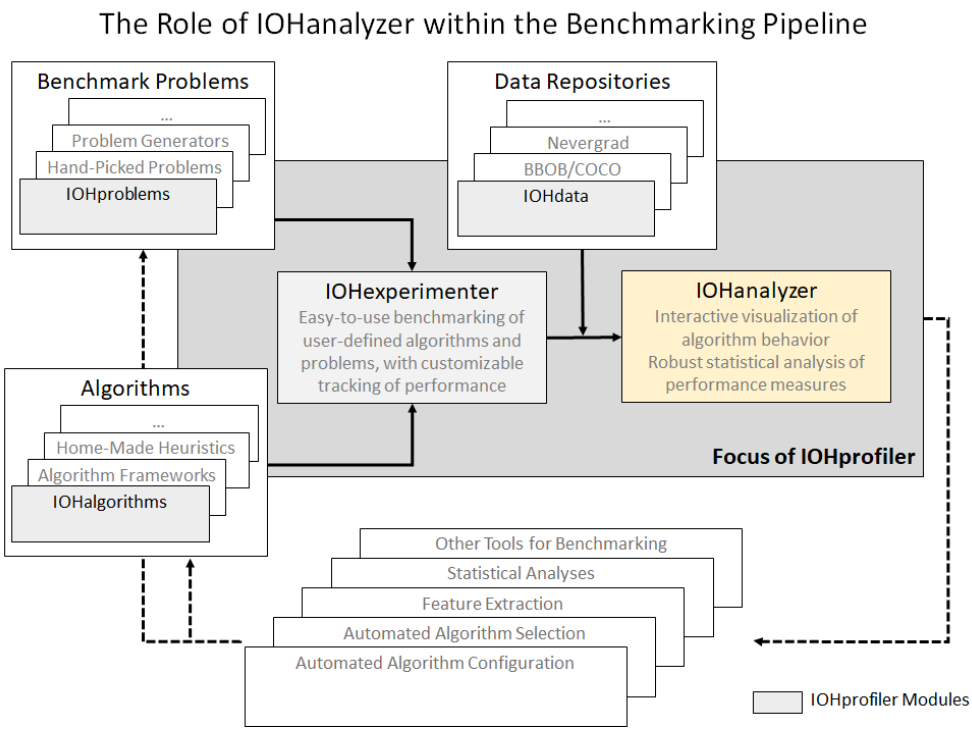


Figure 3.1: Schematic overview of the IOHprofiler, where the focus lies on the IOHanalyzer and IOHexperimenter.

noiseless version of the COCO repository [4]. **IOHdata** also contains performance data from Facebook's Nevergrad benchmarking environment [200], which is updated periodically.

- **IOHexperimenter:** the experimentation environment that executes IOHs on **IOHproblems** or external problems and automatically takes care of logging the experimental data, which will be discussed in Section 3.1.1.
- **IOHanalyzer:** the data analysis and visualization tool presented in Section 3.1.2.

Related Benchmarking Environments

As argued above, benchmarking IOHs is an essential task towards a better understanding of IOHs. It is therefore not surprising that a large number of different tools

have been developed for this purpose. We summarize a few of the tools that come closest to IOHprofiler in terms of functionality and scope.

In evolutionary computation, the arguably best established benchmarking environment is the already mentioned **COCO platform** [95]. Originally designed to compare derivative-free optimization algorithms operating on numeric optimization problems [94], the tool has seen several extensions in the last years, e.g., towards multi-objective optimization [228], mixed-integer optimization [227], and large-scale optimization [238]. COCO consists of an experimentation part that produces data files with detailed performance traces, and an automated data analysis part in which a fixed number of standardized analyses are automatically generated. The by far most reported performance measures from the COCO framework are *empirical cumulative distribution function* (ECDF) curves, see Section 2.4 for definitions. The COCO software has a strong focus on *fixed-target performances* [92], i.e., on the time needed to find a solution of a certain quality.

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

Another important software environment for benchmarking sampling-based optimization heuristics is the **Nevergrad framework** [200]. As with COCO, Nevergrad implements functionalities for both experimentation and performance analysis, accommodating continuous, discrete, and mixed-integer problems. It has a strong focus on noisy optimization, but also comprises several noise-free optimization problems. In addition to studying IOHs, Nevergrad has a special suite to compare one-shot optimization techniques, i.e., non-iterative solvers. The current focus of Nevergrad is to be seen on the problem side, as it offers several new benchmark problems, ranging from modified versions of BBOB to problems optimizing adversarial attacks of image detectors. Nevergrad also provides interfaces to the following benchmark collections: LSGO [143], YABBOB [145], Pyomo [99], MLDA [199], and MuJoCo [226]. The performance evaluation, however, is much more basic than those of COCO or IOHanalyzer, in that only the quality of the finally recommended point(s) is stored, but no information about the search trajectory. That is, apart from taking a *fixed-budget perspective*, Nevergrad does not store performance traces, but only the final output.

3.1.1 IOHexperimenter

To systematically collect performance data from IOHs, a robust benchmarking setup has to be created that allows for rigorous testing of algorithms. For this purpose, we introduce IOHexperimenter. This section is based primarily on [53].

Numerous benchmark problems have been proposed within the evolutionary computation community, and these are often implemented many times over, without an overarching structure or proper maintenance [143, 144, 221]. The importance of using overarching frameworks to facilitate the benchmarking process has been gaining increasing traction within the community in the last decade, especially after [89] showed the benefits that these kinds of tools can provide. Since then, two of the most popular benchmarking tools have been COCO [95] and Nevergrad [200]. While these tools enable users to benchmark their algorithms with relative ease, their overall design has some drawbacks.

In the case of COCO, the enforced design of a suite-based structure allows for very robust benchmarking on problems made available by the developers. However, this simultaneously restricts users to using only that set of available problems and adds a complexity barrier for benchmarking algorithms on other problems. In addition, the logging of performance data follows a fixed framework, and extending it, e.g., to keep track of dynamic algorithm parameters, is not straightforward. Nevergrad, in contrast, offers great flexibility with respect to adding new benchmark problems but is severely limited in terms of the information that is tracked about algorithm performance and behavior. As mentioned above, it essentially only stores the final solution quality after exhausting a user-defined optimization budget.

With IOHexperimenter, we offer a benchmarking module that emphasizes extendability and customizability, allowing users to easily add new problems while providing a comprehensive set of built-in defaults. The logging of performance data is flexible and allows users to customize the content and frequency of the data collected. To improve ease of use, several out-of-the-box storage structures are made available, one of which can be used to collect the same type of data as COCO.

Within the IOHprofiler pipeline, IOHexperimenter can be considered the interface between algorithms and problems, allowing consistent collection of performance data and algorithmic data such as the evolution of control parameters that change during the optimization process. To perform the benchmarking, three components interact with each other: *problems*, *loggers*, and *algorithms*. Within IOHexperimenter, an interface is provided to ensure that any of these components can be modified without

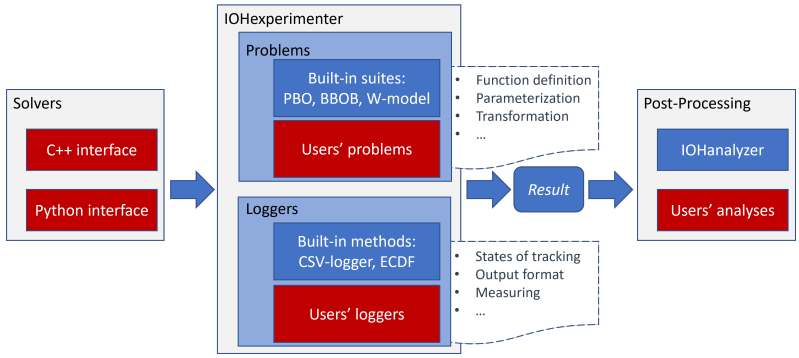


Figure 3.2: Workflow of IOHexperimenter.

impacting the behavior of the others, in the sense that any changes to their setup will be compatible with the other components of the benchmarking pipeline.

Functionality

At its core, IOHexperimenter provides a standard interface towards expandable benchmark *problems* and several *loggers* to track the performance and the behavior (internal parameters and states) of *algorithms* during the optimization process. The logger is integrated into a wide range of existing tools for benchmarking, including *problem* suites such as PBO [63] and the W-model [256] for discrete optimization, COCO's noiseless real-valued single-objective BBOB problems [95] for the continuous case, and submodular problems for constraint optimization [176]. On the *algorithms* side, IOHexperimenter has been connected to several algorithm frameworks, including ParadisEO [121], a modular genetic algorithm [264], a modular CMA-ES [51], and the optimizers in Nevergrad [200]. In [150, 135], the flexibility of IOHexperimenter was demonstrated by generating interfaces between two aforementioned benchmarking tools to execute algorithms from the Nevergrad framework on the BBOB problems from COCO.

Figure 3.2 shows how IOHexperimenter can be placed in a typical benchmarking workflow. The key factor here is the flexibility of its design. IOHexperimenter can be used with any user-provided solvers and problems given a minimal overhead. It also ensures that the output of experimental results follows conventional standards. Because of this, the data produced by IOHexperimenter is compatible with post-processing frameworks like IOHanalyzer [255], enabling an efficient path from algorithm design to performance analysis. In addition to the built-in interfaces to existing software, IOHexperimenter aims at providing a user-friendly, easily accessible way to customize the

benchmarking setup. IOHexperimenter is built in C++, with an interface to Python. In this section, we describe the functionality of the package on a high level, without going into implementation details.[1] In the following, we introduce the typical usage of IOHexperimenter, as well as how it can be customized to fit different benchmarking scenarios.

Problems

Single-Objective Optimization. IOHexperimenter is developed with a focus on single-objective optimization problems, i.e., instances defined as $F = T_y \circ f \circ T_x$, in which $f\colon X \to \mathbb{R}$ is a benchmark problem (e.g., for OneMax $X = \{0,1\}^d$ and the sphere function $X = \mathbb{R}^d$), 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 $X = \mathbb{R}^d$). 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 d of the problem. Any problem instances that reconcile with this definition of F, can easily be integrated into IOHexperimenter, using the C++ core or the Python interface.

The transformation methods are particularly important for robust benchmarking, as they allow for the creation of multiple problem instances from the same base function. They also allow the user to check algorithm invariance to transformations in search and objective space. Built-in transformations are available for pseudo-Boolean functions [63] and for continuous optimization, implementing the transformations used by [95]. Problems can be combined in a *suite*, which allows the user to easily run solvers on collections of selected problem instances.

Constrained Optimization. Similar to benchmark problems, constraints are defined as free functions that compute a value on an evaluated solution, i.e.; $C\colon X \to \mathbb{R}$, that is non-zero in the case the constraint is violated. IOHexperimenter supports both *hard constraints* C_h and *soft constraints* C_s, of which multiple can be added to any given problem. The single-objective constrained problems are defined by $F_c = F \circ C_h \circ C_s$, which evaluates to ∞ when one of the hard constraints C_h is violated (given minimization). Otherwise, $F_c = F + \sum_{i=0}^{k-1} w_i (C_s^i)^{\alpha_i}$, where k is the number of soft constraints. The weight $w_i \geq 0$ and exponent α_i of a constraint C_s^i can be used by the user to customize a penalty for a constraint violation. In this fashion, arbitrary functions can be added as constraints (thus allowing for both equality and

[1] Technical documentation, a getting-started, and several use-cases are available for both C++ and Python on the IOHexperimenter docs at https://iohprofiler.github.io/IOHexperimenter/.

inequality constraints) to the benchmark problems in IOHexperimenter, allowing the conversion of existing unconstrained problems into constrained problems.

Data Logging

IOHexperimenter provides *loggers* to track the performance of algorithms during the optimization process. These *loggers* can be tightly coupled with the problems: when evaluating a solution, the attached loggers will be triggered to store relevant information. Information about solution quality is always recorded, while the algorithm's control parameters are included only if specified by the user. The events that trigger a data record are customized by the user; e.g., via specifying a frequency at which information is stored, or by choosing quality thresholds that trigger a data record when met for the first time.

A default logger makes use of a two-part data format: meta-information such as function ID, instance, and dimension, written to `.json`-files, and the performance data that gets written to space-separated `.dat`-files. A full specification of this format can be found in [255]. Additional loggers to store the data in memory or use different file structures are available. In addition to the built-in loggers, users can also create their own custom logging functionalities. For example, a logger storing only the final calculated performance measure was created for algorithm configuration tasks [65].

3.1.2 IOHanalyzer

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

Several other tools have been developed for displaying performance data and/or the search behavior in decision space. However, all tools that we are aware of allow much less flexibility with respect to the performance measures, the ranges, and the granularity of the analysis or focus on selected aspects of performance analysis only (e.g., [33, 68] study statistical significance, whereas [78, 209] aim to visualize performance with respect to multiple objectives). The ability of IOHanalyzer to link the evolution of algorithms' parameters to the evolution of solutions' quality seems to be unique.

IOHanalyzer takes as input benchmarking data sets, generated, e.g., by IOHexperimenter, through the COCO platform, or through the Nevergrad environment. Of course, users can also use their own experimentation platform (IOHanalyzer has a flexible interface for uploading custom `csv`-files). IOHanalyzer provides an evaluation platform for these performance traces, which allows users to choose the performance measures, the ranges, and the precision of the displayed data according to their needs. In particular, IOHanalyzer supports both a fixed-target and a fixed-budget perspective, and allows various ways of aggregating performances across different problems (or problem instances). In addition to these performance-oriented analyses, IOHanalyzer also offers statistics about the evolution of non-static algorithmic components, such as, for example, the hyperparameters suggested by a self-adjusting parameter control scheme.

To illustrate the functionality of IOHanalyzer, we highlight a selected subset of the available functionality (which is listed in more detail in Tables 3.1 and 3.2).

Fixed-Target Results ▶ Single Function ▶ Data Summary: This setting provides basic statistics on the distribution of the fixed-target running time, which are grouped in 3 different tables:

- Table `Data Overview:` This table provides a high-level summary of the currently loaded data set. It simply summarizes the range of *function values* observed in the data set, offering users a quick overview of the quality of the solutions that were evaluated by the algorithms by showing summary statistics of the function values found for the selected function.

- Table `Runtime Statistics at Chosen Target Values:` A screenshot of this table is given in Figure 3.3. The user can set the range and the granularity of the results in the box on the left. The table shows fixed-target running times for evenly spaced target values.[2] More precisely, the table provides the success rate and the number of successful runs as defined in Eq. (2.14), the sample mean, median, standard deviation, the sample quantiles: $Q_{2\%}, Q_{5\%}, \ldots, Q_{98\%}$, and the *expected running time* (ERT) as defined in Eq. (2.17). The user can download this table in `csv` format, or as a LaTeX table.

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

Table 3.1: *Fixed-budget* functionality of IOHanalyzer (v0.1.7).

Section	Group	Functionality	Description
Single Function	Data Summary	*Data Overview*	The minimum and maximum of running times for selected algorithms.
		Target Value Statistics	The mean, median, quantiles of the function value at a sequence of budgets controlled by $B_{\min}, B_{\max}$ and ΔB.
		Target Value Samples	The function value samples at an evenly spaced sequence of budgets controlled by $B_{\min}, B_{\max}$ and ΔB.
	Expected Target Value	*Expected Target Value: single function*	The progression of expected function values over budgets, whose range is controlled by the user.
	Probability Density Function	*Histogram*	The histogram of the function value a user-chosen budget.
		Probability Density Function	The probability density function (using the Kernel Density Estimation) of the function value at a user-chosen budget.
	Cumulative Distribution	*ECDF: single budget*	On **one** function, the ECDF of the function value at **one** budget specified by the user.
		ECDF: single function	On **one** function, ECDFs aggregated over **multiple** budgets.
		Area Under the ECDF	On **one** functions, the area under ECDFs of function values that are aggregated over **multiple** budgets.
	Algorithm Parameters	*Expected Parameter Value*	The progression of expected value of parameters over the **budget**, whose range is controlled by the user.
		Parameter Statistics	The mean, median, quantiles of recorded parameters at an evenly spaced sequence of budgets controlled by $B_{\min}, B_{\max}$ and ΔB.
		Parameter Sample	The sample of recorded parameters at an evenly spaced sequence of budgets controlled by $B_{\min}, B_{\max}$ and ΔB.
	Statistics	*Hypothesis Testing*	The two-sample Kolmogorov-Smirnov test applied on the running time at a target value for each pair of algorithms. A partial order among algorithms is obtained from the test
Multiple Functions	Data Summary	*Multi-Function Statistics*	Descriptive statistics for all functions at a single target value.
		Multi-Function Hitting Times	Raw hitting times for all functions at a single target value.
	Expected Target Value	*Expected Target Value: all functions*	The same as above expect that the expected function values are grouped by functions and the range of budgets are automatically determined.
		Expected Target Value: Comparison	The expected function value at the largest budget found on each function is plotted against the function ID for each algorithm.
	Deep Statistics	*Ranking per Function*	Per-function statistical ranking procedure from the Deep Statistical Comparison Tool (DSCTool) [69].
		Omnibus Test	Use the results of the per-function ranking to perform an omnibus test using DSC.
		Posthoc comparison	Use the results of the omnibus test to perform the post-hoc comparison.
	Ranking	*Glicko2-based ranking*	For each pair of algorithms, a function value at a given budget is randomly chosen from all sample points in each round of the comparison. The glicko2-rating is used to determine the overall ranking from all comparisons.

Table 3.2: *Fixed-target* functionality of IOHanalyzer (v0.1.7).

Section	Group	Functionality	Description
Single Function	Data Summary	*Data Overview*	The best, worst, mean, median values and success rate of selected algorithms.
		Runtime Statistics	The mean, median, quantiles, success rate and ERT at an evenly spaced sequence of targets controlled by $f_{\min}, f_{\max}$ and Δf.
		Runtime Samples	The running time sample at an evenly spaced sequence of targets controlled by $f_{\min}, f_{\max}$ and Δf.
	Expected Runtime	*ERT: single function*	The progression of ERT over targets, whose range is controlled by the user.
		Expected Runtime Comparisons	Comparing the ERT values of selected algorithms at pre-computed targets across all problem dimensions on a chosen problem.
	Probability Mass Function	*Histogram*	The histogram of the running time at a target specified by the user on **one** function.
		Probability Mass Function	The probability mass function of the running time at a target specified by the user on **one** function.
	Cumulative Distribution	*ECDF: single target*	On **one** function, the ECDF of the running time at **one** target specified by the user.
		ECDF: single function	On **one** function, ECDFs aggregated over **multiple** targets.
	Algorithm Parameters	*Expected Parameter Value*	The progression of expected value of parameters over **targets**, whose range is controlled by the user.
		Parameter Statistics	The mean, median, quantiles of recorded parameters at an evenly spaced sequence of targets controlled by $f_{\min}, f_{\max}$ and Δf.
		Parameter Sample	The sample of recorded parameters at an evenly spaced sequence of targets controlled by $f_{\min}, f_{\max}$ and Δf.
	Statistics	*Hypothesis Testing*	The two-sample Kolmogorov-Smirnov test applied on the running time at a target value for each pair of algorithms. A partial order among algorithms is obtained from the test.
Multiple Functions	Data Summary	*Multi-Function Statistics*	Descriptive statistics for all functions at a single target value.
		Multi-Function Hitting Times	Raw hitting times for all functions at a single target value.
	Expected Runtime	*ERT: all functions*	The progress of ERTs are grouped by functions and the range of targets are automatically determined.
		Expected Runtime Comparisons	The ERTs at the best target found on each function (one fixed dimension) is plotted against the function ID for each algorithm.
	Cumulative Distribution	*ECDF: all functions*	On **all** functions, ECDFs aggregated over **multiple** targets.
	Deep Statistics	*Ranking per Function*	Per-function statistical ranking procedure from the Deep Statistical Comparison Tool (DSCTool) [69].
		Omnibus Test	Use the results of the per-function ranking to perform an omnibus test using DSC.
		Posthoc comparison	Use the results of the omnibus test to perform the post-hoc comparison.
	Ranking	*Glicko2-based ranking*	For each pair of algorithms, a running time value at a given target is randomly chosen from all sample points in each round of the comparison. The glicko2-rating is used to determine the overall ranking from all comparisons.
	Portfolio	*Contribution to portfolio (Shapley-values)*	Calculate the approximated Shapley values indicating the contribution of each algorithm to the overall portfolios ECDF.

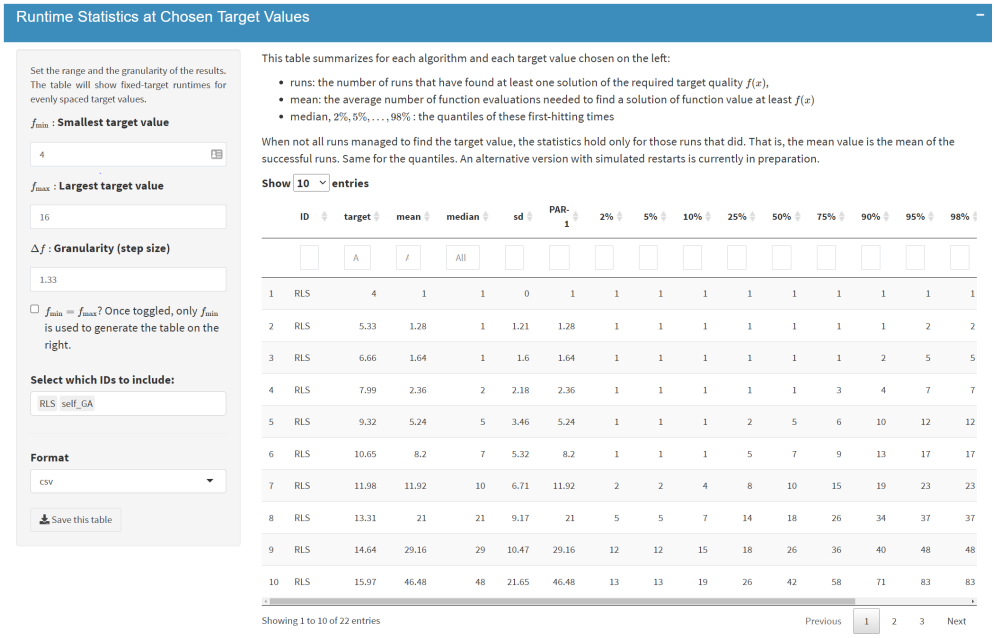


Figure 3.3: Screenshot of the data summary table of some descriptive statistics on the running time.

- Table `Original Runtime Samples:` This table uses the same principle as the `Runtime Statistics:`, but instead displays the values for each individual run. For this table, the user can choose between a "long" (all sample points are arranged in a column) and a "wide" format (all sample points are arranged in a row).

Fixed-Target Results ▶ Single Function ▶ Expected Runtime: An interactive plot illustrates the fixed-target running times. An example of this plot is shown in Figure 3.4. The interactive plot can be adjusted in the menu on the left as shown in the figure. These options include showing/hiding mean and/or median values along with standard deviations and scaling the axes logarithmically. The user selects the algorithms to be displayed as well as the range of target values within which the curves are drawn. By default, this range is set as $[Q_{25\%}, Q_{75\%}]$ of all function values measured in the data set. The displayed curves can be switched on and off by clicking on the legend on the bottom of the plot.

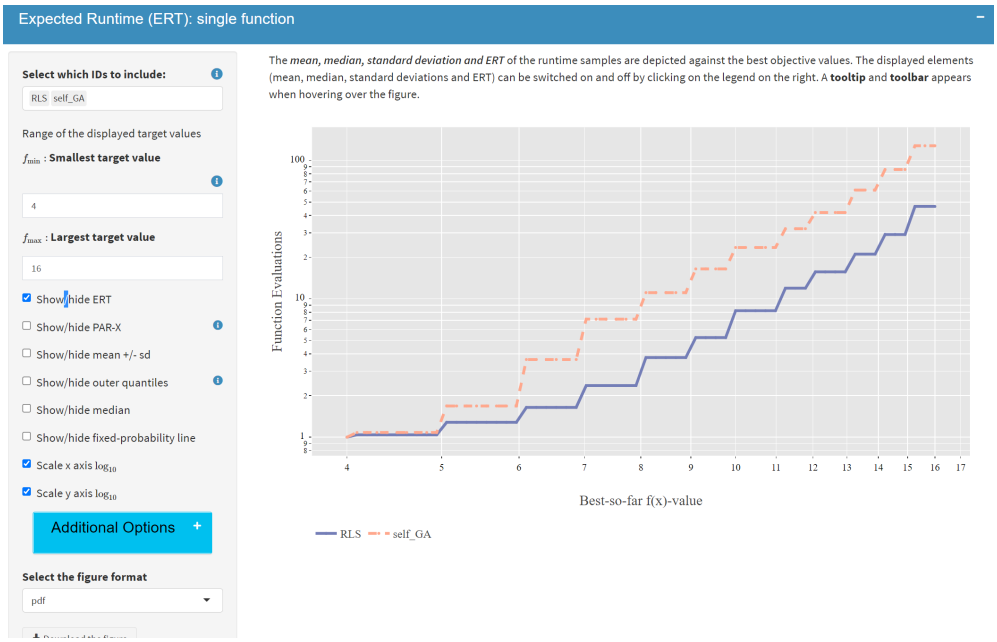
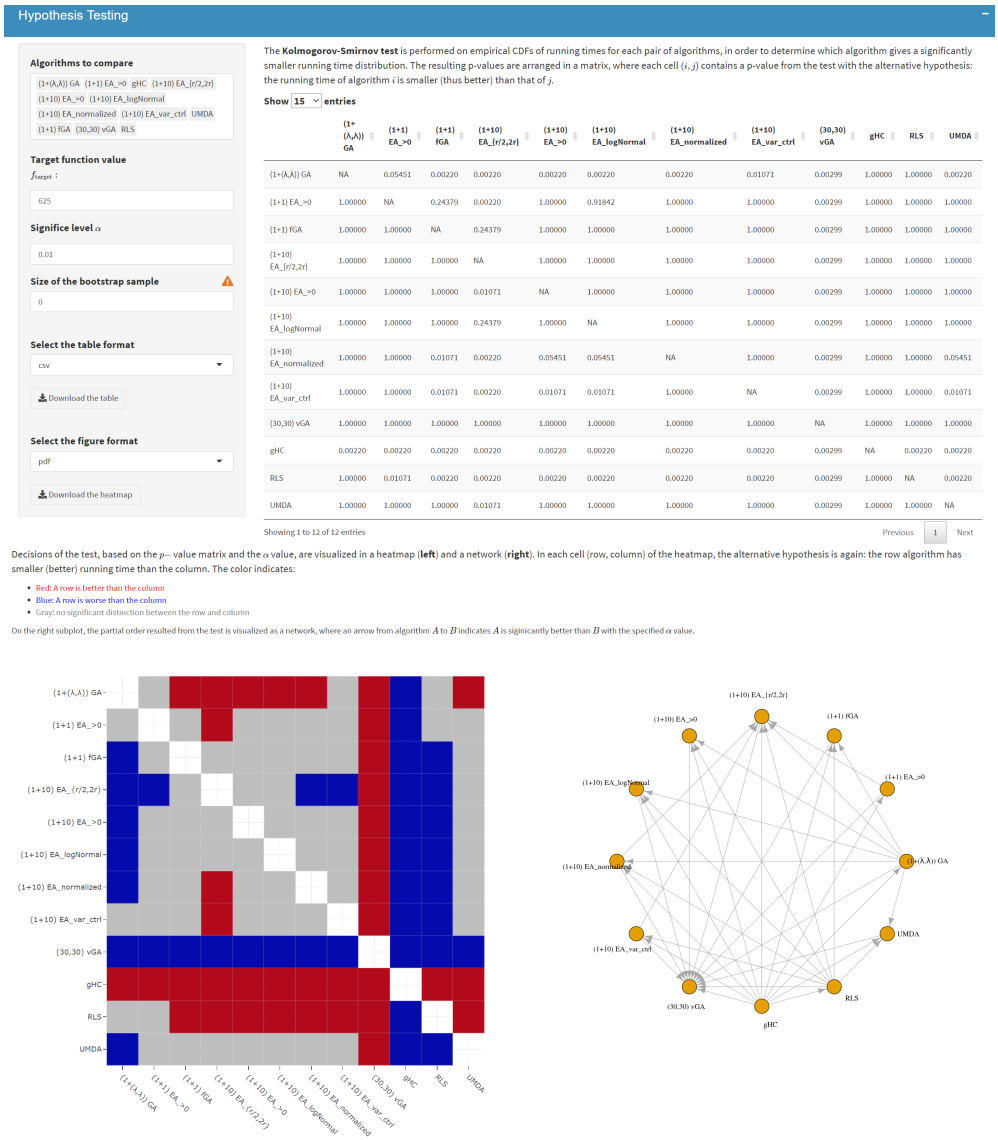


Figure 3.4: Screenshot of the expected running time plot.

Fixed-Target Results ▶ Single Function ▶ Algorithm Parameters: One of the key motivations to build IOHprofiler was the ability to analyze, in detail, the evolution of control parameters which are adjusted during the search. Such dynamic parameters can be found in most state-of-the-art heuristics. While in numerical optimization a non-static choice of the search radius, for example, is needed to eventually converge to a local optimum, dynamic parameters are also more and more common in discrete and mixed-integer optimization heuristics [119, 60]. In the fifth group of fixed-target results for a single function, the evolution of the parameters is linked to the quality of the best-so-far solutions that have been evaluated. In the experimentation (i.e., data generation) phase, the user selects which parameters are logged along with the evaluated function values. These values are then automatically detected by IOHanalyzer and can be chosen in this group for analysis.

As with the interactive plots on expected running time, the user can choose the range of targets, which parameters and algorithms to plot, and the scale (either logarithmic or linear) of x- and y-axis. We omit the example for parameters as the GUI is similar to the one in Figure 3.4. As with "Fixed-Target Results ▶ Single Function ▶ Data Summary", this subsection also provides for each parameter tables of descriptive

statistics (sample mean, median, standard deviation, and some quantiles) as well as the original parameter values.



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

Figure 3.5: Screenshot of the multiple testing procedure applied on all 12 reference algorithms on function $f1$ and dimensionality 625. The table shows the p-values resulting from the pairwise KS-test between each pair of algorithms. Then, based on the $\alpha = 0.01$, the resulting hypothesis-rejections are shown in both the matrix-plot and the network.

Fixed-Target Results ▶ Single Function ▶ Statistics: To address the robustness of empirical comparisons, the samples from all algorithm must undergo a proper statistical test procedure [103]. In IOHanalyzer, a standard multiple testing procedure is implemented to compare the fixed-target running time for each pair of algorithms on a single function, for which the well-known Kolmogorov-Smirnov [161] test is applied to the ECDFs of running times. Moreover, the Bonferroni procedure [19] is used to correct the p-value in multiple testing. To demonstrate this functionality, we show, in Figure 3.5, the testing outcome of a data set from running 12 reference algorithms.[3] It can be loaded to the web-based GUI by selecting the PBO data set in the "upload data" section. The data set comprises the results of the experimental study described in [63]. on the PBO problem set from [63], instead of the exemplary two-algorithm data set used previously. Here, the test is conducted across all 12 algorithms on function $f1$ and dimensionality 64 with a confidence level of 0.01. The result of this procedure is illustrated by a table of pairwise p-values, a color matrix of the statistical decision, and a graph depicting the partial order induced by the test (i.e., an arrow pointing from Algorithm 1 to Algorithm 2 is to be read as Algorithm 1 dominating Algorithm 2 with statistical significance). As with all tables and figures in IOHanalyzer, these can be downloaded in several formats, including `tex` and `csv` for tables and `pdf` and `eps` for figures.

Fixed-Target Results ▶ Multiple Functions ▶ Cumulative Distribution: In this group, ECDFs of running times are aggregated across multiple functions, as defined in Eq. (2.19). This functionality is illustrated in Figure 3.6: a table of precalculated target values are provided for each function (all test functions are included by default). This table of targets can easily be edited directly in the GUI, or by a downloading-editing-uploading procedure (which should, of course, not change the format of the tables, just the values). Note that for these ECDF-figures, the corresponding Area Under the Curve (AUC) can also be calculated to get a single value for each algorithm. These AUC-tables are available in the same tab as the ECDF plot.

3.1.3 IOHproblems: Benchmark Suites

Within the IOHprofiler environment, we have the ability to access a wide variety of benchmark problems. Because of the emphasis on extensibility, several benchmark suites have been integrated and used in works related to this thesis. In this section,

[3] This data set is available at https://github.com/IOHprofiler/IOHdata/blob/master/iohprofiler/2019gecco-ins1-11run.rds

3.1. IOHprofiler

Figure 3.6: Screenshot of aggregated ECDF curve across multiple functions and targets.

we provide an overview of the available suites and highlight some of the most interesting results obtained when performing several benchmarking studies. Throughout this thesis, we place a heavy emphasis on the Black-Box Optimization Benchmarking (BBOB) suite [96]. As such, we will discuss this suite in detail in Section 3.2.

The PBO-suite and W-model The Pseudo-Boolean Optimization (PBO) suite was the first set of benchmarking problems integrated in IOHprofiler. This suite consists of 25 problems, ranging from the well-known OneMax to N-Queens. Included in this suite is a set of problems based on the W-model problem generator proposed in [258] (the W-model generator is also available in IOHprofiler). The full description of the problems, as well as results from several well-known optimizers, are available in [63]. This paper also highlights the benefits of flexible parameter tracking, allowing for a detailed analysis of self-adaptive parameters of several GA-variants. Later, another well-known generator of pseudo-Boolean problems (MK-landscapes) was integrated [225].

Submodular Optimization As part of the 2023 Competition on Submodular Optimization at the Genetic and Evolutionary Computation Conference (GECCO), we integrated four different types of (constrained) submodular optimization problems into IOHprofiler: Maximum Coverage, Maximum Influence, Maximum Cut, Packing While Traveling. For each of these problems, we provide several underlying graph instances, leading to a total of 66 functions. Since these functions can either be treated as unconstrained single-objective (by integrating the constraint into the objective function), constrained single-objective or multi-objective, they allow for an interesting comparison of different types of solvers [177, 80].

Star Discrepancy Discrepancy measures are designed to quantify how regularly a point set is distributed in a given space. Among the many discrepancy measures, the most common one is the L_∞ star discrepancy (referred to as simply *star discrepancy* from here on), which is especially important in numerical integration [129, 102], but also in e.g. the design of experiments [207] or in the context of one-shot optimization [21, 20].

The computation of star discrepancy of a set P of n points in a d-dimensional unit-hypercube can naturally be formulated as a real-valued optimization problem on $[0, 1)^d$. However, there is an equivalent discrete formulation [179] on the space $[0, n)^d$. These two formulations of the same underlying problems again allow for potential

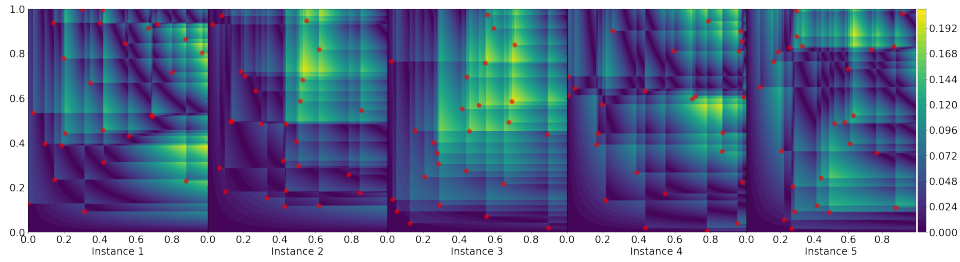


Figure 3.7: Discrepancy values of the first 5 instances of the 2-dimensional version of F31: Uniform sampler with 20 points. The red points indicate the originally sampled points.

comparisons of very different optimization algorithms on the same underlying problem.

To create specific problems for benchmarking, we vary the number of samples in our point sets, as well as the sampler used to create those points. By using Uniform, Halton and Sobol samplers for $n \in \{10, 25, 50, 100, 150, 200, 250, 500, 750, 1000\}$, resulting in 30 problems in IOHexperimenter, with function IDs 30–59. Each problem can additionally be scaled to arbitrary dimensionality, and different instance are obtained by using different seeds for the samplers.

To illustrate the structure of the continuous version of the star discrepancy problems, we show several landscapes in Figures 3.7 and 3.8. Figure 3.7 illustrates the local star discrepancy values for an instance in $d = 2$. We can already observe that the problem of maximizing the local L_∞ star discrepancy bears two important challenges: (1) it is a *multimodal* problem, i.e., there can be several local optima in which the solvers can get trapped (this problem becomes worse with increasing dimensionality); (2) there are *sharp discontinuities* in the local discrepancy values. Slightly increasing one parameter can result in a point falling inside the considered box, causing a $1/|P|$ difference in the local star discrepancy value. Figure 3.8 shows that the problem structure also depends strongly on the point set considered.

To test the performance of our black-box optimization algorithms, we look at the final solutions found by each of the 8 selected optimizers from the Nevergrad library [200]. In order to create a fair comparison, we transform the original discrepancy values to a relative measure, based on the bounds found by two algorithms specifically designed for star discrepancy computation: TA [84] and DEM [59]. Specifically, we

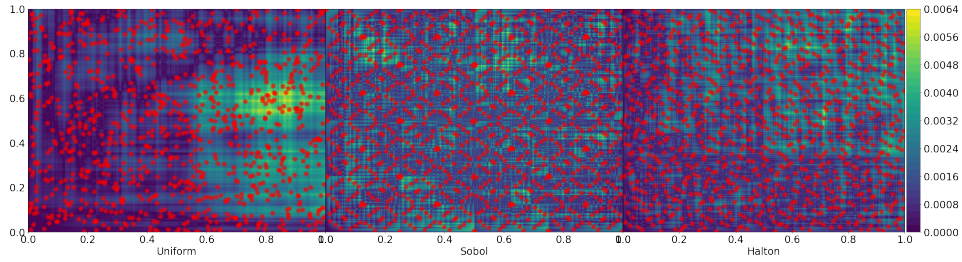


Figure 3.8: Comparison of the 3 samplers for 1000 points in 2D, and the corresponding discrepancy landscapes (instance 1 for F39, F49, and F59 in IOHexperimenter respectively).

consider the following measure:

$$R(x) = \frac{OPT(x) - f(x)}{OPT(x)} \to \min,$$

where $f(x)$ is the final value found after the optimization run, and $OPT(x)$ is the bound calculated by the parallel DEM or TA, depending on the instance size.

Using this relative measure, we can compare the final solutions found by each optimization algorithm across a set of different values of n and d. This is visualized in Figure 3.9. In this figure, we see that the SPSA algorithm is clearly performing poorly, while Random Search seems to be competitive with, if not superior to, all other algorithms for every scenario. In addition to the ranking between algorithms, we also note a clear increase in problem difficulty as the dimensionality increases. Conversely, the number of samples seems to have a rather limited impact on the relative difficulty. This suggests that the structure of the point set has little influence on the performance of the optimizers.

Overall, these results suggest that the star discrepancy benchmarks are challenging for current black-box optimization approaches since random search outperforms every other algorithm considered.

Strict Box Constrained BBOB The final set of benchmark problems we consider is the suite of strict box-constrained BBOB problems (SBOX), which was introduced as part of the workshop at GECCO 2023 under the same name. This suite does not contain any new problems, but instead modifies the BBOB suite by adding strict box constraints ($[-5, 5]^d$, evaluating outside the domain yields a value of ∞). In addition, it modifies some of the instance generation procedures to allow the optimum to be

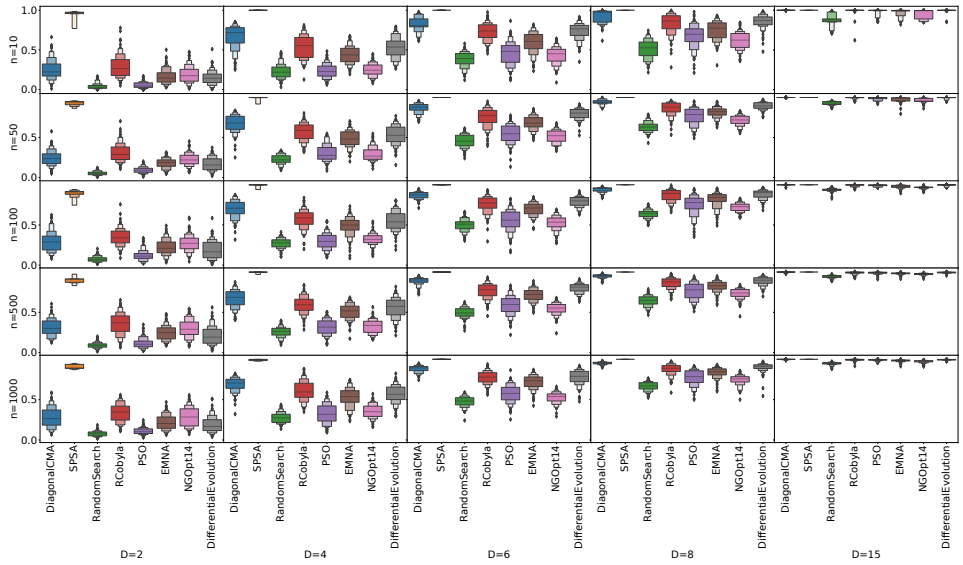


Figure 3.9: Relative final discrepancy value found by each of the used optimizers. Values of 0 correspond to finding the optimal solution, while 1 corresponds to the worst achievable value (0 discrepancy). Box-plots are aggregations of 10 runs on 10 instances, all for the uniform sampler.

located closer to the bounds of the domain. Submissions to this workshop opened up some insightful discussions about the need for box-constraint handling and its relations to reproducibility [242, 167, 57, 27, 114].

3.1.4 Benchmark Data: OPTION

As evident from the large number of benchmark suites and optimization problems, rigorous benchmarking has the potential to generate vast amounts of data. Within the current structure of IOHprofiler, we integrate data for hundreds of algorithms, achieved by integrating with several other platforms. We incorporate COCOs historical data collected from over a decade of workshops, Nevergrads ever-expanding set of benchmarks and optimizers, our own competitions, and much more. By making this data easily accessible in IOHanalyzer, comparisons to existing data are made easier. However, this technique does not fully exploit all information present in this benchmark data. To gain more insight from the existing data, a structured way of storing and querying is required.

One well-established solution to the problem of managing such complex data pools

is the use of ontologies. Ontologies are specifications of a shared conceptualization of data from distributed and heterogeneous systems and databases, and as such, they enable data interoperability, efficient data management and integration, and cross-database search. For this purpose, we collaborated on the creation of the **OPTimization Algorithm Benchmarking ONtology (OPTION)**. This ontology was designed with the main goal of standardizing and formalizing knowledge from the domain of benchmarking optimization algorithms, where an emphasis was put on the representation of data from the benchmark performance space. OPTION offers a comprehensive description of the domain, covering the benchmarking process, as well as the core entities involved in the process, such as optimization algorithms, benchmark problems, evaluation measures, etc. We summarize some of the key design choices for the design of OPTION.

Performance Data There exist many different benchmarking platforms for optimization, each with its own way of storing performance and algorithm data. Three main approaches to the storage of performance data are described below:

- `csv`**-based**: The data is stored as a single file per experiment in a `csv`-based format, where each column represents a performance measure or other meta-information. An example is the format used in Nevergrad [200]. This allows for storing data on many different functions/problems into a single file, with the drawback that the granularity of the data is often limited.

- **Textfile-based**: The data is separated into a single file per function/problem, where the meta-information is delimited in some way, followed by the performance information. This format is easily extendable and human-readable, but it can be hard to work with when files become large. An example of this format is used by the SOS platform [38].

- **COCO/IOH-like**: The data is separated into multiple files and folders: generally, folder structure splits along algorithms and functions/problems. Each folder then contains a file with meta-information about the runs, with links to the files where the raw performance data is stored. This structure makes it easy to find the data sought, but the different links to the files can be an obstacle for practitioners who are not used to this format. Variants of this data format are used by COCO [95] and IOHprofiler [62].

As mentioned, each of these data formats has its advantages and disadvantages. While

there are some commonalities between different methods, the particularities in handling meta-data make interoperability of the data from different sources challenging. Furthermore, these differences lead to more limited post-processing functionalities available to the users of these platforms since they are only compatible with those tools that support their particular data format. While these tools are slowly becoming more interoperable, this process requires significant effort from the developers of the individual tools to make sure all data formats are fully supported. A common data structure would be useful to the benchmarking community to avoid each developer having to do this individually.

In order to create such a common structure, it is crucial to identify the core components of performance data that would be needed in the analysis. Then, an explicit conversion needs to be made for each data format, which extracts these properties from the original data format. Some previous efforts show that it is possible to make such conversions and then jointly analyze data which has been originally stored in different data formats. In particular, performance data from Nevergrad, COCO, IOHexperimenter, and from the SOS platform can be conveniently analyzed through the IOHanalyzer [255].

Problem Landscape Data In addition to the performance data discussed above, we also integrate information about the problems themselves, in the form of ELA features (see Section 2.5). In particular, we use the dataset [202] containing precalculated ELA features for the instances of the BBOB problems we consider.

Domain challenges for data integration A source of complexity in recording performance data from black-box optimization is that we typically do not use a single performance measure. Instead, we are interested in analyzing algorithm performance from different perspectives: small vs. large budgets, the time needed to identify solutions that meet specific quality criteria, the robustness of the algorithm in search and performance space, etc. [8].

To enable such detailed analyses, researchers often record performance data in a multi-dimensional fashion, spanning at least the time elapsed (measured in terms of CPU time and/or function evaluations), solution quality, and robustness. We may also be interested in how dynamic parameters evolve during the optimization process, in which case we record their values along with the performance data. Both requirements add another level of complexity to the data formats and may explain why they differ so much in practice.

There are several other factors that further complicate the interoperability and re-usability of publicly available performance data from different benchmarking experiments:

- Most black-box optimization algorithms are, in fact, families of various algorithm instances. They can be selected by specifying the (hyper-)parameters of the algorithm and/or the operators (e.g., one may speak of Bayesian optimization regardless of the internal optimization algorithm that is used to search the surrogate model, or one may use different acquisition functions, different techniques to build the surrogate, etc.). Different configurations can lead to drastically different search behavior (and hence performance), and it is crucial to associate the recorded data to the appropriate algorithm instance and not only to an algorithm family. However, this is not an easy task, as it can happen that essentially the same algorithm is published under different names (see [35, 218] for recent examples and a discussion, respectively).

- A similar issue appears on the problem side. Different instances of the same problem can be of different complexity, and it is not always clear which problem instances were used within a given benchmark study. In addition, some benchmarking suites automatically rotate, shift, permute, or translate the problem instances, to test specific unbiasedness characteristics and the generalizability of the algorithms. Other suites do not do this (e.g., because the variable order or absolute values carry some meaning) but still refer to problems of different complexity under the same name. As for the algorithms, we can also have the same problem appear under different, possibly multiple, names. The OneMax problem, for example, is sometimes called CountingOnes, OnesMax, the Hamming distance problem, or Mastermind with 2 colors. All these names refer to the same problem.

Identifying such issues cannot (as of yet) be done automatically but require human expertise to annotate the data correctly. While this requires a significant amount of effort for the large amounts of currently available benchmarking data, we aim for the procedure to convert from different data formats to be automated where possible (e.g., by involving the authors of the different benchmarking platforms) and clearly structured where not. In the future, this would then become second nature when introducing a new algorithm / problem / experimental setup, allowing the data ontology to grow organically. The creation of reproducible and readily available data will eventually benefit the optimization community as a whole, so the efforts invested to

achieve this goal would be very much worthwhile.

The Knowledge Base The OPTION ontology contains the semantic model, represented in a formal and standardized way. The OPTION Knowledge Base (KB), on the other hand, leverages the power of the ontology and holds the actual data that has been semantically annotated. This KB is created by annotating data from COCO and Nevergrad for performance data, and ELA features for the BBOB functions for the landscape data.

OPTION in IOHprofiler To query the OPTION KB, SPARQL queries can be used. However, SPARQL queries can become very complex and sometimes are seen as a bottleneck to the broader acceptance of Semantic Web technologies. We recognize that SPARQL query construction is an error-prone and time-consuming task that requires expert knowledge of the whole stack of semantic technologies. Even experts find it sometimes challenging to query semantic data since they first must get familiar with the data annotation schemes or the structure of the knowledge base.

To facilitate the use of the OPTION ontology, we provide a simple GUI that can be used to gain access to performance data without needing to write SPARQL queries. This interface is connected directly to IOHanalyzer, which enables the loaded data to be used directly in performance analysis and visualization, and even be compared to data that might not yet be included in OPTION or to user-submitter performance data. Furthermore, the GUI provides access to a parameterized search process, which can be used without any underlying knowledge about the used semantic data model. Users can express their query by selecting from several drop-down options, which specify the required information, such as suite, function, algorithm, etc., and load the corresponding performance data to analyze. This interface is shown in Figure 3.10. While this interface is static, it illustrates the power of integrating the ontology into IOHanalyzer: users without any background knowledge can use it to gain insight into the performance of the selected algorithms/functions.

Additionally, this interface can be easily expanded based on the community's wishes. To illustrate this potential, we created another entry point into OPTION, which can be used to load all performance data which originated in a specified paper. To this end, the user selects a paper by its title, which then populates the relevant information about the used algorithms and functions in that study. By loading this pre-selected data, the user has full access to the performance data of the selected study, which they can then investigate in more detail by making use of the visualiza-

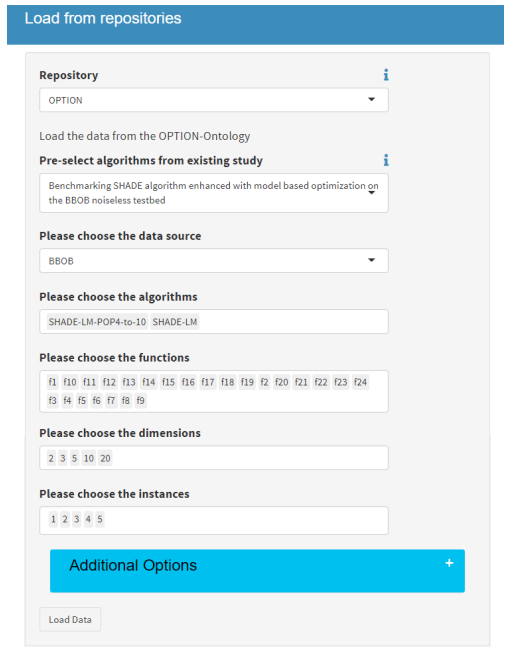


Figure 3.10: The interface of the OPTION-ontology queries within IOHanalyzer (version 1.6.3), available at https://iohanalyzer.liacs.nl/.)

tions within IOHanalyzer. This type of interactive analysis then allows the user to look at the data from different perspectives and to compare it to other algorithms.

3.2 Benchmark Suites: BBOB

Originally developed as part of the COCO platform [95], this set of 24 continuous, single-objective, noiseless problems has become the de facto standard when comparing numerical optimization algorithms for continuous optimization problems. These 24 functions can be separated into five core classes based on their global properties, as seen in Table 2.1. In fact, the suite has been designed to ensure that each problem poses a different 'challenge' to the optimization algorithm. For example, the ellipsoidal function adds ill-conditioning the the sphere problem, so when an algorithm performs well on the latter but not the former the designer gains insight into the deficiencies of their algorithm.

For each BBOB function, arbitrarily many problem instances can be generated by applying transformations to both the search space and the objective values [96] – such

mechanism is implemented internally in BBOB and controlled via a unique identifier (also known as IID) which defines the applied transformations (e.g. rotation matrices). For most functions, the search space transformations are made up of rotations and translations (moving the optimum, usually uniformly in $[-4, 4]^d$).[4] Since the objective values can also be transformed, the performance measures used generally are relative to the global optimum value to allow for comparison of performance between instances, typically in logarithmic scale.

This instance generation method is certainly useful for many applications. For instance, different instances have been considered to enable comparisons between stochastic and deterministic optimization algorithms [189], since using a different instance can in some way be considered as changing the initialization of the deterministic algorithm. It also enables an algorithm designer to test for some invariance properties, particularly with regard to scaling of the objective values, and rotation of the search space [96]. Recently, instances have also been used in a more machine learning (ML) based context, e.g., methods for algorithm selection are trained and tested based on different sets of instances [135].

While creation of different instances of the same function has been very useful to many benchmarking setups, *the underlying assumption that the function properties are preserved is a rather strong one.* For a simple sphere function, the impact of moving the optimum throughout the space can be reasoned about relatively easily, but the impact of the more involved transformation methods on more complex functions is challenging to quantify directly. In addition, the fact that black box optimization problems are in practice often considered to be box-constrained [8], while BBOB was originally designed based on unconstrained function definitions [95], introduces the possibility that some transformations might change key aspects of the function. In fact, it has been shown that the properties of box-constrained functions captured using landscape analysis are not necessarily consistent across instances [171].

In order to analyze the resulting low-level properties of optimization problems, various features of the landscape can be computed. This falls under the field of exploratory landscape analysis (ELA) [165]. While some analysis into the ELA features across instances of BBOB problems has been previously performed [175], we extend the scope of our analysis to include a much wider range of instances. In addition, we consider several other low-level features, such as the location of the global optima, to develop an extensive understanding of the way in which instances might differ. Since

[4] While the suite is originally intended to be used for unconstrained optimization, in practice however, black box optimization functions like this are often considered to be box-constrained [8], in the case of BBOB with domain $[-5, 5]^d$.

we deal with the box-constrained version of the BBOB problems, we also investigate the performance of a set of algorithms, in order to verify that these algorithms perform similarly on different instances of a function – this extends the approach taken in [242].

In this section, we investigate whether the problem characteristics (in the form of ELA features) are preserved across different problem instances and whether the first few (most commonly used) instances are representative of the wider set. In addition, we study whether statistically significant differences in algorithm performance can be found between different instances of the same BBOB problem.

Instance Similarity using ELA

We first focus on analyzing the problem characteristics of different BBOB problem instances based on the ELA approach. For each BBOB instance, we generate 100 sets of DoE data with 1 000 samples each using the Latin Hypercube sampling (LHS) method (so the DoEs are identical for all instances), in order to obtain the ELA feature distribution. We consider a total of 68 ELA features that can be computed without additional sampling, using the package `flacco` [127, 125] and the pipeline developed in [148]. Three of the ELA features, which resulted in the same value across all instances, are deemed not informative and hence dropped out; this means that a final set of 65 ELA features is being considered here.

Comparing Distributions. To investigate how comparable the characteristics of different problem instances are, we carry out the (pairwise) two-sample Kolmogorov-Smirnov (KS) test [161], with the null hypothesis that the ELA distribution is similar in both (compared) problem instances. This results in $\frac{500 \cdot 499}{2} = 124\,750$ comparison pairs per ELA feature. To account for multiple comparisons, we apply the Benjamini-Hochberg (BH) method [12]. To get an overview of differences for a particular ELA feature of each BBOB function, we compute the average rejection rate of the aforementioned null hypothesis of the KS test by aggregating all problem instances (i.e. number of rejections divided by total number of tests). In other words, it shows the fraction of tests which rejected each combination of ELA feature and BBOB function, as shown in Figure 3.11.

On the $5d$ problems, we notice that some features clearly differ between instances, in particular the `ela_meta.lin_simple.intercept`. However, this does not necessarily indicate that all instances should indeed be considered to be different since, as illustrated in [253], some features including this linear model intercept are not invariant to scaling of the objective function. For some other features, such as those

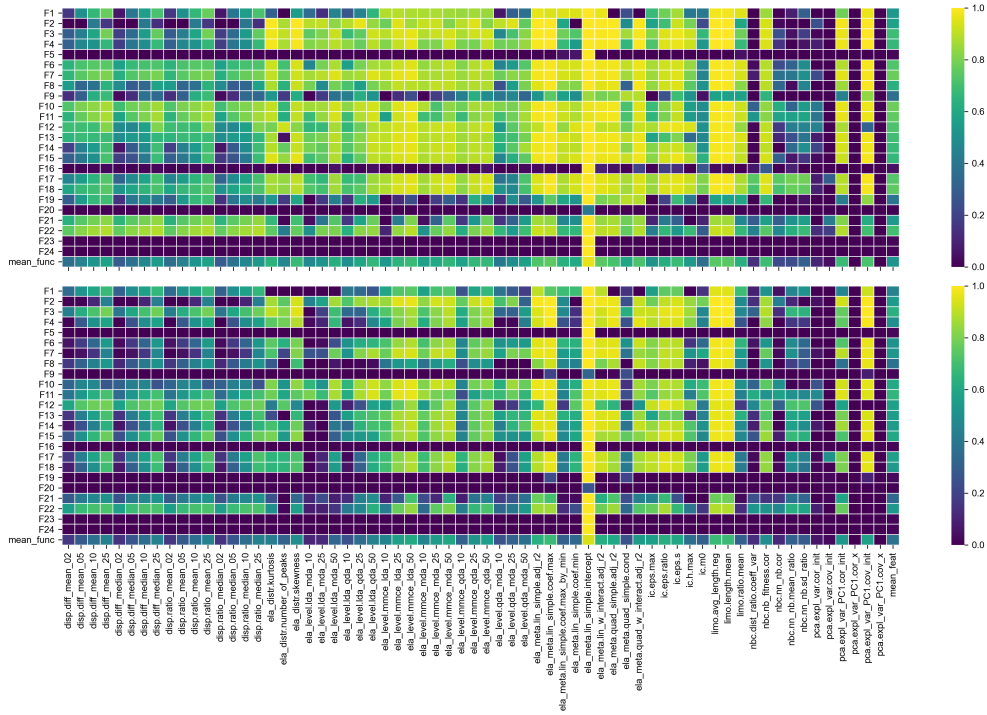


Figure 3.11: Average rejection rate of null hypothesis *distribution of ELA feature between instances is similar*, aggregated over 500 BBOB problem instances in $5d$ (top) and $20d$ (bottom). A lighter color represents higher rejection rate. An extra row (bottom) for the mean over all BBOB functions and an extra column (right) for the mean over all ELA features is added in each heatmap.

related to the principal component analysis (PCA), we notice that barely any test rejections are found. This is largely explained by considering that this feature set is built primarily on the the samples in the search-space, which are identical between instances (the same 100 seeds are used in the calculations for each instance). While the objective values still have an influence on some of the PCA-features, their impact is relatively minor. For the remaining sets of features, we see some commonalities on a per-function basis. Functions F5 (linear slope), F16 (Weierstrass), F23 (Katsuura), and F24 (Lunacek bi-Rastrigin) show no difference between instances.

It is worthwhile to point out that even for a simple function such as F1 (sphere), many features differ between instances. Since translation is the only transformation applied in F1 [96], which (uniformly at random) moves the optimum to a point within $[-4, 4]^d$, it is clear that the high-level function properties are preserved. If the problem

is considered unconstrained, this transformation would indeed be a trivial change to the problem. However, since for ELA analysis, we are required to draw samples in a bounded domain, we have to consider the problems as box-constrained, and thus moving the function can have a significant impact on the low-level landscape features. This might explain why many ELA-features differ greatly across instances on the sphere function.

On the other hand, the same overall patterns can be seen in $d = 20$ as on $d = 5$, albeit with a reduced magnitude. Moreover, functions F9 (Rosenbrock), F19 (Composite Griewank-Rosenbrock), and F20 (Schwefel) now barely show any statistical difference between instances.

Dimensionality Reduction. In addition to the statistical comparison approach, we visualize the ELA features in a $2d$ space using the t-distributed stochastic neighbor embedding (t-SNE) approach [230], as shown in Figure 3.12 for features standardized beforehand by removing mean and scaling to unit variance. It is clear that most instances of each problem are tightly clustered together. Nonetheless, there are outliers, where several instances of a function are spread throughout the projected space, indicating that these instances might be less similar. This is particularly noticeable in $5d$, where several functions are somewhat spread throughout the reduced space. In $20d$, function clusters appear much more stable, matching the conclusion from the differences with regard to dimensionality in Figure 3.11. It is worthwhile to note that differences between BBOB functions are indeed *easier to be detected* in higher dimensions using ELA features, as shown in previous work [204], which matches the more well-defined problem clusters we see in Figure 3.12.

Algorithm Performance across Instances

We now analyze the optimization algorithm performances across different BBOB problem instances. Here, we consider single-objective unconstrained continuous optimization with the following eight derivative-free optimization algorithms available in Nevergrad [200] (all with default settings as set by Nevergrad): DiagonalCMA (a variant of covariance matrix adaptation evolution strategy (CMA-ES)), differential evolution (DE), estimation of multivariate normal algorithm (EMNA), NGOpt14, particle swarm optimization (PSO), random search (RS), constrained optimization by linear approximation with random restart (RCobyla) and simultaneous perturbation stochastic approximation (SPSA). We run each algorithm on each of the 500 instances of the $5d$ BBOB problems, 50 independent runs each, resulting in a total of 4.8 million

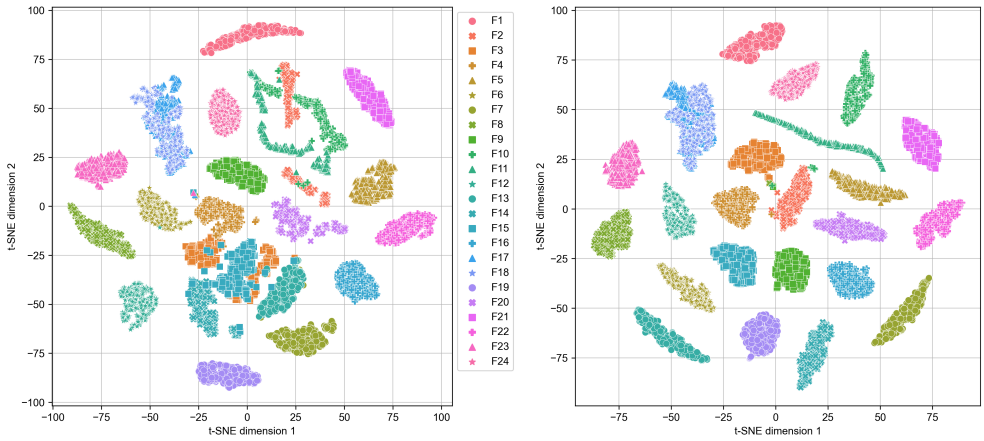


Figure 3.12: Projection of high-dimensional ELA feature space (altogether 64 features, without `ela_meta.lin_simple.intercept`) onto a $2d$ visualization for the BBOB problems in $5d$ (left) and $20d$ (right) using t-SNE approach with default settings.

($= 8 \times 24 \times 500 \times 50$) algorithm runs, each run having a budget of 10 000 function evaluations.

We consider the best function values reached after a fixed budget of 1 000 and 10 000 evaluations. Since we have 50 runs of each algorithm on each instance, we use a statistical testing procedure to determine whether there are significant differences in performance between instances – here, we use the Mann-Whitney U (MWU) test [158] with the BH correction method [12]. In addition to the pairwise testing, we consider the same procedure in a one-vs-all setting. In other words, we repeatedly compare the algorithm performances between the selected instance and the remaining (499) instances. The results are visualized in Figure 3.13, as fractions of times the test rejects the stated null-hypothesis.

We note that RS indeed seems to be invariant across instances, which is to be expected since we make use of relative performance measure (precision from the optimum) rather than the absolute function values. Furthermore, with exception of SPSA, all algorithms have stable performance on F5, F19, F20, F23 and F24, which mostly matches the results from Figure 3.11. The fact that SPSA shows differences in performance between these instances, even on F1, shows that this algorithm is not invariant to the transformations used for instance generation. This matches with previous observations that SPSA displays clear structural bias [244].

We would expect several other algorithms, specifically DiagonalCMA and DE, to

be invariant to the types of transformation used for the BBOB instance generation. However, for some problems, e.g. F12 (bent cigar), such assumption does not seem to hold. This indicates that for these problems, the instances lead to statistically different performance of these invariant algorithms. This might be explainable considering the fact that these algorithms treat the optimization problem as being box-constrained, while the BBOB function transformations make the assumption that the domain is unconstrained [95]. In addition, while the algorithms might in principle be invariant to rotation and transformation, applying these mechanisms does impact the initialization step, which can have significant impact on algorithm performance [247]. This is an intended feature of the BBOB suite, since it is claimed that *"If a solver is translation invariant (and hence ignores domain boundaries), this [running on different instances] is equivalent to varying the initial solution"* [95]. While this is true for unconstrained optimization, it is *not as straightforward* when box-constraints are assumed, as is commonly done when benchmarking on BBOB, since here changing the initialization method might significantly impact algorithm behavior.

Properties across Instances

For most functions, the general transformation mechanism consists of rotations and translations. However, in order to preserve the high-level properties, these transformations are not applied in the same manner for each problem. While translation and

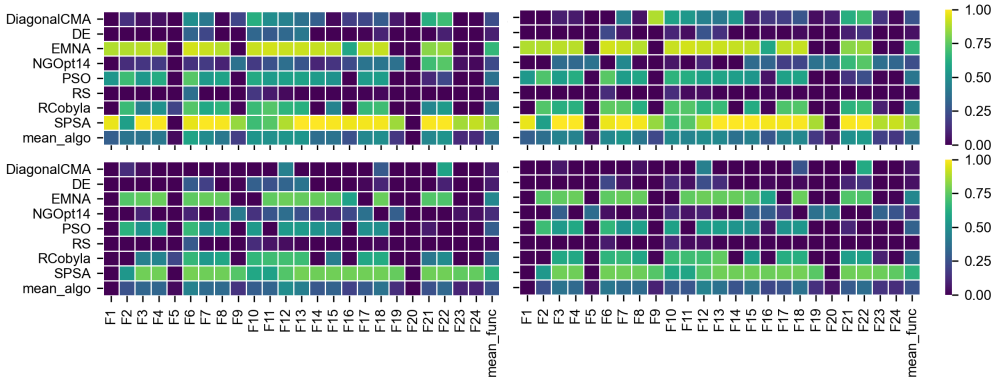


Figure 3.13: Average rejection rate of null hypothesis *algorithm performances are similar across instances*, aggregated over problem instances per function. Left and right column show results for 1 000 and 10 000 function evaluations, respectively. Top and bottom rows show pairwise and one-vs-all comparisons, respectively. Average values are shown in the last column and row of each figure.

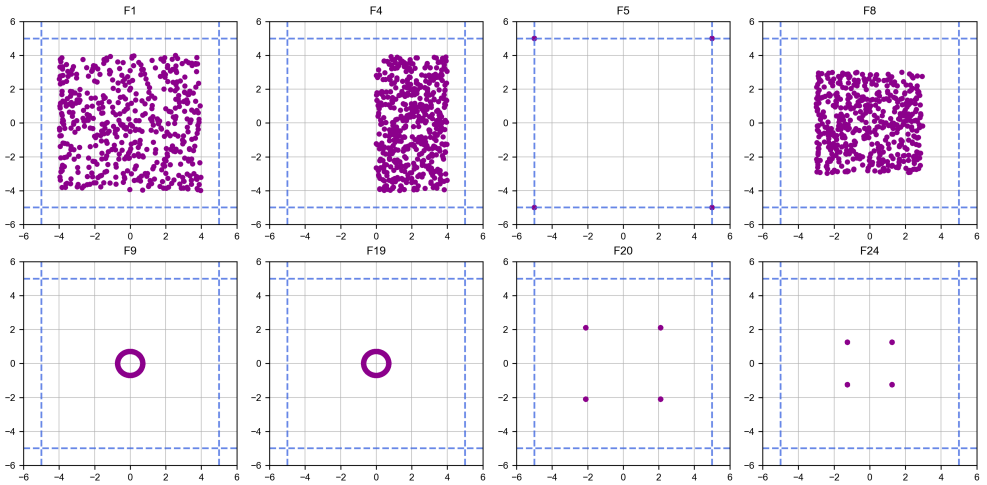


Figure 3.14: Locations of global optima of 500 instances for selected BBOB functions in $2d$. Each dot represents the optimum of a BBOB-problem instance. The remaining 16 BBOB functions (not shown here) have a similar distribution pattern as F1. Dashed lines mark the commonly used boundary of search domain, i.e., $[-5, 5]^2$.

rotation are indeed the core search space transformations, the order in which they are applied in the chain of transformation which creates the final problem can change. For simple functions such as the sphere, the transformation is straightforward (a translation only, since rotating a sphere has no impact). For other functions, such as the Schaffers10 function (F17), one rotation is applied, followed by an asymmetric function and another rotation, after which the final translation is applied. The precise transformations and their ordering is shown in [96]. *While these different transformation processes are necessary to preserve the global properties of the problems, their impact on the low-level features of the problem can not always be as easily interpreted.* As a result, the amount of difference between instances on each function is impacted by its associated transformation procedure, which can make some functions much more stable than others.

One aspect of the instances which is treated differently across problems is the location of the global optimum. By construction, for most BBOB problems, location of this optimum is uniformly sampled in $[-4, 4]^d$. This is achieved by using a translation to this location, since for the default function the optimum is located in the origin. However, for some other problems, such as the linear slope (F5), a different procedure is used. Here, we visualize *true locations of optima across the first 500 instances* of the BBOB functions in $2d$ in Figure 3.14. We note that on most functions the situation

is equivalent to that of F1, with some exceptions: (i) the asymmetric pattern for F4 (Büche-Rastrigin) stems from the even coordinates by construction being used in a different way from the odd ones; (ii) on F8 (Rosenbrock), a scaling transformation is applied before the final translation, resulting in the optimum being confined to a smaller space around the origin; (iii) for the remaining functions (F9, F19, F20, F24), construction of the problem requires a different setup (to ensure the relevant challenges of the problem remain fully inside the domain), and as such the optima will be distributed differently.

In addition to considering the location of the optimum, we aggregate the instances together, resulting in an *overview of regions of the space which are on average better performing, across multiple instances*. This highlights potential bias in the function definition, see Figure 3.15. We observe, e.g., that for sphere function (F1), the domain center has a much lower function value on average than the boundaries, which matches our intuition. This also indicates that initializing a (reasonably designed) algorithm close to the center might more likely result in good algorithm performance, as we on average directly start with better function values. For the BBOB suite overall, we see a *clear skew towards the center of the space*. While this is reasonable given the construction of problems (and the underlying implicit assumption that optimization is unconstrained), it potentially hints towards a set of *functions which are not represented in the suite*, namely those which have optima located near the boundaries, or in general give lower fitness to points close to the bounds. It is also worth mentioning that, unexpectedly, instance generation on some functions, such as the linear slope (F5) does not lead to equal treatment of dimensions, which results in consistently better regions along the boundary of one dimension only. Such a skew is clearly an artefact of a particular choice of slopes for F5.

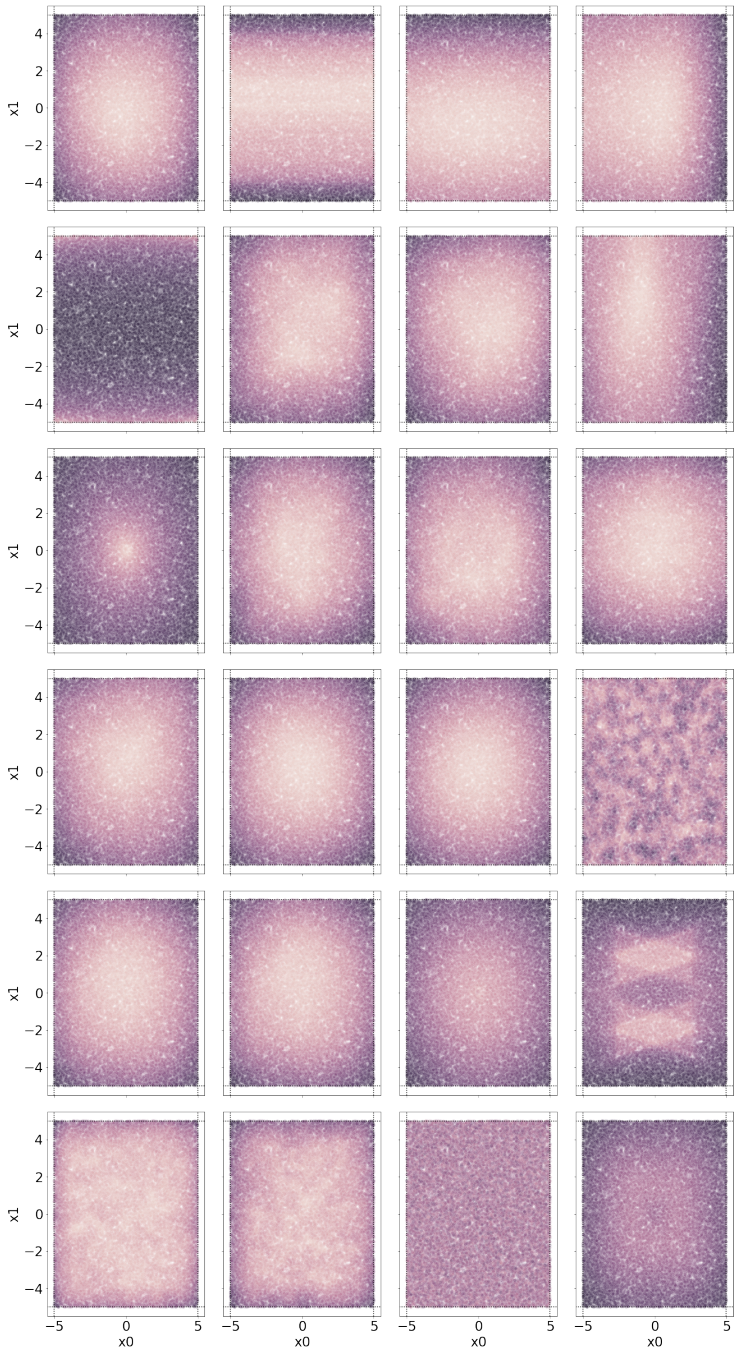


Figure 3.15: Geometric average of relative function value (precision) across the first 500 instances of each BBOB function in 2d.

3.3 Benchmarking Algorithm Behavior: Structural Bias

In addition to the performance or internal state of an algorithm, analyzing the behaviour in terms of points sampled in the domain can lead to important insights into the nature of the optimization procedure. One aspect which we can investigate is whether the algorithm is inherently pulled to some region of the domain, even when there is no selective pressure present. This bias towards a specific part of the search space is called *Structural Bias* (SB) [133]. In this section, we will discuss how SB can be detected, and show some analysis of the SB of several types of optimization algorithms.

The notion of structural bias is built on the assumption that we cannot practically make any assumptions on the location of the optimum within our domain – having an algorithm that consistently finds an optimum only located in the origin is of no use. Therefore, good algorithms should not be biased towards specific locations of the search space, e.g., towards solutions at the origin, centre, or in the borders of the search space. Extrapolating such reasoning, a good optimisation algorithm should be able to find the optima regardless where exactly they are located within the domain. Or, even stronger, a good algorithm should ideally locate solutions anywhere in its domain with equal 'effort', assuming no prior knowledge about the problem is available.

In iterative optimisation heuristics, we can view points sampled during the initialisation as being 'moved' within the domain over time. This movement occurs under the influence of algorithm's operators, which are aiming to find improvements in fitness value. In effect, such movement of the algorithm towards the optima gets steered by the differences in the values of the objective function in the sampled points or their derivatives of some kind. Any feedback that is external to the objective function or domain knowledge might hinder such progression to the optima. Such external feedback stemming from the iterative nature of the algorithm is referred to here as *Structural Bias (SB)*.

Because of the high interdependence between the fitness landscape and the information on the fitness obtained from this cyclical application of the algorithm's operators, the structural bias contribution during the search for optima cannot be easily unveiled if not by means of a specific objective function capable of nullifying such interaction over multiple optimisation runs. The f_0 function serves this purpose and can be used to decouple these effects, thus separating the bias component, arising from algorithmic design choices, from the main driving force represented by the sampled differences in

the fitness landscape [133]. Function f_0 is defined as follows:

$$f_0 : [0,1]^d \to [0,1], \text{ where } \forall x, f_0(x) \sim \mathcal{U}(0,1). \tag{3.1}$$

To identify the structural bias of an algorithm we can thus run it on f_0 and investigate the resulting points found at the end of the optimization run.[5] Up until now, methods to check the uniformity of the distribution of best solutions over multiple runs included visual or statistical inspections.

Displaying locations of the best solutions collected from multiple runs in the so-called 'parallel coordinates' [105] appears to be the most effective way for visualising SB on a multidimensional problem [133]. This approach is easily reproducible, graphically valid and hence convenient. However, when a large number of images is generated [39, 235], visual inspection can become too laborious. Such an approach is also clearly subjective and therefore not reliable for cases where graphical artefacts or unclear patterns cannot be judged by a naked eye.

Instead, we can consider the problem of testing uniformity from the statistical perspective. Assume that a heuristic optimisation algorithm was run N times to minimise f_0. At the end of each run i, the best solution $\mathbf{x}^{(i)}$ found by the algorithm by the end of the run is recorded, where naturally $\mathbf{x}^{(i)} \in [0,1]^d$. The random sample $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \cdots, \mathbf{x}^{(N)}\}$ represents the set of best solutions retrieved by the N runs of the algorithm. Assume that $\{x_j^{(1)}, x_j^{(2)}, \cdots, x_j^{(N)}\}$, $j \in \{1, 2, \cdots, d\}$ was drawn from a probability distribution with a continuous cumulative distribution function $\mathcal{F}_d$. A goodness-of-fit test can be used to test the null hypothesis $H_0 : \mathcal{F}_d \sim \mathcal{U}(0,1)$.

The Kolmogorov–Smirnov test [130] was employed with a sample of size $N = 50$ and significance level $\alpha = 0.01$ in [133]. Subsequently, following the 'power analysis' performed in [131] across three common tests, namely Kolmogorov-Smirnov, Cramér-Von Mises [47], and Anderson-Darling [1] tests, the latter test was chosen and used in combination with the Benjamini–Hochberg [11] correction method for multiple comparisons to achieve higher statistical power. However, it was noted that the original sample size was not adequate for testing all algorithms under investigation. Hence, a higher number $N = 100$ of runs had to be used for some algorithms in order to achieve a satisfactory level of statistical power. Similar problems were encountered in a further study on SB in a subclass of Estimation of Distribution Algorithms [132], even when using an aggregated measure of SB defined as the sum of the statistically significant (across all dimensions) test statistics of the Anderson-Darling test.

[5] Alternatively, the best-so-far point can be used, to track the emergence of SB over time [235]

It was concluded that the described statistical approaches can effectively detect most cases of 'strong' SB but are deficient on other scenarios, including clear 'mild' SB that can be identified with the visual approach. Reasons for these discrepancies between the two methodologies might be a conservative nature of the employed tests combined with the relatively low sample size. Indeed, more accurate SB detection results could be obtained with $N = 600$, as used in [235], instead of $N = 100$.

Using a large sample size ($N = 600$) indeed seems to catch bias more often, but still gives no guarantee to detect *all* different kinds of SB, at any significance level [235]. Even larger sample sizes are necessary for smaller levels of significance, higher desired power and smaller sizes of the deviations to be identified [118]. However, given the limited computational resources to run heuristic optimisation algorithms, it is not always possible to obtain (very) large sample sizes. Therefore, tests better able to detect significant deviations from uniformity given limited sample sizes are desirable for detecting SB.

The BIAS Toolbox As can be concluded from the above, there is a clear need for a better automated statistical testing procedure to detect SB readily available to the community as a software package. We thus propose a toolbox, called BIAS (Bias in Algorithms, Structural), to improve the detection of SB based on algorithm runs on f_0. Based on a large set of statistical tests, BIAS provides an indication of whether or not structural bias is present in the sample of final positions and, in case some SB is found, an assessment of the possible type of SB observed using a random forest model. In addition to the structural bias detection, the toolbox also contains the needed functionality to benchmark other statistical tests for detecting structural bias. The BIAS toolbox is available as a Python package.

Tests

BIAS makes use of statistical tests on the final solutions found by many runs of the algorithm on f_0 (denoted as $\mathcal{F}_d$). These tests all have the **null hypothesis** $H_0 : \mathcal{F}_d \sim \mathcal{U}(0, 1)$. Since the algorithm can work on search spaces of arbitrary dimensionality, we have two types of tests: per-dimension tests and across-dimension tests.

The following tests are designed to work on an individual dimension. However, by aggregating all data we can run these tests on a sample size which is effectively d times larger. If these tests are run on a per-dimension basis, correction strategies need to be applied to deal with the multiple-comparison problem. For this purpose, the Benjamini-Holberg (BH) method was proposed originally, since it is less stringent

Table 3.3: Tests present in the BIAS toolbox for per-dimension (top) and across-dimension (bottom) detection of uniformity.

Test Name	Shorthand	Reference
1-spacing-based test	1-spacing	[194]
2-spacing-based test	2-spacing	[194]
3-spacing-based test	3-spacing	[194]
Sample Range-based test	range	[244]
Sample Minimum-based test	min	[244]
Sample Maximum-based test	max	[244]
Anderson-Darling	AD	[1, 75]
Transformed Anderson-Darling test	tAD	[241]
Shapiro test	Shapiro	[214]
Jarque–Bera test	JB	[111, 220]
Minimum Linear Distance-based test	LD-min	[244]
Maximum Linear Distance-based test	LD-max	[244]
Kurtosis-based test	Kurt	[185]
Minimal Minimum Pairwise Distance-based test	MPD-min	[244]
Maximal Mininimum Pairwise Distance-based test	MPD-max	[244]
Wasserstein distance-based test	Wasserstein	[117]
Neyman-Smooth test	NS	[178, 14]
Kolmogorov-Smirnov test	KS	[130, 75]
Cramer-Von Mises test	CvM	[44, 50]
Durbin test	Durbin	[67, 50]
Kuiper test	Kuiper	[30, 50]
1st Hegazy-Green test	HG1	[101, 50]
2nd Hegazy-Green test	HG2	[101, 50]
Greenwood test	Greenwood	[86, 50]
Quesenberry-Miller test	QM	[196, 50]
Read-Cressie test	RC	[46, 50]
Moran test	Moran	[169, 50]
1st Cressie test	Cressie1	[45, 50]
2nd Cressie test	Cressie2	[45, 50]
Vasicek test	Vasicek	[239, 50]
Swartz test	Swartz	[222, 50]
Morales test	Morales	[168, 50]
Pardo test	Pardo	[184, 50]
Marhuenda test	Marhuenda	[159, 50]
1st Zhang test	Zhang1	[265, 50]
2nd Zhang test	Zhang2	[265, 50]
The mutual information-based test	MI	[213, 186]
Maximum Minimum Pair-wise Distance-based test	MMPD	[244]
Maximum Difference per Dimension between a linear uniform distribution-based test	MDDLUD	[244]

than the standard Bonferroni method. However, in this chapter, we also investigate the effects of other multiple comparison correction methods.

We consider a total of 36 per-dimension tests, listed in Table 3.3. In addition to the tests which work on a per-dimension basis, we can also perform tests on the full set of 30-dimensional data at once. This can be done by grouping together the samples or distances and performing the same test as the per-dimension testing on the aggregated data. For this purpose, we use all per-dimension tests, with the exception of the sample limits-based tests, *LD*, *MPD*, and *Wasserstein* tests. Alternatively, we can use tests which are explicitly designed to handle multiple dimensions at once. In this toolbox, we consider three of these tests, listed at the bottom of Table 3.3.

Table 3.4: Overview of parameterised data sampling scenarios in $[0, 1]$, the total of 194 / 249 scenario settings (per dimension / across dimensions), per considered sample size.

scenario name	how sampled	parameter 1	parameter 2	number of settings	diagnosis[6]
`Uniform`[7]	sample full sample size via $\mathcal{U}(0, 1)$	–	–	1	no bias
`Cut Uniform`[8]	subscenario 1: sample full sample size via $\mathcal{U}(z_c, 1)$, subscenario 2: sample full sample size via $\mathcal{U}(\frac{z_c}{2}, 1 - \frac{z_c}{2})$	fraction cut $z_c \in \{0.01, 0.025, 0.05, 0.1, 0.2\}$		10/10	centre-bias
`Cut Normal`[9]	sample full sample size via $\mathcal{N}(\mu, \sigma)$, remove all points outside $[0, 1]$ and repeat until full sample size	$\sigma \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$	$\mu \in \{0.5, 0.6, 0.7\}$	15/15	centre-bias
`Inverse Cut Normal`	same as above, but transform to have most mass at bounds	same as above	same as above	15/15	bound-bias
`Cut Cauchy`	similar to `Cut Normal` but for Cauchy distribution	same as above	same as above	15/15	centre-bias
`Inverse Cut Cauchy`	same as above, but transform to have most mass at bounds	same as above	same as above	15/15	bound-bias
`Clusters`	sample n_c cluster centre points c_i via $\mathcal{U}(0, 1)$, sample remaining points around them via $\mathcal{N}(c_i, \sigma)$[10]	number of clusters $n_c \in \{1, 2, 3, 4, 5\}$	$\sigma \in \{0.01, 0.025, 0.05, 0.1, 0.2, 0.3\}$	30/30	cluster-bias
`Consistent Clusters`[11]		same as above	$\sigma \in \{0.01, 0.025, 0.05, 0.1, 0.2, 0.3\}$	0/30	cluster-bias
`Loose Clusters`	sample z_u portion of sample size via $\mathcal{U}(0, 1)$. For each remaining point, select an existing point x_i and sample $\mathcal{N}(x_i, \sigma)$	fraction of uniform samples $z_u \in \{0.1, 0.25, 0.5\}$	$\sigma \in \{0.01, 0.02, 0.05, 0.1\}$	12/12	cluster-bias
`Gaps`	sample full sample size via $\mathcal{U}(0, 1)$, select n_c sampled points x_i, remove all sampled points in $[x_i - r_g, x_i + r_g]$, resample missing points outside gaps via $\mathcal{U}(0, 1)$ until full sample size[12]	number of centres $n_c \in \{1, 2, 3, 4, 5\}$	gap radius $r_g \in \{0.01, 0.02, 0.03, 0.04, 0.05\}$	25/25	gap-bias
`Consistent Gaps`[13]		same as above	same as above	0/25	gap-bias
`Spikes`	randomly sample integers in $[1, n_s]$, rescale as uniformly placed spikes in $[0, 1]$	number of spikes $n_s \in \{25, 50, 100, 150, 200, 250, 500, 1000\}$	–	8/8	discretization-bias
`Noisy Spikes`	same as above, but spike locations are independently shifted according to $\mathcal{N}(0, \sigma)$	same as above	$\sigma \in \{0.005, 0.01, 0.02, 0.03, 0.04, 0.05\}$	48/48	discretization-bias

[6] See Section 3.3
[7] Sanity check, excluded from the tests
[8] Two subscenarios: 1. modify only min (equivalent to varying only max, so only one is used to save time); 2. modify both min and max at the same time, with the same parameter setting (half cut on both sides)
[9] Vary μ only to one side since it is equivalent to the other side
[10] For across-dimension tests, need to differentiate between same cluster-centres in each dimension (`Consistent Clusters`) vs. different gaps (`Clusters`)
[11] Only used in across-dimension tests, as it is equivalent to `Clusters` per-dimension
[12] For across-dimension tests, need to differentiate between same gaps in each dimension (`Consistent Gaps`) vs. different gaps (`Gaps`)
[13] Only in across-dimension tests, as it is equivalent to `Gaps` per-dimension

To determine rejection based on the test statistic, we need to either calculate the corresponding p-values, or check if the test statistic exceeds the corresponding critical value. Several of the test we include calculate the p-value by default, but for the others we will use the critical values. To get accurate estimates of these values, we use a 100 000 samples Monte Carlo simulation of the test statistic under the uniform distribution, from which we determine the α-quantiles for $\alpha \in \{0.01, 0.05\}$. The Monte Carlo test is a well known procedure for implementing hypothesis tests [181]. It enables calculating the critical values when the true (sampling) distribution of the test statistic is unknown. The resulting critical values calculated using this procedure are made available.

Methodology

To effectively judge the performance of the proposed tests for different types of SB, we have defined a large portfolio of bias scenarios according to which we can generate an arbitrary number of samples. This set of scenarios is chosen in such a way that most common types of SB are represented. Additionally, these scenarios are parameterised to control the level of bias, which enables us to better judge the robustness of tests. The specification of these scenarios is shown in Table 3.4.

Adding up all parameterizations gives us 194 scenarios to consider in the per-dimension case, and 249 scenarios in the across-dimension case. For each of these scenarios, we generate data with sample sizes $\{30, 50, 100, 600\}$. In the per-dimension case, we collect 1 500 independent sets of samples for each use-case, while the across-dimension cases all use 100 sets of 30-dimensional samples.

For each of the generated sets of samples, we apply the corresponding test-battery with $\alpha \in \{0.05, 0.01\}$: 36 tests for per-dimension case and 32 for the across-dimension case.[14] Using this setup, we thus collect $194 \cdot 1\,500 \cdot 4 \cdot 36 = 4.19 \times 10^7$ test statistics / p-values for the per-dimension tests, and $249 \cdot 100 \cdot 4 \cdot 32 = 3.19 \times 10^6$ for the across-dimension tests.

We show an example of the set of statistical tests applied to an instance of the `Cut Normal` scenario in Figure 3.16. This figure shows the rejections for each dimension individually, as well as the corresponding sample on which this decision is based. This visualisation is available as part of the BIAS toolbox, and provides a visual way to inspect the structural bias present in the scenario.

[14] Several per-dimension tests can not be used for the across-dimension case, full details in [244].

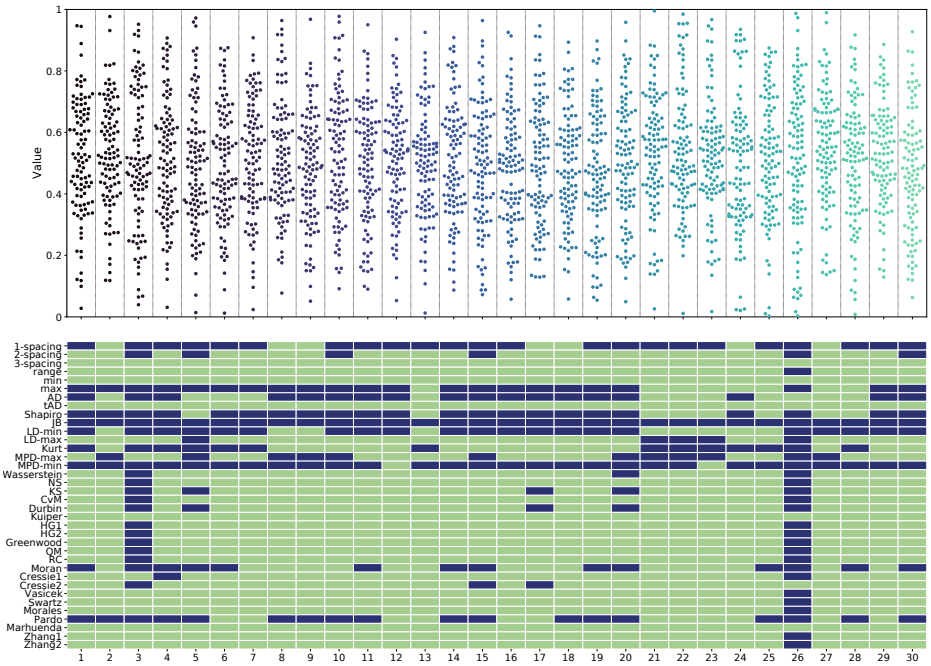


Figure 3.16: Example of an instantiation of the `Normal Cut` scenario with $\mu = 0.5$ and $\sigma = 0.2$, with 100 samples in each of 30 dimensions. The top figure shows the assumed distribution of the final positions potentially returned by an optimisation algorithm in each dimension. Jitter is applied here to reveal vertically overlapping points. The colour scheme is used to highlight different dimensions. The binary heatmap in the bottom figure shows in green which tests reject the null-hypothesis of uniformity per dimension with $\alpha = 0.01$ (no multiple comparison correction applied).

Sample size

To study the impact of the available sample size on the overall performance of different statistical tests, we can aggregate the number of rejections over all parameterizations of each scenario. This allows us to show the fraction of cases of a scenario which are rejected by each test given a certain sample size. Figure 3.17 shows this for the `Shifted Spikes` scenario. From this figure, we can see that the effect of sample size is not the same across all tests. As an example, the *`AD`* test has a relatively high number of rejections at 30 samples, but doesn't reach the same precision as other tests when increasing sample size to 600. This indicates that analysis of the performance of the tests should take the number of available samples into account, as this will influence which tests are more distinguishing.

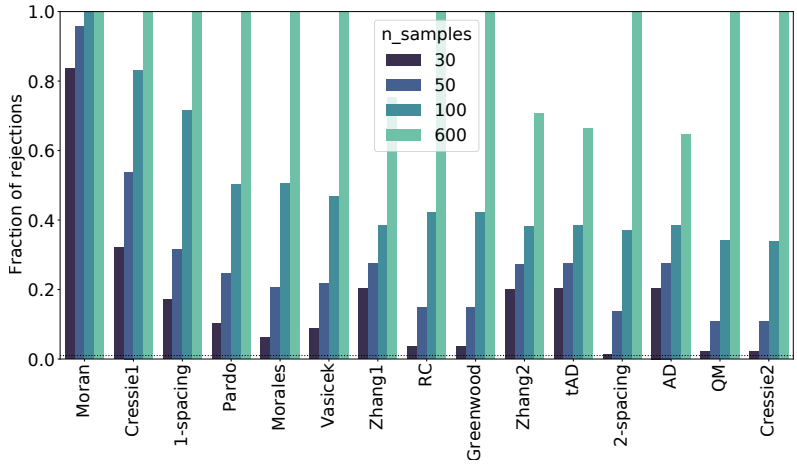


Figure 3.17: Fraction of rejections for each test on the `Shifted Spikes` scenario, with $\alpha = 0.01$ and no multiple comparison correction method applied. Data is aggregated over all parameterizations of the scenario, as described in Table 3.4. This figure shows 15 tests with the most rejections (when aggregated over the different sample sizes). Note that the negative space over each bar $(1 - x)$ is equivalent to the false negative rate of the test.

From Figure 3.17, we can also see that the ***`Moran`*** test significantly outperforms all others on this scenario, but even this test does not reject all cases when the sample size is small. This reinforces the notion that if possible, increasing the sample size is beneficial to the ability to detect less clear cases of structural bias. However, we also note that for most scenarios, a sample size of 50 seems to be sufficient to detect the presence of structural bias. While increasing the sample size would increase the ability to detect less obvious cases of SB, $N = 50$ should be able to correctly identify the most blatant ones.

Overall analysis

With the rejection data, we can investigate the interplay between statistical tests and the scenarios, in order to find what set of tests is more suitable to each kind of structural bias. For this analysis, we make use of the concept of Shapley values [215] to assess the contribution of each test to a portfolio of tests for finding bias in each type of scenario. In particular, we define the marginal contribution of a test t to a

portfolio of tests $T' \subset T$ on scenario S as follows:

$$c(t, T', S, n, \alpha) = \sum_{s \in S} \sum_{i=0}^{n-1} \max_{t' \in (T' \bigcup \{t\})} \mathbb{1}_{t'(s_i) < \alpha} - \sum_{s \in S} \sum_{i=0}^{n-1} \max_{t' \in T'} \mathbb{1}_{t'(s_i) < \alpha} \tag{3.2}$$

where n is the number of realizations s_i of scenario s. The indicator function $\mathbb{1}$ corresponds to the test t rejecting the null hypothesis with significance α on the data from realization s_i.

Based on this definition of marginal contribution, we can compute approximate Shapley values by sampling random permutations calculating the marginal contribution for each test at each position within this permutation [229, 41]. This can be formulated as follows:

$$S(t, S, n, \alpha) = \sum_{r} \sum_{i=0}^{m} c(t, T', S, n, \alpha) : T' \subset T, |T'| = i \tag{3.3}$$

where r is the number of repetitions used, and m is the maximum size of these permutations, which is introduced to ease with computations and because the impact of larger permutations on the total sum is relatively minor – in this paper, we set $m = 10$.

From our experiments, we have found that no single test is clearly preferable over all others. Moreover, an analysis of the Kendall-Tau [122] correlations between the rejections of tests across all scenarios shows that very few tests are highly correlated. Figure 3.18 shows the correlation heatmaps for sample size 600 and $\alpha = 0.01$. We can observe relatively higher correlations among some of the tests listed from `NS` to `Greenwood`, and among some of the tests listed from `QM` to `Pardo`. However, these higher correlations involve very few of the tests and the correlations among the other tests are very low. Therefore, it is likely that different tests are best suited to recognise different types of deviations from uniformity. For this reason, we include *all considered tests* in the BIAS toolbox.

For the per-dimension tests, we should take into account the fact that multiple tests are being done, and thus the p-values should be changed using a correction procedure. For this purpose, we use the Benjamini-Yekutieli (BY) [13] correction method, which we found to obtain the best tradeoff between false positives and false negatives [244].

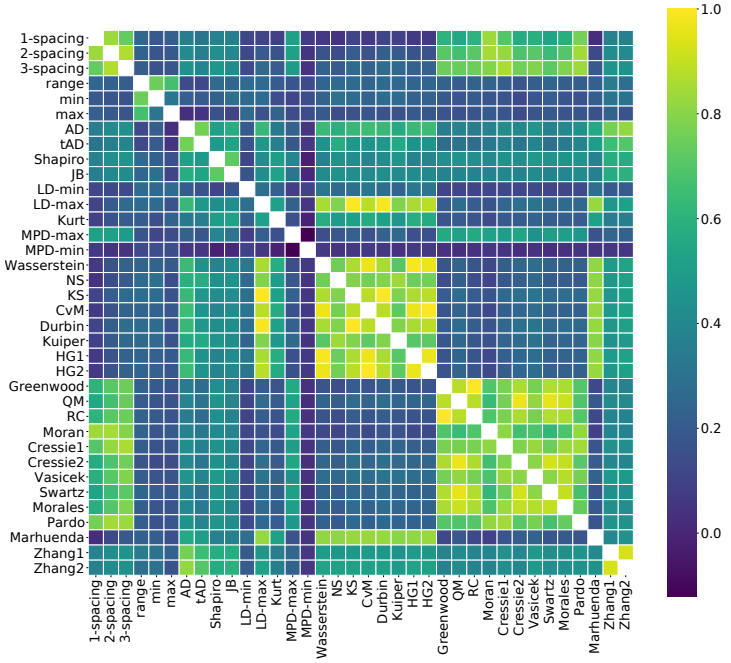


Figure 3.18: Cluster plot showing the Kendall-Tau correlations [122] between test rejections on all scenarios, with sample size 600, $\alpha = 0.01$.

Estimation of the SB type

Since we use the results of many statistical tests to find bias in artificially generated samples and different tests may be better at capturing different deviations from uniformity, we can use these tests to not only check if structural bias is present, but also to identify what the most likely form of bias is. This provides an answer to RQ2. To achieve this, we build a random forest (RF) model, which takes as input the *test-rejections from all per-dimension tests*. This is done to allow scaling to arbitrary dimensions while having one model for all sample sizes. Specifically, if we use statistical test values directly, we would need one model per sample size, and a way to aggregate the resulting predictions. Instead, a RF based on rejections only needs to deal with the aggregation problem.

The data used to train and evaluate the random forest model consists of the full set of scenario results (per-dimension version) on all tests, with the output being the scenario-type it comes from. However, if for a specific sample no test rejects the null-hypothesis, these samples are discarded, since we have no evidence of structural bias.

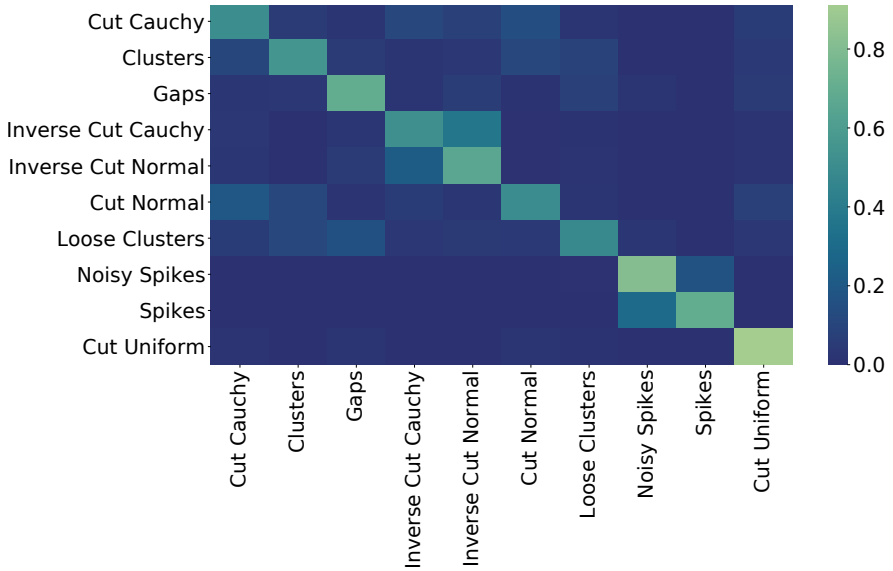


Figure 3.19: Confusion matrix for the random forest model trained on test rejections, aggregated over all sample sizes. The true scenario is shown on the y-axis, while the predicted scenario is on the x-axis.

This two-stage approach leaves us with 1 158 000 biased samples, on which we train the RF model with 100 trees and balanced class weights. A confusion matrix created from an 80-20 test split is shown in Figure 3.19 (F1-score of 0.56). We see that the distinction between the `Cut Uniform` and the other scenarios can be challenging to accurately detect. However, this doesn't have to be an issue for practical detection of SB, since the scenarios misidentified as `Cut Uniform` might show similar types of bias, even though their initial creation mechanism is different.

To provide a more practical estimation of SB in our toolbox, we create an additional model to predict the type of bias, as shown in the final column of Table 3.4. These 5 categories are more distinct from each other, removing overlap between some similar classes, i.e. between `Spikes` and `Noisy Spikes`. Overall, this model gives us an improved F1-score of 0.79 on a similar 80-20 split.

To use these models in the BIAS toolbox to predict bias of the multi-dimensional test, we need to perform some aggregation across dimensions to transform it into a binary vector. We do this by checking the number of false positive tests in 30D uniform samples. We run 10 000 simulations, where we record the maximum number of test rejections by each test. This gives us a total of 92 cases where a test gives 2

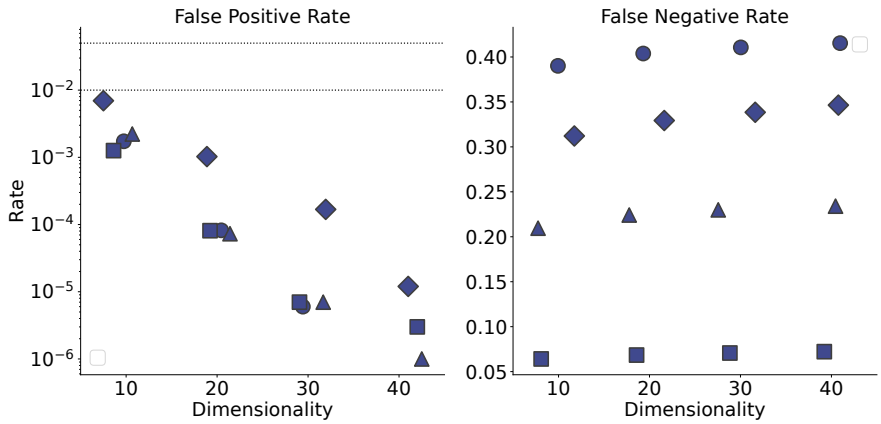


Figure 3.20: Evaluation of toolbox in different dimensions at $\alpha = 0.01$: fraction of false positives (on the left) and aggregated fraction of False Negatives across all scenarios (on the right). On both figures, markers identify the used sample size: ◯, ♢, △ and □ are 30, 50, 100 and 600, respectively.

rejections, and 2 cases where a test gives 3 rejections. As such, we set the threshold for the aggregation of multi-dimensional data to $0.1 \cdot d$. If no test is rejected in this aggregation, we consider the samples to be non-biased. This threshold value is then used to create the binary input vector for the RF model.

To verify that this works for other dimensionalities as well, and to gauge the overall performance of the toolbox, we simulate the false positive and false negative rates. This is achieved by sampling (with replacement) from the set of test-statistics on each of our used scenarios and applying this aggregation rule. For false positives, this is done 1 000 00 times on the (true) uniform data, while for false negatives it is done 10 000 times on every non-uniform scenario. The results, shown in Figure 3.20, indicate that while the $0.1 \cdot d$ threshold is rather conservative on higher dimensionalities, the FPR is well below the selected $\alpha = 0.01$, while the FNR is not needlessly increased.

Benchmarking SB of real algorithmic data

We use data from a heterogeneous pool of heuristics executed over f_0 at dimensionality $d = 30$ for a maximum of $10000 \cdot d$ fitness function calls. In total, we consider 432 optimisation heuristics, which fall into the following categories (all except the latter use $N = 100$, while the latter uses $N = 50$ runs each):

- Variants of Differential Evolution (195 configurations),

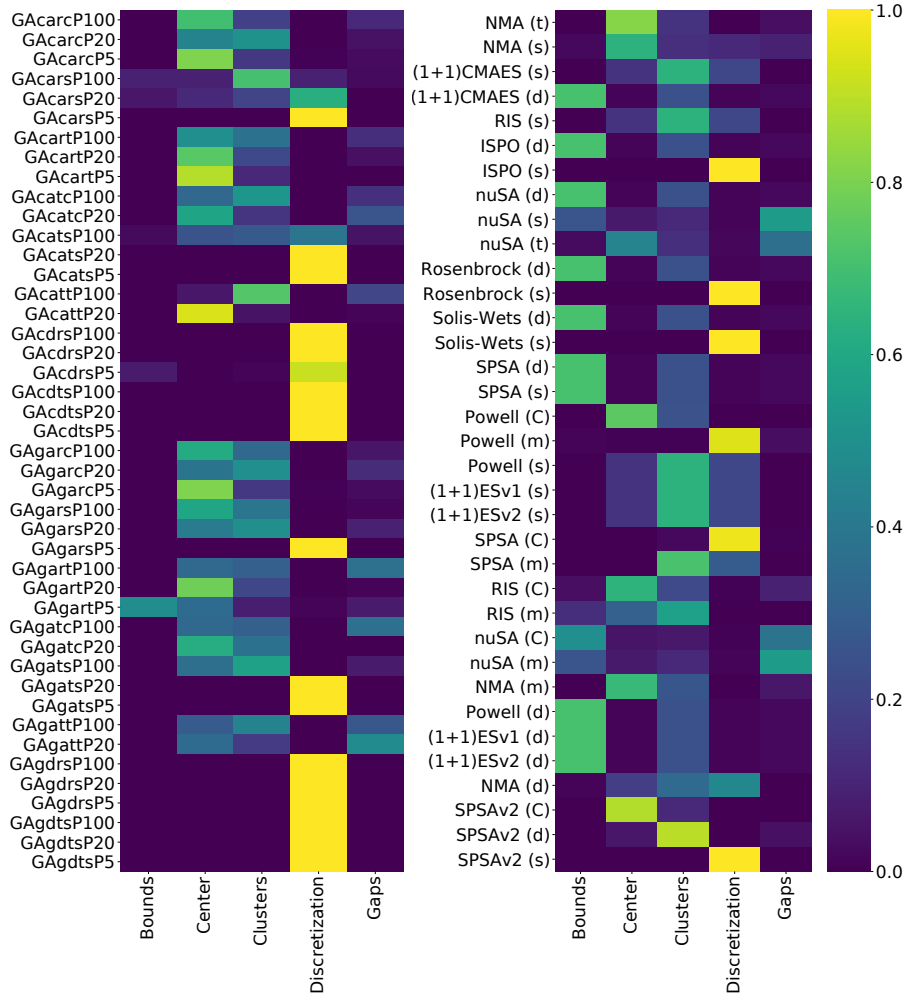


Figure 3.21: Predicted SB class probabilities of the biased GA configurations (left, sorted alphabetically) and the biased single-solution algorithms (right), using the random forest model. Names for the GA are structured as mutation–crossover–selection–SDIS–population size. For the single-solution algorithms, the character in brackets refers to the used SDIS.

- Compact optimisation algorithms (81 configurations),
- Single-solution algorithms (60 configurations),
- Variants of Genetic Algorithms (96 configurations).

For each of the considered algorithm configurations, we collect their final positions and feed these into the BIAS toolbox. In Figure 3.21 (left side), we show the outcome from the RF predicting the type of structural bias present in the different GA configurations (only the biased ones are shown). This shows that there are quite some differences in the detected bias, even within this limited algorithm design space. It is also interesting to note that the population size seems to have a relatively small impact on the type of predicted bias, which seems to be mostly impacted by the operator configuration.

For the single-solution algorithms, we see in the right part of Figure 3.21 that the strategy of dealing with infeasible solutions (SDIS) seems to drastically change the type of detected bias. For example, the Powell algorithm is classified as 'discretization' bias when using mirror strategy, while the classification changes completely with a COTN strategy. Such differences can give us useful insight into the effect of these SDIS methods on the optimisation behaviour of these algorithms.

DEEP BIAS In addition to the statistical approach used in the BIAS toolbox, we can make use of deep learning techniques to identify deviations from uniformity. To this end, we extended the BIAS toolbox with DEEP-BIAS: a convolutional neural network which detects both the presence and type of structural bias. This network structure is visualized in Figure 3.22.

By not going through the intermediary step of getting test statistics from the per-dimension test, this setup allows for a better classification of the type of bias detected, while remaining competitive with the statistical test in terms of bias detection, as is highlighted in Figure 3.23.

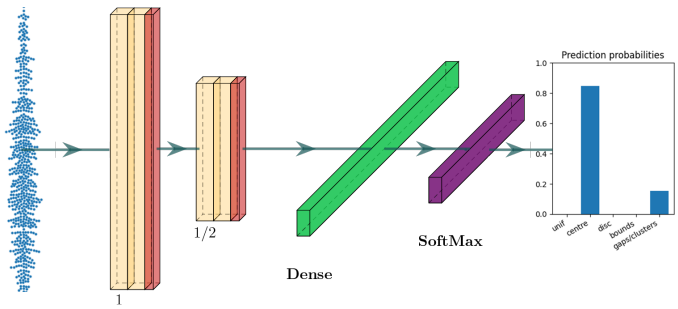


Figure 3.22: General one-dimensional CNN architecture with optimal hyper-parameters per sample size. The network takes as input a sorted distribution fixed sample size. Yellow layers are 1d-CNN layers, red layers are max-pooling layers, green layer is a dense layer and finally a classification head with SoftMax activation function resulting in five class probabilities per sample.

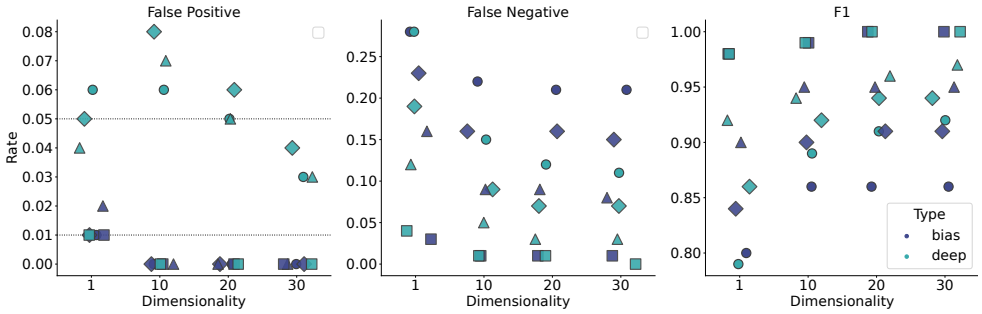


Figure 3.23: Comparison (with $\alpha = 0.01$) of the original BIAS toolbox (blue) and the Deep-BIAS (teal) in terms of false positives (left), false negatives (middle) and F1-score (right). On all figures, markers identify the used sample size: $\circ$, $\diamondsuit$, $\triangle$ and $\square$ are 30, 50, 100 and 600, respectively.