Universiteit
Leiden
The Netherlands

From benchmarking optimization heuristics to dynamic algorithm configuration

Vermetten, D.L.

Citation
Vermetten, D. L. (2025, February 13). *From benchmarking optimization heuristics to dynamic algorithm configuration*. Retrieved from https://hdl.handle.net/1887/4180395

Version:	Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/4180395

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

Chapter 4

Algorithm Configuration and Selection

In the previous chapter, we have seen that benchmarking plays a critical role in the understanding of optimization algorithms. By observing the performance of an algorithm on a variety of different problems, we can gain insights into its strengths and weaknesses. Critically, different algorithms naturally take advantage of different kinds of problem structures. A greedy hill-climbing algorithm for example is very efficient on a sphere-model, while it would perform poorly on highly multi-modal functions. Compare this to a purely random search, and it is clear that we want to choose a different algorithm to solve our sphere problem than our multi-modal example.

It should be noted that the notion of exploiting algorithm complementarity is not limited to the optimization context [217]. For example, in machine learning, complementarity between predictors is a large source of improvement, and top-performing frameworks for automated machine learning rely on the ensembling of complementary models to achieve state-of-the-art performance [76, 74]. Similarly, SAT-solving is a context in which the exploitation of algorithm complementarity has led to big improvements in the state-of-the-art [261, 104].

In general, *algorithm selection* aims to exploit these kinds of complementary strengths of algorithms by selecting a different optimization algorithm for each problem [205]. In the optimization context, algorithm selection attempts to find the best algorithm A from a portfolio $\mathcal{A}$ to solve a specific function f from a set of functions $\mathcal{F}$. Specifically, this static version of algorithm selection can be defined as follows:

Definition 4.1 (Static Algorithm Selection)**.** Given an algorithm portfolio $\mathcal{A}$ and a function $f \in \mathcal{F}$, we aim to find:

$$\arg\min_{A \in \mathcal{A}} \text{PERF}(A, f) \ ,$$

where $PERF : \mathcal{A} \times \mathcal{F} \to \mathbb{R}$ is a performance measure (which assigns lower values to better-performing algorithms).

One key aspect of algorithm selection in the context of black-box optimization is the representation of the problem. The black-box nature of the functions implies that we have very limited information available to base our decisions on. While algorithm selection approaches based only on the available information (problem dimensionality, variable types, evaluation budget) are being developed [166], the amount of complementarity these approaches can exploit is naturally limited. As such, the most common setup is to spend part of the total evaluation budget to collect samples from the function and use information from these samples as the problem representation [174]. In this setup, an algorithm selector is a machine learning model trained on a dataset containing the performance of each algorithm from $\mathcal{A}$ on each function from $\mathcal{F}$, where the function is characterized by a set of features [124]. These features are usually extracted via Exploratory Landscape Analysis (ELA, see Section 2.5), which needs relatively low sample sizes to calculate large sets of problem characteristics [164].

In addition to algorithm selection, we consider the *algorithm configuration* scenario. In this setup, we have a parameterized algorithm or algorithm family, and we aim to find the parameter setting which performs best on the selected (set of) problem(s). Where algorithm selection exploits complementarity between the algorithms in the portfolio, algorithm configuration benefits from the sensitivity of an algorithm's performance to its parameter settings.

Definition 4.2 (Static Algorithm Configuration)**.** Given an algorithm A with parameterization Θ_A and a function $f \in \mathcal{F}$, we aim to find:

$$\arg\min_{\theta \in \Theta_A} \text{PERF}(A_\theta, f)$$

Where many algorithm selection techniques rely on the availability of a complete enumeration of all algorithms and functions, this is generally not feasible in an algorithm configuration context. Algorithm configuration is thus treated as an optimization problem in its own right, where noisy evaluations, mixed-variable and conditional

search spaces, and expensive evaluations are common. Nevertheless, applying algorithm configuration to optimization heuristics has been very successful, and a wide variety of specific tools have been developed for this purpose.

In this chapter, we illustrate how we apply modular design principles to create large parameterized search spaces for both differential evolution and evolution strategies. We then apply a state-of-the-art algorithm configurator to these spaces and show what insights we gain into the impact of different algorithmic components on the final performance in certain landscapes. We end the chapter by illustrating some of the limitations we still face, with a particular focus on the large variability in performance of algorithm configuration when applied to the modular CMA-ES.

This chapter is based on the following publications: [51, 240, 247].

4.1 Modular Algorithm Design

Many popular optimization algorithms have been well-studied over the last decades. This has led to significant improvement and allowed for a great deal of specialization to different types of problems. While these modifications are all interesting in isolation, the true impact they have on the state-of-the-art is often hard to assess. One particular reason for this arises from the fact that algorithms can be inherently challenging to implement. Inconsistencies in the description, ignored edge cases, and even potential bugs can have a significant impact on the behaviour of an algorithm and the interpretation of results [26, 16]. Issues such as these have raised questions regarding the reproducibility of research in computer science as a whole, and evolutionary computation is no exception [151]. Because of this, comparing different variants of algorithms can be difficult to do fairly. Since researchers often implement the underlying algorithm from scratch, to then add their proposed modification (and in most cases a selected set of other algorithm variants for comparison), clear comparisons are often hard to find.

In an ideal setting, the community would maintain standardised implementations of core algorithms and the proposed modifications would be compared against the same set of state-of-the-art algorithm variants. Unfortunately, this might still be an impossible goal. However, algorithm modifications can still be fairly compared, as long as they are implemented in one common framework. This can be achieved through modular algorithms. From one common core algorithm, the variants are implemented as modules that can easily be swapped out.

The ideas behind modular algorithms have been around for decades [32, 153, 156],

but although they have been shown to be extremely useful [65], their adoption in evolutionary computation has been relatively slow. In recent years, several new modular implementations of popular algorithms have been released, including the modular CMA-ES [51, 234] and the Particle Swarm Optimisation framework [34]. These works highlight the benefits of modular algorithms not only for fair comparisons, but also hint at the potential to study interactions between modules.

4.1.1 Modular DE

In this section, we propose a first step towards a modular version of Differential Evolution (DE), a heuristic originally introduced in [219] to optimise a single-objective real-valued fitting problem, and whose design took into consideration elements from evolutionary algorithms and swarm intelligence optimisation (see [40, 242] for some insights on these aspects) and a simple core mechanism based on computing difference vectors through linear combinations of candidate solutions. DE has been around for almost 30 years and its popularity means that a wide variety of modifications have been proposed over the years [49]. However, when comparing the benchmark data, the relative benefits of many of these modifications seem to vary widely. Our objective is to provide an initial analysis of the performance of a set of 14 independent modules. This does not cover the full space of DE variants, but nonetheless highlights the potential of modular algorithms to aid in understanding the contributions made by these algorithmic variations.

Included Modules

Similar to other heuristic optimisers, DE naturally lends itself to a reformulation as a modular algorithm made up of a number of connected modules/operators where every independently made choice for a module is fully compatible with all choices for other modules. In fact, previous work has shown the usefulness of considering these operators as independent modules, e.g. to rigorously analyse the impact of the crossover operator [36]. In this section, we use this modularity to create a framework which we call *Modular DE* where a full combinatorial range of modules is available for each algorithm component, see Table 4.1.

Initialisation

To create the initial population, we implemented several sampling strategies (`Sampler`, see Table 4.1). The most common is to create a uniform distribution across the

entire domain. Alternatives are to use other distributions or low-discrepancy sampling methods. We choose to include the Halton and Sobol sequences to represent low-discrepancy sampling and a Gaussian distribution (centred around the origin, with $\sigma = (U - L)/6$, where U and L are the upper and lower bounds, respectively) to represent other kinds of distribution. Furthermore, a previous study has proposed using an oppositional initialisation strategy [197] (`Opposition`), where each time we generate an individual for the initial population, we also generate its mirror image around the origin.

Mutation

The mutation operator has been the focus of many modifications of DE, see, e.g. [266, 73, 106, 49, 37]. To capture the most established mutation variations of the kind `x/y` (where x is the base vector and y the number of differences), and to give flexibility in adding new variants, we implement the mutation operator through the combination of 3 modules. The first two modules, namely `Base` and `Ref`, help define the strategy `x`. Note that the reference solution `Ref` can be set to `none`, while the `Base` solution is not optional. In this scenario `x = Base`. Conversely, when `Ref` is one of the admissible reference solutions displayed in Table 4.1, a scaled version of the vector directed from `target` to the reference point is generated and added to `Base`, i.e. `Base + F(Ref-target)`. Therefore, when `Ref` is not `none`, one obtains any of the classic strategies of the kind `x = target-Refs`, plus new ones by varying the base vector. The third module, namely `Diffs`, is used to set the number `y` of difference vectors.

In addition to this restructuring of the definition of the mutation operator, we implement the option of using `WeightedF`, which reduces F at the beginning of the search and then increases it towards the end [25].

One more modification makes use of an archive of external solutions, as done, e.g., in [266], where one of the solutions in the archive is chosen to be part of one of the difference vectors - a scheme that has been shown to lead to improvements in the past and is activated via the module `Archive`.

Table 4.1: Available modules and parameters for the modular DE, their type ('c' for categorical, 'i' for integer or 'r' for real) and their domain. The choices shown in bold correspond to the default settings. For the numerical parameters, the default values are added after their domain. The 'Shorthand' column indicates the names used for these modules in the figures throughout this paper.

Operation	**Module Name**	**Shorthand**	**Type**	**Domain**
Initialization	Base sampler	`Sampler`	c	{'gaussian', 'sobol', 'halton', **'uniform'**}
Initialization	Oppositional initialisation	`Opposition`	c	{true, **false**}
Mutation	Base vector	`Base`	c	{**'rand'**, 'best', 'target'}
Mutation	Reference vector	`Ref`	c	{**none**, 'pbest', 'best', 'rand'}
Mutation	Number of differences	`Diffs`	c	{**1**, 2}
Mutation	Use weighted F	`WeightedF`	c	{true, **false**}
Mutation	Use archive	`Archive`	c	{true, **false**}
Crossover	Crossover method	`Crossover`	c	{**'bin'**, 'exp'}
Crossover	Eigenvalue transformation	`EigenX`	c	{true, **false**}
Bound correction	Bound correction	`SDIS`	c	{none, **'saturate'**, 'unif-resample', 'COTN', 'toroidal', 'mirror', 'hvb', 'expc-target', 'expc-center', 'exps'}
Adaptation	F adaptation method	`AdaptF`	c	{**none**, 'shade', 'shade-modified', 'jDE'}
Adaptation	CR adaptation method	`AdaptCR`	c	{**none**, 'shade', 'jDE'}
Adaptation	Population size reduction	`LPSR`	c	{true, **false**}
Adaptation	Use JSO caps for F and CR	`Caps`	c	{true, **false**}
Parameter	Population size	λ	i	$\{4, \ldots, 200\}$ $(\mathbf{4+\lfloor(3\log(}\boldsymbol{d}\mathbf{))\rfloor})$
Parameter	Scale factor	F	r	$[0, 2]$ $(\mathbf{0.5})$
Parameter	Crossover rate	CR	r	$[0, 1]$ $(\mathbf{0.5})$

Crossover

The classical studies in DE generally consider two types of crossover: binomial (`z=bin`) and exponential (`z=exp`) [193], where the names refer to the distributions used for the probability of exchanging design variables between `target` and `mutant`. Both these types of crossover are included in this work.

Furthermore, we also include the option of performing the procedure from [87], by activating the eigenvalues transformation module `EigenX`, which allows using the `bin` or the `exp` operator and still maintaining rotational invariant behaviour. This is obtained by producing a covariance matrix from the individuals that make up the current population and diagonalising it with the Jacobi method [54] to calculate the eigenvalues and eigenvectors. These are real-valued and form an orthogonal basis (since the covariance matrix is symmetric and surely diagonalisable) and are arranged in a matrix R used to rotate `target` and `mutant` before performing the crossover. Note that the obtained `trial` has to be transformed back to the original coordinate system. This is an easy task, as the conjugate matrix R^* is equivalent to R^T in this scenario. Therefore, the multiplication between the transposed transformation matrix R^T and the newly generated point returns the desired `trial`.

Boundary Correction

There exist several mechanisms for boundary correction in the literature that allow us to deal with infeasible solutions. The most used within the DE community can be found in [17, 134]. For the proposed modular DE framework, we selected a varied set of 10 strategies for box-constrained problems.

Parameter Adaptation

Most state-of-the-art DE variants make use of adaptive parameters. So, in the proposed modular framework we implement adaptation methods for the DE core parameters, namely F, CR, and λ. The simplest is `LPSR`, which linearly reduces the population size over time [24]. For F and CR, we implement the adaptation mechanisms of SHADE and jDE [23, 223]. For F, we add an additional mechanism which uses the mean of the memory, instead of generating a different distribution for each individual, in the SHADE's adaptation strategy.

One final option to change the adaptation process is to use JSO [25] caps for F and CR (`Caps`), which, once activated, caps the values of these two parameters with different thresholds depending on conditions on the used computational budget.

4.1.2 Modular CMA-ES

Similar to the modular Differential Evolution, we also consider a modular variant of CMA-ES. This framework is in large part a redesign of the Modular Evolutionary Algorithms (ModEA) framework introduced in [234]. The modifications focus on the CMA-ES family of algorithms, to such an extent that the design of other evolutionary algorithms is no longer possible, thus requiring the change of names. The new framework was dubbed the Modular CMA-ES (modCMA) and is available as an open-source Python package within the IOHprofiler [63] environment.[1]

To design the Modular CMA-ES, we use the implementation from the popular CMA-ES tutorial [90] as a starting point. This work provides a detailed description of the CMA-ES algorithm, including a practical guide to its implementation. From this basic design, we separate the CMA-ES in a number of functionally related blocks, in order to allow a customization of a specific part of the algorithm. This allows us to implement algorithmic variants of the CMA-ES as functional modules. From a user perspective, any of these modules could then be combined in order create a custom instantiation of the CMA-ES, by selecting an option for each available module.

In ModEA, eleven of such modules were already implemented. These were all reimplemented in the Modular CMA-ES, with a few changes to the structure of the options. Specifically, we removed the *Pairwise Selection* as a module. Instead, we incorporated this option in the *Mirrored Sampling* module as the option *Mirrored Sampling with Pairwise Selection*, converting this module from binary to ternary. This is done because the pairwise selection method is not suited for use without mirrored sampling [3].

We implemented a new module for performing boundary correction, and added five alternative options for performing step-size adaptation. These two extensions to the framework will be the focus of our analysis through out this work. This set of changes give us the following list of modules for the redesigned Modular CMA-ES:

1. **Active Update**: Bad candidate solutions are penalized in the covariance matrix update using negative weights [112]. Note that in [90], this is given as the default version, here we consider it to be optional.

2. **Elitism**: $(\mu + \lambda)$ - selection instead of (μ, λ) - selection.

3. **Orthogonal Sampling**: All the newly sampled points in the population are orthonormalized using a Gram-Schmidt procedure [254].

[1] https://github.com/IOHprofiler/ModularCMAES

4. **Sequential Selection**: Candidate solutions are immediately ranked and compared with the current best solution. If an improvement is found, no additional objective function evaluations are performed [28].

5. **Threshold Convergence**: A method for balancing exploration with exploitation, scaling the mutation vectors to a required length threshold, which decays over time [187].

6. **Step-Size Adaptation**: Supplementary to the default Cumulative Step-size Adaptation (CSA), Two Point step-size Adaption (TPA) [88] is implemented. TPA requires two additional objective function evaluations, used for evaluating both a shorter and a longer version of the population's center of mass. The version which shows the higher objective function value determines whether the step-size should be increased or decreased. Five newly added mechanisms for performing step-size adaptation are implemented.

7. **Mirrored Sampling**: For every newly sampled point, its mirror image is added to the population, by reversing its sign [3]. This can be turned on or off, or as a third option this module can be set to *Mirrored Pairwise Selection*, where only the best point of each mirrored pair is used in recombination.

8. **Quasi-Gaussian Sampling**: Instead of performing the simple random sampling from the multivariate Gaussian, new solutions can alternatively be drawn from quasi-random sequences (a.k.a. low-discrepancy sequences) [6]. We implemented two options for this module, the Halton and Sobol sequences.

9. **Recombination Weights**: Three options are implemented; 1) default weights (see [90]), 2) equal weights: $w_i = 1/\mu$, and 3) $w_i = 1/2^i + 1/(\lambda 2^\lambda)$ for $i = 1, 2, \ldots, \lambda$.

10. **Restart Strategy**: When the optimization process stagnates, the CMA-ES can be restarted using a restart strategy. Two strategies are implemented in addition to the default 'off' setting. IPOP [5] increases the size population after every restart by a constant factor. BIPOP [155] also changes the size of the population, but alternates between larger and smaller population sizes.

11. **Boundary Correction**: If candidate solutions are sampled outside the search domain, they can be transformed back into the search domain by applying a boundary correction operation. In Section 4.1.2, we describe six options for performing boundary correction which have been implemented.

Table 4.2: The modules available for the Modular CMA-ES. The numeric index for each module corresponds to the index used in the text of Section 4.1.2. Newly added modules/options are given in bold.

#	0 (default)	1	2	3	4	5	6
1	off	on	-	-	-	-	-
2	off	on	-	-	-	-	-
3	off	on	-	-	-	-	-
4	off	on	-	-	-	-	-
5	off	on	-	-	-	-	-
6	CSA	TPA	**MSR**	**PSR**	**xNES**	**m-xNES**	**p-xNES**
7	off	on	on w. PS	-	-	-	-
8	off	Sobol	Halton	-	-	-	-
9	default	$\frac{1}{\lambda}$	$\frac{1}{2^i} + \frac{1}{\lambda 2^\lambda}$	-	-	-	-
10	off	IPOP	BIPOP	-	-	-	-
11	off	**UR**	**MCS**	**COTN**	**SCS**	**TCS**	-

In Table 4.2, an overview is given of all currently implemented modules and their options in the Modular CMA-ES framework.

Boundary Correction

In the original modEA framework [233], a boundary correction function taken from [141] was implemented, and always applied after each mutation. In some cases, however, this operator can degrade the performance of the algorithm quite drastically. We therefore decided to make the boundary correction optional, and to implement it as a module, for it to only be used when beneficial. A number of different boundary correction strategies were implemented, taken from [40]:

1. **None**: No correction is applied to infeasible coordinates of solutions.
2. **Uniform Resample (UR)**: Replaces all infeasible coordinates of a solution with new coordinates sampled uniformly at random within the search space.
3. **Mirror Correction Strategy (MCS)**: Mirrors all infeasible coordinates of a solution with respect to its closest boundary.
4. **Complete One-tailed Normal Correction Strategy (COTN)**: All infeasible coordinates are replaced with new coordinates inside the search space according to a rescaled one-sided normal distribution centered on the boundary.
5. **Saturation Correction Strategy (SCS)**: All infeasible coordinates is set to the closest corresponding bound.

6. **Toroidal Correction Strategy (TCS)**: All infeasible coordinates get reflected off the opposite boundary.

Step-Size Adaptation

We consider a number of alternative step-size adaptation mechanisms for the Modular CMA-ES. We take inspiration from [137], which provides a qualitative evaluation of multiple step-size adaptation mechanisms used in ES. In addition to the CSA and TPA step-size adaptation methods, which were already available, we added the following procedures:

1. **Median success rule (MSR)** [72]: The MSR mechanism adapts the step-size σ as follows: it firstly computes a success rate by checking the number of current individuals that are better than some user-defined quantile of the function values in the previous population, then accumulates such success rates in every iteration, and finally decides to increase the step-size if the cumulated value is bigger than $1/2$ and decrease it otherwise.

2. **Population success rule (PSR)** [154]: PSR determines the success rate of the current population using a rank-based approach. It firstly sorts all individuals in the current and previous population together, then retrieves the set of ranks of individuals belonging to the current iteration and the one for the previous iteration, and finally calculates the average rank difference between those two sets as the population success rate, which controls the step-size updates.

3. **xNES step-size adaptation (xNES)** [83, 260, 137]: This method calculates the length of each standardized mutation vector and subtracts from it the expected length of the standard Gaussian vector. The resulting difference is then scalarized using the same weights used in the recombination, which is finally fed into an exponential function to generate a multiplicative coefficient to modify the step-size.

4. **mean-xNES step-size adaptation (m-XNES)** [137]: This mechanism functions similarly to xNES, with the exception that it takes the standardized differential vector between current center of mass and the one in the previous iteration and compares it to the expected length of the standard Gaussian vector.

5. **xNES with log normal prior step-size adaptation (p-xNES)** [137]: This approach resembles the principle of self-adaptation for step-sizes, where λ trial

Table 4.3: Set of 11 commonly used DE variants and the way they are implemented in modular DE. Empty cells indicate default values are used.

Name/Author		Mutation Settings	F	CR	λ	Other Settings
L-SHADE	[223]	Base : target, Ref : pbest	adaptive		$18 \cdot d$	LPSR, Archive, AdaptF_CR : shade
SHADE		Base : target, Ref : pbest	adaptive		$10 \cdot d$	Archive, AdaptF_CR : shade
DAS1	[48]		0.8	0.9	$10 \cdot d$	
DAS2		Base : target, Ref : best	0.8	0.9	$10 \cdot d$	
Qin1	[195]		0.9	0.9	50	
Qin2			0.5	0.3	50	
Qin3		Ref : best	0.5	0.3	50	
Qin4		Ref : best, Diffs : 2	0.5	0.3	50	
Gamperle1	[82]	Ref : best, Diffs : 2	0.45	0.4	$2 \cdot d$	
Gamperle2		Ref : best, Diffs : 2	0.6	0.9	$2 \cdot d$	
jDE	[23]		adaptive		100	AdaptF_CR : jDE

step-sizes are generated from a log-normal distribution which takes the current step-size as its mean and each trial step-size is used to sample a candidate point. To determine the new step-size, this method calculates the weighted sum of the log-transformed trial step-sizes, where those assigned to their corresponding candidate points in the recombination.

4.2 Algorithm Configuration for Modular Algorithms

4.2.1 Results of Configuring ModDE

Experimental Setup

Experiment 1 In order to analyse the potential of modular implementation of DE, we recreate a set of 11 *known* versions of DE within our framework (referred to as *common* variants). These algorithms are shown in Table 4.3, where all non-default parameters are mentioned. In addition to this, we can create a set of 30 *single-module variations*: DE versions where all modules are set to their default value, except for one. As such, each non-default module option is enabled in exactly one *single-module* variant. For these *single-module* variants, we set $F = CR = 0.7$, and $\lambda = 10 \cdot d$, based on the recommendations of [146].

For each DE variant, we collect performance data on all 24 BBOB problems (Section 3.2), using IOHexperimenter [53] for data collection. We perform 50 runs per function, spread over 10 instances (5 independent runs per instance). We repeat this for dimensionalities $d \in \{5, 10, 20\}$, where we give each run a budget of 50 000 function evaluations.

To evaluate the performance of each algorithm, we opt to use the Empirical Cumulative Distribution Function (ECDF). In particular, we use a normalized *Area Over*

the ECDF Curve (AOC) as an *anytime performance* measure [93] (more details in Section 2.4).

Experiment 2 For our second set of experiments, we use the algorithm configuration tool irace [152] to tune the performance of the modular DE on the same set of BBOB problems. Each irace run uses a budget of 10 000 evaluations, where each evaluation corresponds to running a DE variant with the selected parameter setting. We use irace, with its first-test parameter set to 5, and the remaining parameters kept at their default values.

We perform 10 independent runs of irace on each function from the BBOB suite, for dimensionalities $d \in \{5, 10, 20\}$, where irace has access to the first 5 instances of the function. We set the targets for ECDF to 81 logarithmically spaced values between 10^8 and 10^{-8}. We use AOC as the target since it has been shown that the increased signal it captures relative to measures such as Expected Running Time (ERT) can lead to overall performance improvements, even when evaluating the result with a different measure [264].

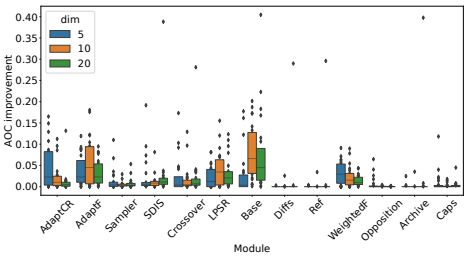
In addition to these per-function tuning runs, we also perform 10 tuning runs where we tune for aggregated performance over all the functions by setting the irace instance set to the 24 BBOB problems.

The resulting *elite configurations* for the across-function tuning are validated using the same settings as the DE variants from the first experiment: 5 independent runs on 10 instances of each BBOB problem. For the per-function tuning, we instead perform 5 independent runs on 50 instances of the problem on which the tuning was performed.

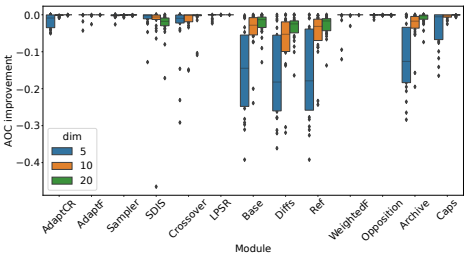
Single-Module and Common DE Variants

First, we investigate the *single-module* DE variants, which can be used to illustrate the impact of each module in isolation. We achieve this by comparing the performance of the default DE (all modules at their default value as seen in Table 4.1) to the variant with the identified best options enabled for each module. The resulting distribution of improvements is shown in Figure 4.1a.

From Figure 4.1a, we can see that some modules have relatively minor impact when the optimal option is selected independently from any other modules. This is the case for e.g. the number of difference components (`Diffs`) and the use of an archive population (`Archive`). In fact, if we instead consider the performance deterioration when making the worst choice for each module, these ones show a significant change over the default setting, as can be seen in Figure 4.1b. The combination of these two figures gives an overall importance of each module, in the sense that if only



(a) Improvement in AOC over the default setting when selecting the best-performing option for each module.



(b) Reduction in AOC over the default setting when selecting the worst-performing option for each module.

Figure 4.1: Impact of selecting the best (a) and worst (b) option for each individual module, measured as the difference in AOC relative to the default configuration.

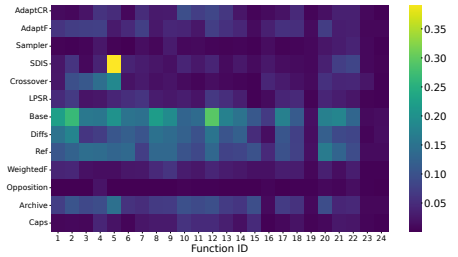


Figure 4.2: Importance of each module to the AOC on each of the 24 BBOB functions, aggregated over the used dimensions. Importance is calculated as the sum of absolute values from Figures 4.1a and 4.1b.

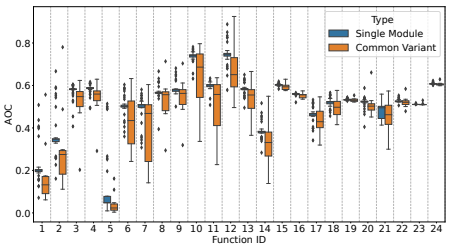


Figure 4.3: Performance distribution (AOC) of the 30 *single-module* DE variants and the 11 *common* DE variants from Table 4.3, for the 10-dimensional BBOB problems.

one module can be modified, some modules will likely have a much larger impact on the overall performance of the algorithm than others. The aggregation of maximum improvements and deteriorations for the selection of different module options is visualized in Figure 4.2. This figure shows the way in which these module importances are distributed across functions. For some functions, all *single-module* configurations perform similarly poorly, e.g. for F24, so no differences are detected. For most others, differences are present, with a clear impact on the choice of the base vector used for mutation (`Base`). In general, the mutation modules have relatively more impact than most others. Somewhat surprisingly, the impact of the adaptation methods for F, CR and population size λ is rather small. This might indicate that these settings work best when combined with other modules or more specific parameter settings. Also

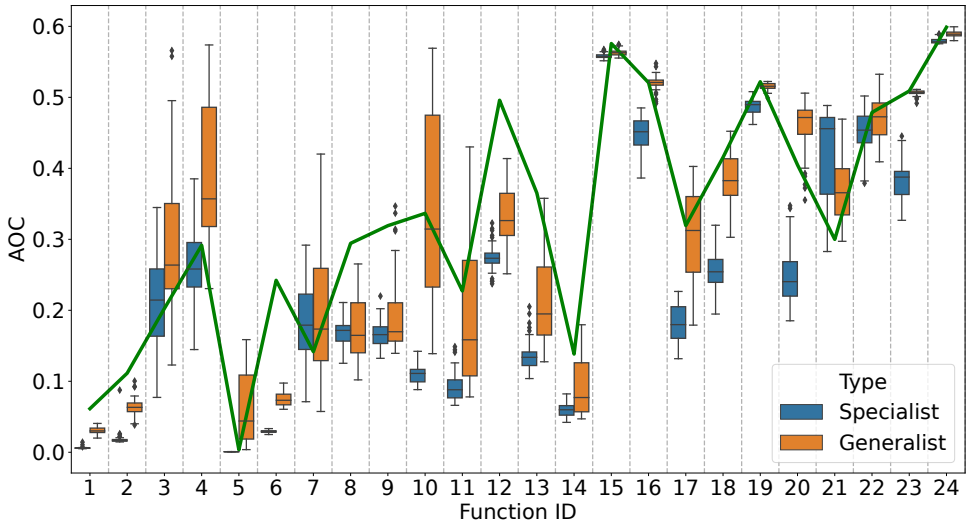


Figure 4.4: Performance distribution (AOC) of the configurations tuned for an individual function (*specialist*) and the configurations tuned for the full BBOB suite (*generalist*), for the 10-dimensional BBOB problems. The green line shows the AOC of the best DE version from the union of *single-module* DE and *common* DE variants from Table 4.3.

worth noting is that boundary correction is usually not impactful, with the exception of F5 (linear slope). For this function, the optimum lies directly on the boundary, so the boundary correction will be triggered often when close to the optimum, and thus have a large impact on the algorithm's performance [134]. All other BBOB functions are known not to have optima in the relative vicinity of domain boundaries [150].

To get insight into how hand-crafted DE versions, such as L-SHADE, compare to the *single-module* ones, we look at the performance distributions on the 24 BBOB problems. This is visualised in Figure 4.3. From this figure, we see that there is a fairly wide distribution of performance in both groups. Overall, the *common* DE variants seem to contain better configurations, although the set of configurations is relatively much smaller.

Performance of Tuned DE

Next, we compare the hand-crafted and *single-module* DE versions to those resulting from tuning the modular DE using irace. The resulting performance on the 10D BBOB problems is visualized in Figure 4.4. From this figure, we can see that generally, both

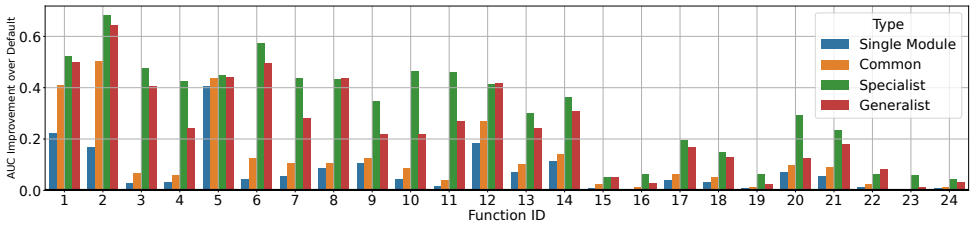


Figure 4.5: Relative improvement in AOC value between the best configuration of each type and the default setting, in 20D.

of the tuned DE settings outperform the hand-crafted ones. As expected, tuning for a particular function improves the performance on that function rather significantly.

Next, we aim to understand the impact of tuning relative to picking the best configuration from the set of *common* variants. To investigate this, we look at the relative gain in AOC over the default, for each set of configurations (*common* variants, *single-module* variants, *specialists* and *generalists*). For each type, we look at the performance of the best configuration of that type on each function and take the improvement it makes over the default setting. These improvements, for the 20D BBOB functions, are visualized in Figure 4.5. From this figure, we can see that the default setting performs particularly poorly on most of the unimodal problems, as even the best *single-module* configuration can outperform it significantly. However, this also shows the additional benefit which can be gained from tuning, which is particularly noticeable e.g. F3 and F4. We should also note that the performance gains shown here are slightly larger than those seen in Figure 4.4, which in turn are slightly larger than those achieved on the 5D version of these problems.

One more important note from Figure 4.4 is the wide distribution of AOC values. For the *generalist* configurations, this is natural, as configurations with different strengths can achieve similar performance when aggregated over the whole BBOB suite, resulting in a large per-function variance when grouped together. However, for the configurations tuned on a single function, the variance on some functions is still clearly visible. This might be caused by the inherent stochasticity of DE which potentially misleads the algorithm configurator when limited samples are available [247].

This variance might also explain why for F21 one of the hand-crafted DE variants outperforms almost all configurations which were tuned on that function. When considering Figure 4.3, we see that the performance might be considered an outlier, which performs much better than the remaining *common* variants. This observation might indicate that using the *common* DE variants to initialize irace could provide

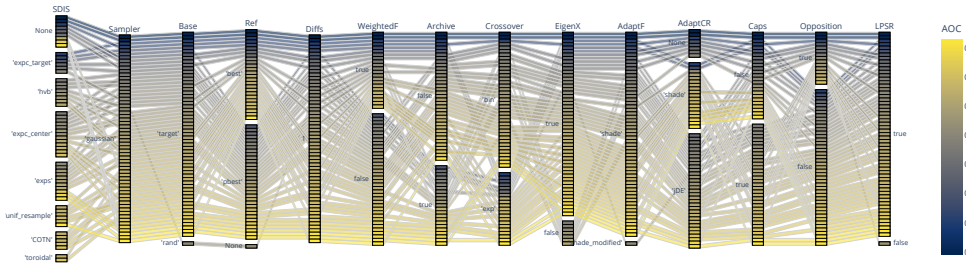


Figure 4.6: Parallel coordinate plot showing the modules activated in the elite configuration found across 10 runs of irace, on F19 in 5D. Configurations are colored based on normalized AOC.

some additional benefits over the current random sampling.

Analysis of Elite Configurations

Since multiple repetitions of irace are performed for each problem, we have a set of between 10 and 50 elite configurations for each setting. By analysing the commonalities between these elites, we can get an overview of the benefit of different parameter settings. This can be done on a global level by aggregating the activations of certain module options across runs and dimensionalities.

To understand which modules are selected often, we consider the module activations of a single function and visualise them as a parallel coordinate plot. Figure 4.6 shows this for Function 19 in 5D. In this figure, we see that all elite configurations make use of a Gaussian sampler for initialisation (`Sampler`). This makes sense when we consider the properties of F19 in more detail. In particular, we should note that for this function, the location of the optimum is not uniformly distributed in $[-4, 4]^D$ as for most BBOB problems, but it is instead limited to the shell of the hypersphere of radius 1, centred at the origin [150]. Because of this, a Gaussian initialisation will significantly outperform any uniform or low-discrepancy initialisation strategy.

In Figure 4.6 we also observe that all configurations, except one, make use of the SHADE-based adaptation for F (`AdaptF`), with 'target' based mutation mechanism (`Base`). This suggests that, unlike the common belief of adding many components in the mutation operator to deal with such problems, adaptation systems based on the history of successful control parameter values are beneficial for multimodal problems similar to F19, especially when combined with 'target'-based mutations and `Diffs`= 1.

4.2.2 Incremental Assessment of Module Performance: ModCMA

While the expansion of modular algorithms leads to an exponential increase in the number of possible configurations, we can still use algorithm configuration techniques to gain insight into the combinations of modules which perform well. When new modules are added we can retain the performance data from earlier experiments, and incrementally build upon this. Rather than looking at the new module in isolation, we use our algorithm configuration setup with the expanded search space and compare the resulting high-performing configurations to find interactions resulting from the modules inclusion.

We propose the following roadmap to formalize this procedure, which is designed to be generic, so that it can function with any modular algorithm, hyperparameter tuner, and performance metric:

1. Select a modular implementation of the base algorithm to which the new module has been added, a hyperparameter optimizer and a performance metric.

2. Collect a list of the existing modules and relevant hyperparameters (without the new module to assess). This will be the search space for the hyperparameter optimization.

3. Run the selected hyperparameter optimizer on this search space, ideally for a wide set of relevant benchmark functions. This data will then serve as the baseline performance.

4. Extend the original search space by including the new module to assess, and run the hyperparameter optimization on this extended search space (using the exact same setup as the baseline).

5. Compare the data from the baseline to the experiment with the extended search space. This should not only be done from a performance perspective, but also from the resulting configurations themselves. This allows for the analysis of potential interactions between modules.

Experimental Overview

To illustrate our proposed approach, we make use of the modular CMA-ES framework introduced in Section 4.1.2. Specifically, we consider the stepsize adaptation and

boundary correction, which were added on top of the previously implemented modules as shown in Table 4.1.

For our experiments, we stick with irace as our algorithm configurator. Four runs of irace are performed for each of the 24 objective functions in the BBOB single objective noiseless problem suite [96, 95], of which the first 5-dimensional function instance is used. Each run of irace is given a budget of 1 000 algorithm evaluations, which themselves have a budget of $10\,000 \cdot d$ function evaluations. We use the AOC attained by a run of a given configuration as the objective function value. Irace will designate one or more configurations as elites, which are the best configurations found. We validate the performance of these elite configurations by performing 25 validation runs, with the same random seeds for all configurations. We use the results of these runs to assess the final performance.

Following our roadmap, we define a baseline by tuning the existing modules from modCMA, which are shown in Table 4.2. In addition, we tune four continuous hyperparameters c_1, c_μ, c_c, and c_σ, which control the dynamics of the adaption of the covariance matrix (c_1, c_μ, and c_c) and of the step-size (c_σ).

We compare two experiments to our baseline where in addition to the existing modules, 1) several new step-size adaptation methods (see Section 4.1.2) are included, and 2) a new boundary correction module (see Section 4.1.2) is added to the tuned parameters. Both of these experiments use the same experimental setup as the baseline experiment (excluding the tuned parameters). Note that in the boundary correction experiment, the new step-size adaptation methods cannot be selected and vice versa.

Single Module Performance

Before considering our proposed method, we run a basic benchmarking experiment on each of the individual module options (including the new options). This is similar to the common approach of benchmarking a new module against a set of other algorithm variants. We show the resulting best single-module configurations (a.k.a. the virtual best solver, VBS for short) relative to the default CMA-ES in Table 4.4. In this table, we see that among the new modules, only two have been selected: MSR for F23 and m-XNES for F5. We can further look at the overall contributions of the newly introduced step-size settings by plotting the ECDF-curves over all functions, as done in Figure 4.7. In this figure, we can clearly see that most methods are quite competitive, with the only exception being xNES, which has an overall worse performance than the others. Overall, the MSR method seems to be quite effective, but there is no strict domination over the other settings.

Table 4.4: Table showing the AOC of the best single-module configuration for each function (VBS), compared to that of the default CMA-ES. The name of the solver corresponds to the module which is active, e.g. <module_name>_<option_value>. Note that these values does not include benefits from tuning the continuous hyperparameters, which are set to the default values for all configurations in this table.

Fid	VBS	AOC of VBS	AOC of Default	Improvement
1	elitist_True	247	326	24%
2	active_True	1 272	1 659	23%
3	local_restart_BIPOP	38 374	44 518	14%
4	local_restart_IPOP	41 746	44 613	6%
5	step_size_adaptation_m-xnes	43	63	31%
6	elitist_True	655	904	28%
7	step_size_adaptation_tpa	1 312	39 199	97%
8	base_sampler_halton	1 186	4 544	74%
9	base_sampler_sobol	959	2 470	61%
10	active_True	1 309	1 729	24%
11	active_True	1 162	1 749	34%
12	base_sampler_sobol	2 186	2 980	27%
13	active_True	1 627	2 191	26%
14	active_True	601	831	28%
15	local_restart_BIPOP	30 380	43 313	30%
16	local_restart_BIPOP	8 172	34 132	76%
17	threshold_convergence_True	12 464	26884	54%
18	threshold_convergence_True	15 764	33724	53%
19	mirrored_mirrored	33567	36 688	9%
20	threshold_convergence_True	36 482	40691	10%
21	local_restart_IPOP	38 028	40 371	6%
22	mirrored_mirrored	566	8 632	93%
23	step_size_adaptation_msr	11 060	34 433	68%
24	local_restart_IPOP	42 099	44 351	5%

Analysis and Results

In this section, we present the results of our hyper-parameter tuning experiment. We consider two paths to analyze the contributions of the newly introduced modules: the performance-perspective and the perspective of the selected modules. We start by examining our baseline. This is followed by an analysis of the performance-perspective and a deeper analysis of the selected modules.

Baseline

As mentioned in Section 4.2.2, we conduct a baseline tuning experiment.

Since we run four runs of irace for each function, this results in 4 sets of elites (each set has up to five configurations), for which we then perform the verification runs. We plot the distribution of the AOC for each of these configurations in Figure 4.8, in addition to this, the AOC of the default CMA-ES and the VBS is shown. From this figure, it is clear that the tuning of all parameters at once is much better than simply selecting a single-module variant, as is to be expected. This plot also highlights the

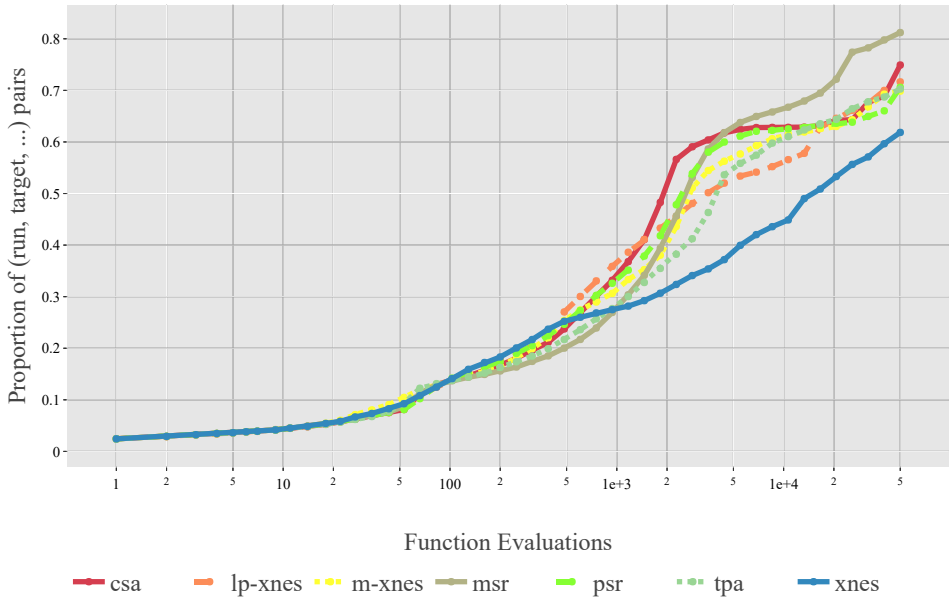


Figure 4.7: ECDF-curve of all single-module stepsize options. Figure generated using IOHanalyzer [255].

variance in performance of the final found configurations. There are two main reasons for this fact: the inherent stochasticity of the CMA-ES itself, and the large impact of the initially generated configurations of irace. We discuss these challenges in detail in Section 4.3.

From this baseline data, we can also study the resulting configurations themselves. This can be done by aggregating the modules which have been selected in the final elite configurations in the separate irace runs, as is visualized in Figure 4.9. In this figure, we can see that there is a large variability in the selected module options, which seems to indicate that they are all usable for at least some functions. One notable exception is the weights option "equal", which is chosen in less than 1% of configurations.

Performance analysis

First, we visualize the distributions of the AOC of the single best configuration found in each run of irace (based on the verification runs) in Figure 4.10. In this plot, we can see that the effect of introducing the new modules is quite mixed. For some functions, performance decreases (e.g., on F8) after introducing new modules, while for others

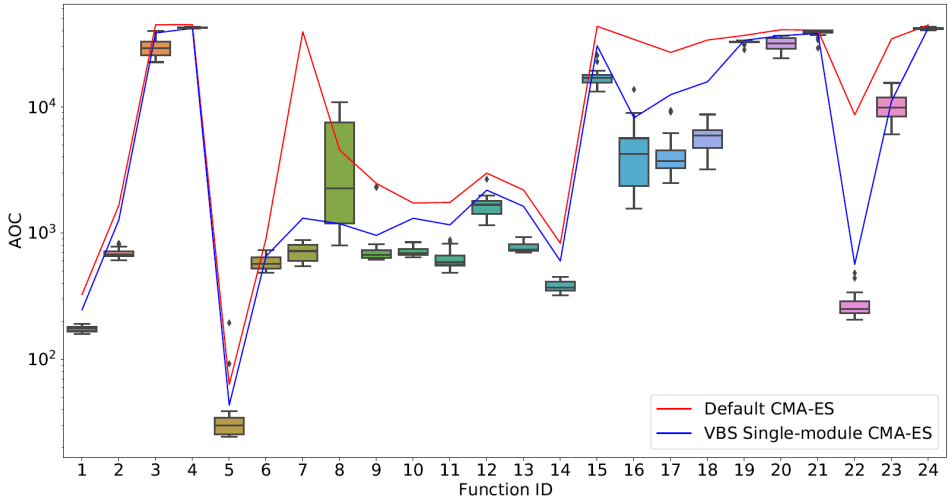


Figure 4.8: Distribution of the area over the ECDF curve for the final elite configuration of the baseline irace runs. All AOC's are averages of 25 verification runs. The VBS single-module configurations can be seen in Table 4.4.

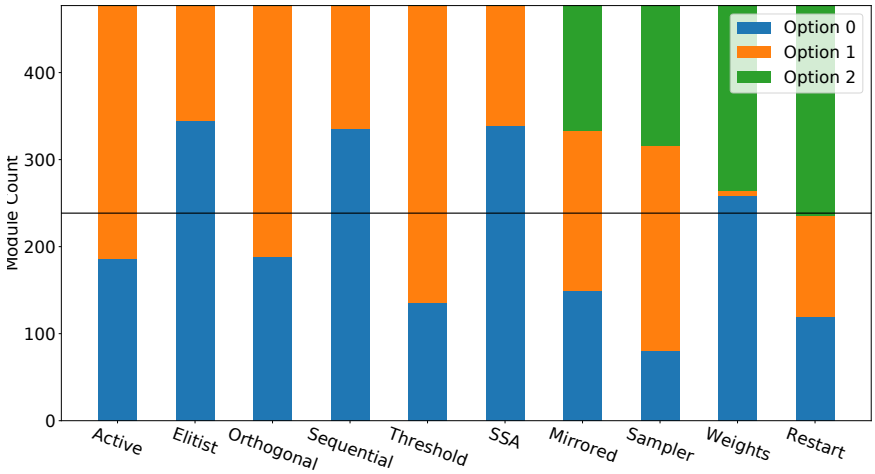


Figure 4.9: Module counts of all elites found in the baseline-experiment, over all 24 BBOB-functions. The option numbers correspond to those in Table 4.2

we see the desired improvement (e.g. on F23).

In order to better show these differences, we show in Figure 4.12 the AOC of the single best configurations found in both the SSA and bound-experiments relative to

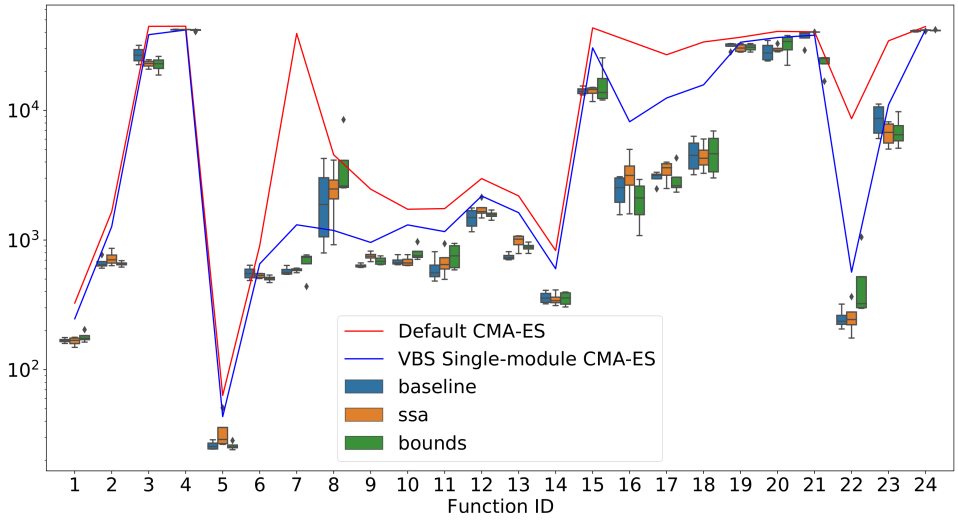


Figure 4.10: Distribution of the single best elites from the baseline and the tuning with the additional modules. AOC values are the result of averaging over 25 verification runs.

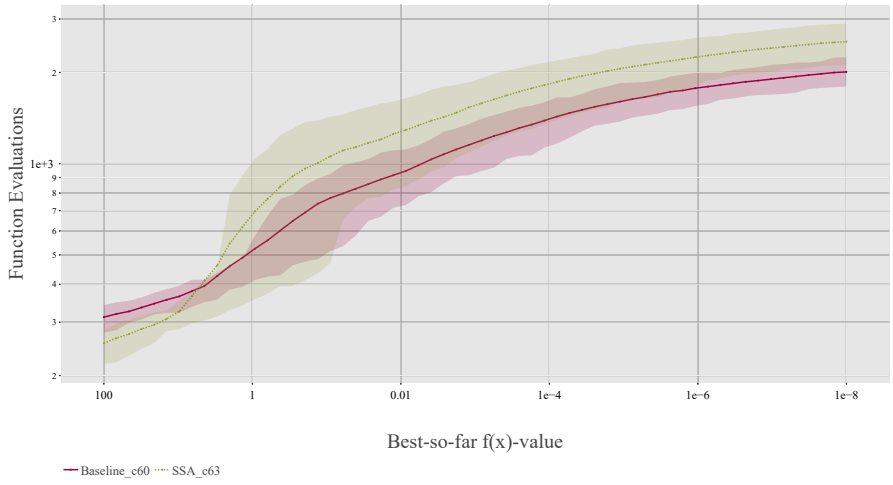


Figure 4.11: Comparison of the Expected Running Time of the best configurations found on F12 by both the baseline and the SSA experiments. Shaded areas indicate the outer quantiles (20-80).

the best configuration from the baseline. Here we observe a generally negative trend, with outliers in both directions. This seems to indicate that these new modules are not always beneficial to the final performance. For example, we can consider F12, where the configuration found by the baseline has an average AOC of 1 159, while the best configuration found when including the new SSA-methods in the search space reaches an average AOC of 1 480. We show the expected running time of these two configurations in Figure 4.11, where we can clearly observe this difference. However, we can observe a large variance between runs, which can partly explain poor performance. Indeed, if we look at the average AOC as found during the irace run (instead of the later verification runs), the difference between these two configurations is only 7%, even though the distance between them in the verification runs is much larger. This leads to an important observation about the assessment of the new algorithmic modules: when judging results purely from the average performance measures, it is necessary to also consider the overall variability of the experiment, as well as the inherent stochasticity of the base algorithm.

We perform the same procedure for the boundary correction methods. The impact of this module is expected to be smaller, since for most of the "easier" functions, the boundary condition is rarely violated. For some of the more challenging functions however, the penalty value given by BBOB function itself might not be sufficient to "guide" the algorithm back in bounds, but an explicit boundary correction could be beneficial in these cases. We can see that this seems to indeed be the case in Figure 4.12, where on the more complex functions, e.g., F21, the performance is improved when the boundary correction module is tuned.

In Figure 4.12, we also see that the inclusion of the new SSA methods manages to improve the overall performance for some functions. As an example, on F23 we saw an improvement of 17.1% over the best baseline configuration. If we consider all four elite configurations and compare the average performance differences, the average improvement is even higher, at 22.3%. The stability of this improvement is promising, but in order to fully grasp how the inclusion of the new SSA mechanisms leads to this improvement, we need to analyze the selected modules across these different experiments.

Module Analysis

We have seen that the performance of the elite configuration found on F23 improves when we include the new SSA modules in the search space. In order to identify what this performance can tell us about the new modules themselves, we should study the

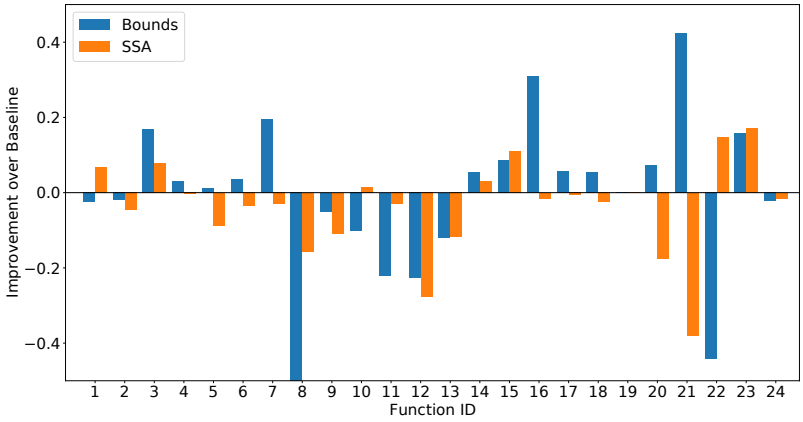


Figure 4.12: The relative improvements per function of the best configuration found by irace relative to the baseline experiment's best configuration.

configurations in more detail. The obvious way to see the difference is by looking at how often the new module options have been selected in the final elite configurations. Over 20 elites, the PSR update was selected 14 times, MSR once, and CSA five times. This shows that these new modules are indeed used in successful configurations. To see how the inclusion of these module options changes the interactions with the other modules, we look at the combined module activation plot, which is shown in Figure 4.13. From this figure, we can see that there are some interesting differences between the two sets of configurations: the options for the restart and mirrored module are not as uniform when using the new SSA methods, and the weights option is changed completely. These observations show that there is a clear interplay between these modules.

We can extend this module analysis to all functions by aggregating the most important differences found between the baseline and SSA-experiments. First, we can plot how often each new module option is selected in the elites for each function, as is done in Figure 4.14. We can use the same principle to study the interaction with the other modules. For the binary modules, we can directly capture the module difference by looking at which modules occur more or less often in the final set of elites, as is visualized in Figure 4.15. From this figure, it becomes clear that the elites on some functions are barely affected by the inclusion of the new modules, while others require completely different module settings to properly exploit the changed search space.

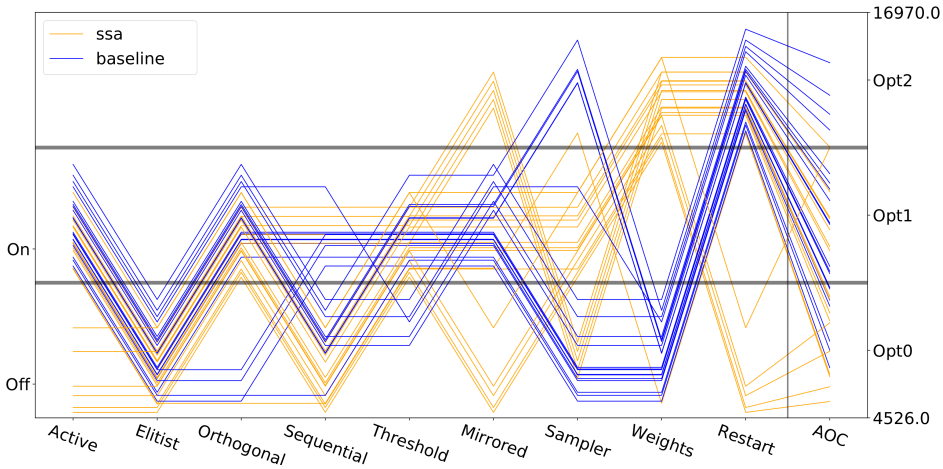


Figure 4.13: Combined module activation plot for the elites found in the baseline and SSA experiments, for function 23. The lower the line, the better its performance, scaled within each band according to the AOC. The option numbers correspond to those in Table 4.2.

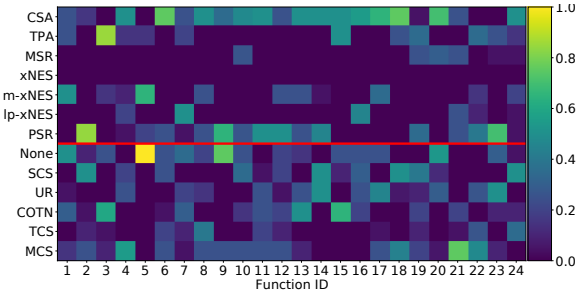


Figure 4.14: Heatmap showing the fraction of the elite configuration in which each of the options for either SSA (top) or boundary correction (bottom) are active.

4.3 Selected Challenges in Algorithm Configuration

While Section 4.2 highlights some of the benefits of algorithm configuration in the context of modular algorithms, our approach still faces some inherent limitations. In this section, we discuss some selected challenges, and illustrate specifically how the inherent stochasticity of the iterative optimization heuristics impacts the results of algorithm configuration.

The cost of tuning The first thing we should note about the incremental tuning approach is that only considering the final elite configurations does not tell the full story of a module's contribution. As noted previously, introducing a new module

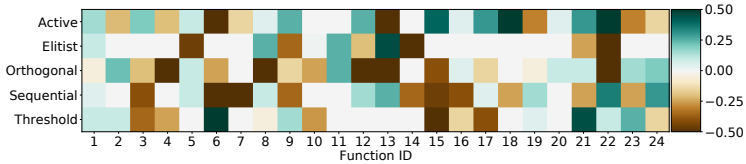


Figure 4.15: Heatmap showing difference in the fraction of the elite configuration in which each the of the binary modules are active, between the baseline and the SSA experiment. Positive values indicate a module is turned on more often in the SSA experiments.

increases the size and complexity of the search space, which has a large impact on the hyperparameter tuning task. If a module is very dependent on the settings of other hyperparameters, this can lead to deterioration of the final results, since the initially sampled configurations are likely to have worse performance than those in the baseline. This is visualized in Figure 4.16, where this is clearly seen on function F5. This is a linear slope function, but the BBOB-specification does not include a sufficient penalty for leaving the search space. As a result, an algorithm which quickly leaves the search space will reach the required objective value very quickly. Thus, when adding boundary correction methods, five out of six random configurations are not able to abuse this loophole, leading to a worse initial performance. While for F5, the function is simple enough that the good configurations can still be found (and the inclusion of the default CMA-ES settings in the initial population means that there is always at least one good configuration present), the same issue exists to a lesser extent in other functions. Figure 4.16 also shows that the "tunability" of modules on different functions varies widely. For instance, on functions F16 - F18, the spread of AOC values is significantly larger than those on functions F19 - F21, suggesting that it is relatively more difficult to tune the modules in the latter since the tuner will very likely take a considerably larger budget to identify optimal configurations. Also, while on some functions it is trivial to get improvement (e.g., F7) over the default CMA-ES, it is a lot more challenging on others, for example on functions F16 - F18.

Limits of the per-instance analysis: As is commonly done, our performance assessment is done on a per-instance basis. While this can be preferred over tuning for large sets of functions/instances [9], it does have some drawbacks. Specifically, if a module is designed to have a good performance over a wide set of functions, but other settings exist for each individual function which outperform it, this new module would not be seen as beneficial. Because of this, we argue that module assessment

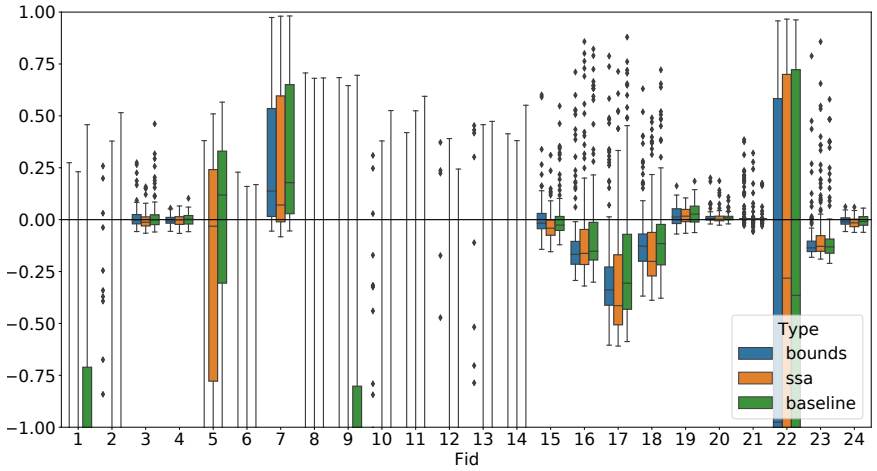


Figure 4.16: Distribution of the relative AOC values found in the initial race of irace (relative to the default CMA-ES configuration; positive values equate to lower AOC.)

by hyperparameter tuning should not replace the traditional assessments, but rather complement it for more in-depth, per-instance analysis.

Influence and stochasticity of the hyperparameter tuning: While we showed that assessing the impact of an algorithmic component by using a hyperparameter tuning approach provides useful insights, there are several factors which can complicate this approach. Since hyperparameter tuning is a very challenging problem, with many different approaches to solving it, the kind of tuner used will have a large impact on the resulting assessment [221]. In this chapter, we used irace, which tends to focus on converging to a single configuration, instead of covering a large set of different solutions. This necessitates running multiple repetitions of the irace procedure itself, as the initialization might otherwise have too much impact on the final configurations. This can quickly become computationally expensive.

4.3.1 Noise in Algorithm Configuration

The final, and perhaps most relevant, limitation we discuss is the inherent stochasticity in the algorithm which we are using. The amount of variance of the algorithm configurations on a certain function has a large impact on the search procedure of irace. Since we generally end up selecting elites based on average performance, we are inherently underestimating the AOC of the final configuration. Even though irace largely mitigates this by using statistical testing in the races to decide when to discard

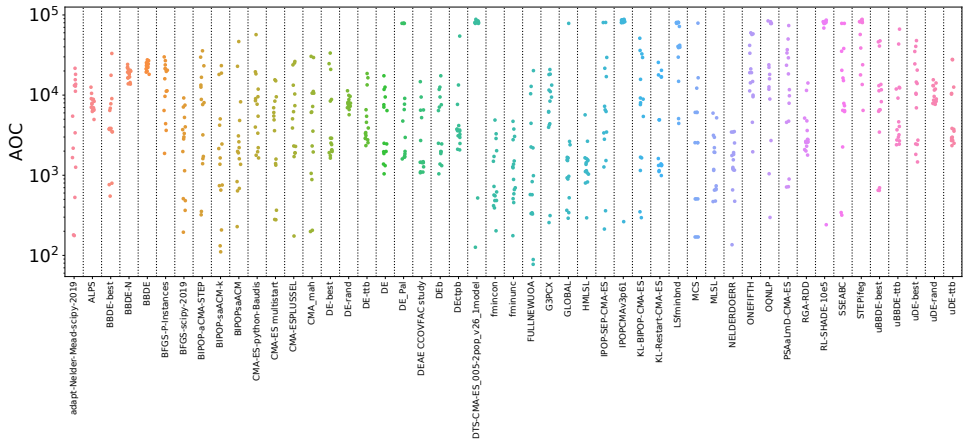


Figure 4.17: Distribution of the AOC values of 15 independent runs of algorithms from the BBOB archive [4] on F21 in 5D.

configurations, there will always be some degree of underestimation of the performance (for example, the median performance in the verification runs from our Section on modCMA is 3.4% worse than predicted from the irace runs).

In this section, we highlight this challenge inherent in comparing the performance of stochastic optimization algorithms. While our focus is on algorithm configuration methods, we show that, for several cases including standard benchmarking-based comparisons, the currently recommended values for the number of samples and testing procedure can lead to mistakes. We show that the distribution of performance values has a large impact on algorithm configuration methods, indicating that there is not one method of performance comparison that dominates all others. Importantly, our results demonstrate that we must identify better ways to handle the stochasticity of iterative optimization heuristics when applying algorithm configuration methods.

Why 15 runs are not enough

While it is clear that any aggregated performance measure used to compare randomized algorithms is an empirical estimation of their true performance, the variance of this estimation is not necessarily equal for all algorithms on all functions. However, for a practical benchmarking setup, this nuance is often ignored in favour of simpler guidelines, such as aggregating a fixed number of *samples* (i.e., individual performance values from independent runs) for each algorithm on each function. The usual recommendation of 15 samples [95] is often enough to make clear decisions on simple

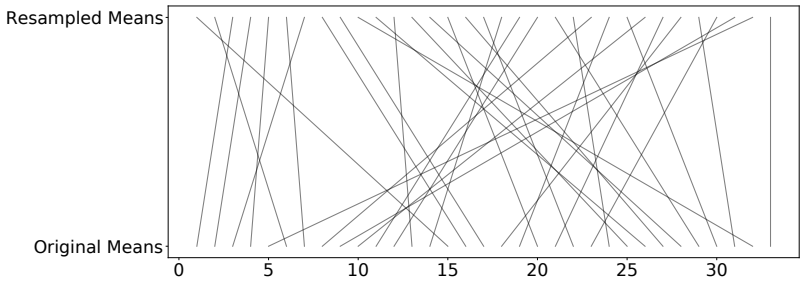


Figure 4.18: Changes in ranking of 33 **random modCMA configurations** based on calculating the mean over a sample of 15 AOC values (*Resampled Means*) versus the 200 verification run samples (*Original Means*), on F21.

uni-modal functions, but the situation is much less clear on more challenging optimization problems.

We illustrate the significant variation in performance between runs by showing in Figure 4.17 the distribution of 15 independent AOC-values for a wide variety of algorithms from the BBOB-repository (https://numbbo.github.io/data-archive/bbob/) on F21 in 5D. This figure also shows that the normality assumption, commonly taken for granted in benchmarking studies, is not well supported by the apparent distribution of the 15 performance values shown for each algorithm.

For some algorithms, the performance distributions even appear to show signs of bi-modality. As such, any analyses made based on this set of samples should be treated with care. While this large amount of variance is very pronounced in F21, it is not limited to this function, as other functions display similar effects but to a slightly lesser extent.

The impact of performance variability can potentially be even larger when considering the task of algorithm configuration. It has previously been observed that the performance of an algorithm configuration on verification runs can differ significantly from the runs performed during the configuration task [51].

We illustrate this effect by showing in Figure 4.18 the changes in the ranking of the 33 **high-quality modCMA configurations** described in Section 4.1.2 when calculating mean performance using a small sample size (15) and a larger number of verification runs (200) on F21.

While this might be considered a rather extreme case, it is by no means the only scenario in which behaviours like this can occur. Since algorithm configuration often generates similarly performing configurations near the end of a configuration run (while exploiting promising regions), making decisions about which configuration to select

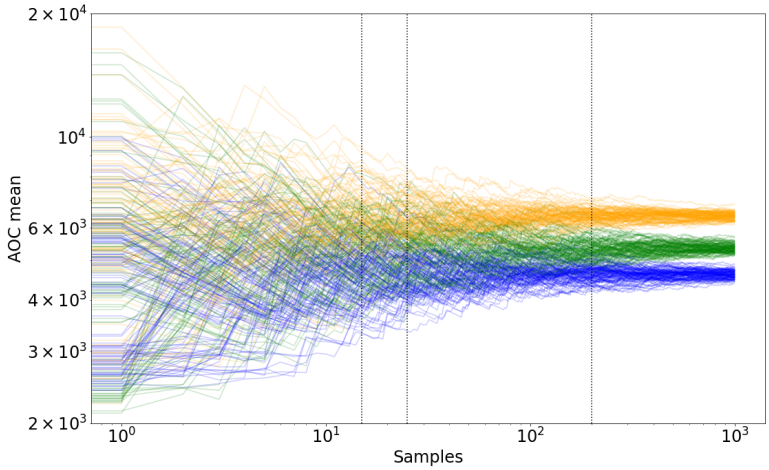


Figure 4.19: Evolution of the cumulative mean over sample sizes of 3 selected **high-quality modCMA configurations** on F18. The vertical lines indicate sample sizes 15, 25 and 200 respectively. Means are based on sampling with replacement from the original 200 samples of each configuration.

might become very noisy when using relatively low sample sizes. This phenomenon is exemplified in Figure 4.19, where we show the evolution of the mean AOC of 3 selected **high-quality modCMA configurations** relative to an incremental number of AOC values. Each horizontal line of the same color corresponds to the cumulative mean of a sequence of values sampled with replacement from the same 200 AUC values. Despite sampling from the same 200 AUC values, the variance of the means of 15 and 25 samples is quite large and those means often poorly estimate the true mean performance.

In practice, making an incorrect decision between two configurations matters less when their true performance is very similar. However, when the set of configurations which are being compared increases in size, the risk of making incorrect decisions between more distinct configurations could potentially grow as well. In some situations in algorithm configuration tasks, we have observed significant differences between the performance of the selected elite configurations, and the best one from all configurations sampled according to the verification runs.

In Figure 4.20, we show the distribution of AOC values for each configuration sampled during a run of irace on each of the 24 BBOB functions. The performance of each configuration is based on the mean of 200 verification runs, and the plot

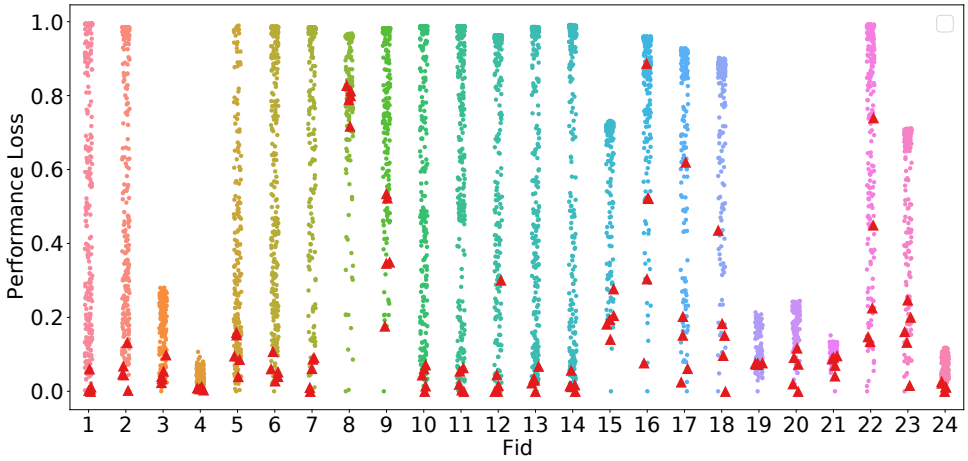


Figure 4.20: Performance losses between all modCMA configurations explored during the execution of one irace run on each function, and the best one from this set of configurations. The final elites of each irace run are marked in larger red triangles. All data points shown are based on 200 independent samples.

shows the relative performance loss to the best of these means. The (up to five) elite configurations returned by irace are marked with a red triangle. The lowest of these elites corresponds to the level of performance loss achieved by irace compared to the best-performing configuration sampled during the configuration process. From this figure, it can be seen that for some of the more complex functions, a 10% performance loss or more can occur, clearly demonstrating that the variability of performance can severely hinder the outcome of the configuration efforts.

Impact on Benchmarking

To simulate a common algorithm comparison scenario, we make use of the set of 33 **high-quality modCMA configurations** from Section 4.1.2 and simulate the benchmarking procedure by randomly re-sampling with replacement AOC values, for sample sizes 2, 5, 10, 15, 25 and 50 from the set of 200. Then, we select the configuration with the best mean for each particular sample size as the winner, and compare its true performance (i.e., over 200 runs) to that of the actual best configuration to get an estimate for the performance loss. This process is repeated 5000 times for each sample size and each function, and the resulting performance loss per function is shown in Figure 4.21. We conclude that using means to determine the best-performing algo-

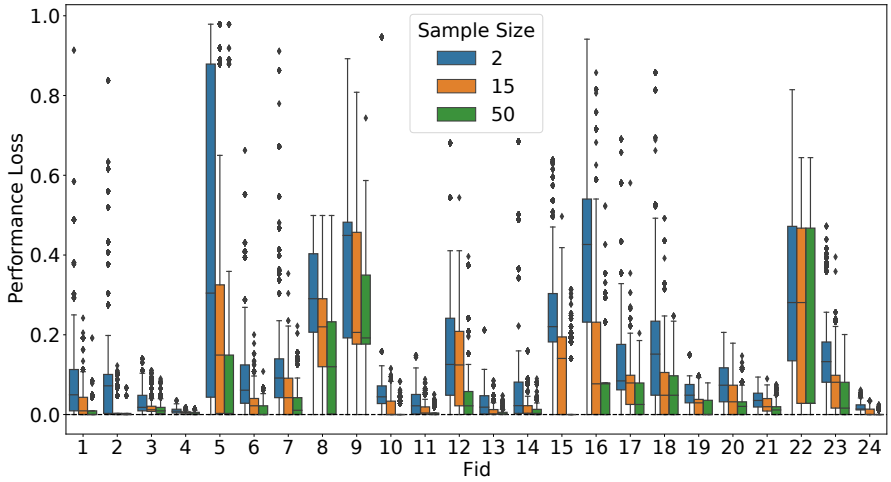


Figure 4.21: Performance loss, relative to the configuration with the best mean calculated over 200 samples, when comparing 33 **high-quality modCMA configurations** based on mean calculated from different number of samples. Each bar represents 5000 repetitions of the experiment.

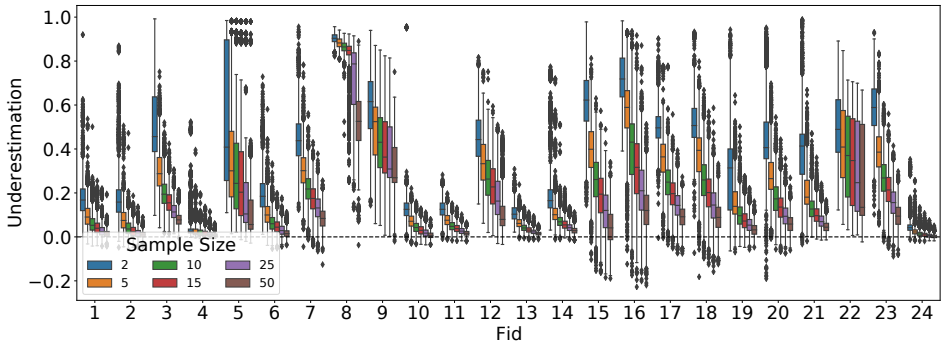


Figure 4.22: Underestimation when comparing 33 **high-quality modCMA configurations** based on mean calculated from different numbers of samples. Each bar represents 5000 repetitions of the experiment.

rithm is not always reliable, and can lead to selecting configurations that are clearly sub-optimal. Many benchmarking studies use non-parametric statistical tests to assess significant differences without assuming normality, yet they still rely on the comparison of means to rank the algorithms. While we can see that increasing the used sample size is always beneficial, even using as many as 50 samples can still see performance losses of 10% and more on some functions.

One possible explanation for these results is that, when we are determining the best algorithm from a large set of algorithms of wide performance variability, our decision is prone to underestimate the true mean due to the small sample size, i.e., we might "luckily" sample many good values for an algorithm with sub-optimal performance.

We quantify this impact by calculating, for each selected configuration and a given sample size, the *underestimation error*, that is, the relative error of the mean estimated from the selected samples relative to its true mean performance (based on the 200 verification runs). Positive values indicate that the sample mean is lower, i.e., better, than the true mean. We plot in Figure 4.22 the underestimation error for **high-quality modCMA configurations**.

We observe large underestimation errors in almost all functions. In some functions, such as F8, the underestimation error is large even for a sample size of 50. We also notice that large underestimation errors in Figure 4.22 often coincide with a large performance loss seen in Figure 4.21. This observation can be explained by looking in more detail at the performance distribution of the used configurations on a particular function, as is done in Figure 4.23 for F8.

We see in this figure that all configurations have a fraction of runs where the AOC value is very large, indicating that these were very poorly performing runs. When calculating the mean value of a configuration from a limited number of samples, if none of these poor runs appears in the samples, then the mean of the configuration will be lower than its true mean, leading to the large underestimation seen in Figure 4.22.

Additionally, since the difference in a configuration's performance seen during configuration and its true mean is often larger than the difference in the means of configurations as estimated from a small number of samples, a relatively poor-performing configuration can end up being chosen simply because it got 'lucky', which can explain the performance losses we observed previously.

Another common way in which the mean is used in benchmarking is in the basic pairwise comparison scenario, where two algorithms are directly compared to each other. To investigate this scenario, we simulate pairwise comparisons based on a limited sample size, and correlate the decisions made by the pairwise comparison to the

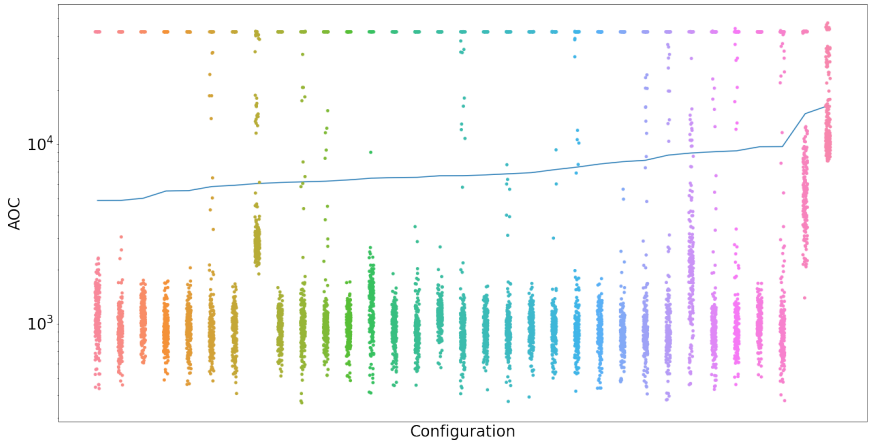
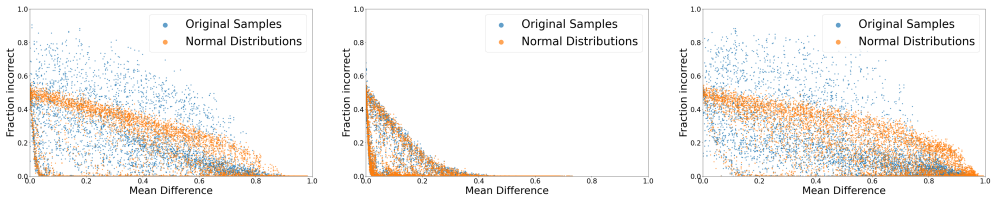
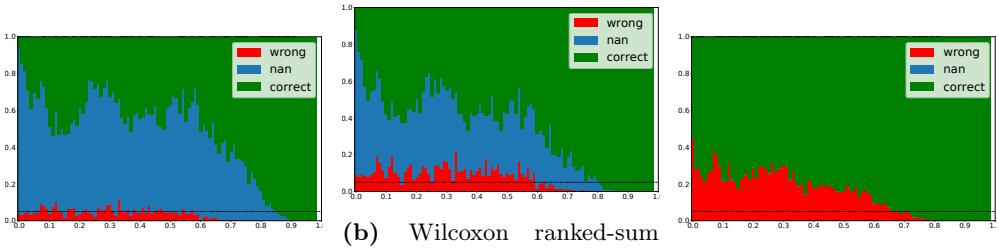


Figure 4.23: Distribution of AOC values of 200 individual runs of **high-quality mod-CMA configurations** on F8. The line indicates the mean AOC value of each configuration, and is the basis for the sorting on the x-axis.



(a) F9 **(b)** F15 **(c)** F22

Figure 4.24: Fraction of incorrect decisions when using the sample mean to compare pairs of modCMA configurations. Each subplot contains 10 000 points. Each point compares two configurations selected uniformly at random from the available configurations. The x-axis indicates the normalized difference between their true means (based on the 200 AOC values per configuration). The y-axis indicates the fraction of incorrect decisions based on 500 independent samplings of 15 AUC values for each of the two selected configurations. *Original samples* refers to sampling with replacement from the 200 AOC values available, while *Normal distributions* refers to sampling values from a normal distribution with the same mean and standard deviation as the 200 values of the corresponding configuration.

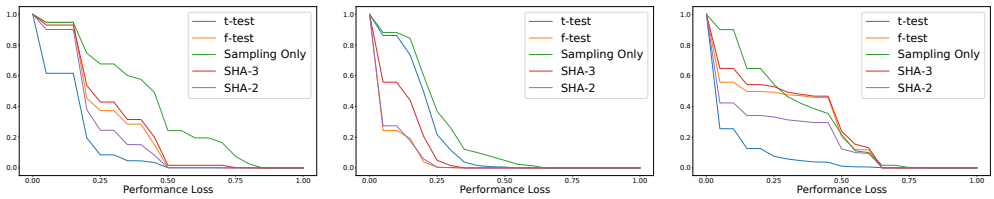


(a) *t*-test **(b)** Wilcoxon ranked-sum test **(c)** Comparison means

Figure 4.25: Correctness of decisions made in pairwise comparisons between modular CMA-ES configurations on F9, using different procedures. The x-axis shows the relative difference in true mean between the selected configurations. The y-axis shows the fraction of comparisons, out of 500 repetitions, that the decision was correct, incorrect or inconclusive (nan) when comparing configurations with this difference. Each repetition samples 15 values out of the 200 available for each configuration compared.

difference in true means between the selected configurations. To achieve this simulation, we use the full set of modCMA configurations generated during an irace run, which totals over 200 configurations on each function. From this set of configurations, we take 10 000 pairs, drawn uniformly at random, to perform the pairwise comparison. Then, for each pair of configurations, we sample a number of AOC values from the 200 values available, calculate the sample means and compare them to decide which configuration is the best. The comparison is *correct* if it gives the same conclusion as comparing the true means. We repeat the sampling and comparison step 500 times to calculate the fraction of times that the comparison is correct. The results of this experiment on F9, F15, and F22, with sample size 15, are displayed in Figure 4.24.

We observe in this figure that, as expected, the fraction of incorrect decisions decreases when the difference in true means increases. However, the decrease is much faster for F15 than for F9 or F22. There are also notable differences when comparing the fraction of incorrect decisions generated by sampling with replacement from the 200 AOC values available (Original samples) versus sampling values from the normal distribution that has the same mean and standard deviation as those 200 values. These distributions are almost identical for F15 but different for F9 and F22, which suggests that the fraction of incorrect decisions made by comparing means for F9 and F22 is impacted by the non-normality of the sample distribution.



(a) F9 **(b)** F15 **(c)** F22

Figure 4.26: Cumulative performance loss of 5 variants of the racing procedure using *FirstTest* = 2: t-test, Friedman-test, sampling and selecting based on mean, and successive halving with reduction factors 2 and 3.

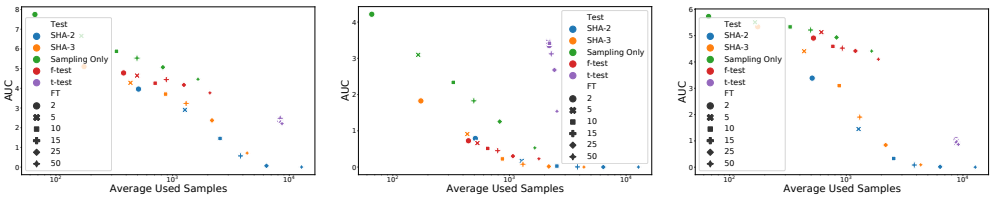
Statistical testing

When considering pairwise comparisons between algorithms, we often use statistical tests to determine if one algorithm outperforms the other. Two of the most common tests are the t-test and the non-parametric Wilcoxon rank-sum test.

To more closely analyze these two testing procedures, we re-sample with replacement, for sample size 15, from the set of 200 AOC values of the 33 **high-quality modCMA configurations**. Then, we apply a one-sided t-test to the samples of size 15 and measure the fraction of pairs in which the test was "correct", "incorrect" or "inconclusive". We consider here that the test is "incorrect" when, for a pair of algorithms A and B, the null hypothesis that A has a lower mean than B is rejected but the mean of A is indeed lower than the mean of B based on the 200 values. When neither of the two one-sided null hypotheses (A has a lower mean than B nor B has a lower mean than A) are rejected, the test is considered "inconclusive".

We zoom in on function F9 in Figure 4.25, and look at the difference between making decisions based on means, t-test and Wilcoxon rank-sum test. We note that both statistical tests show an error rate that is larger than α for pairs of configurations with a difference in means up to 60%. We also note that even though the t-test is less frequently incorrect, it is also more frequently inconclusive compared to the Wilcoxon rank-sum test, even for configurations whose means differ significantly.

Inconclusiveness is not a factor when comparing based on means, but that comes with the cost of making more incorrect decisions as well. While the number of incorrect decisions decreases when adding more samples, the overall observations for the three comparison procedures remain similar.



(a) F9 **(b)** F15 **(c)** F22

Figure 4.27: Comparison of AUC value of Cumulated performance loss (Figure 4.26), relative to the average amount of samples used by each process.

Racing

To investigate the impact of performance variability on algorithm configuration, we focus on the racing procedures used by irace, which we simulate using the **high-quality modCMA configurations** from Section 4.1.2. In particular, we consider two variants of the racing procedure [160] using either the t-test or the Friedman-test. In addition to these racing variants, we also consider two variants of Successive HAlving (SHA) [120] with reduction factors 2 and 3, respectively. For the races using statistical tests, we loosen the total budget restriction, which is usually used as stopping criteria [152], (e.g., in irace) to 10 000 total samples, which means we continue the race until 5 or fewer configurations remain, or until we exceed 10 000 sampled runs ('target runs' in irace terminology). We simulate this race 1000 times for each function and several values of *FirstTest*, and show the resulting performance loss for F9, F15, and F22 in Figure 4.26. In this figure, the performance loss is defined as the difference in the true mean of the best elite (configuration with the best sampled mean during the race) against the best configuration which was present in the race.

The cumulative performance loss is compared for both the Friedman-test and t-test variants of the racing procedure, as well as a naive *sampling-only* approach that selects based on means after *FirstTest* samples have been collected for each configuration.

When comparing the different approaches, we note that there is not a clear winner across all functions and values of *FirstTest*. Interestingly, for some of the functions where Figure 4.20 shows the largest performance losses of irace elites, the races using the Friedman test seem to perform relatively poorly. This might indicate that for these functions, we could regain some of the lost performance, if it can be detected during the algorithm configuration that a different testing strategy would be required.

From Figure 4.26, we can clearly see that any variant of racing or SHA is much more reliable than the *sampling-only* approach. However, racing uses more total samples,

since it adds runs when needed, while the sampling-only approach uses a fixed number of samples. The SHA method uses a fixed number of samples as well, but this number is significantly larger than the sampling-only approach and depends on the reduction factor used.

In order to account for the differences in total budget, we summarize cumulative performance loss curves, such as those in Figure 4.26, using their corresponding AUC values, and plot these AUC values against the total samples used in Figure 4.27.

Here, we see an explanation for the great performance of the t-test: it uses significantly more samples for the same *FirsTest* value than any of the other methods. This can happen when the test can not make any conclusive decision between the configurations and thus fails to reject enough configurations to reach the 5 elites, using up the full budget of 10 000 evaluations in the process. This matches our findings from Figure 4.25a, where we could see that the pairwise t-test often does not give any decision, even when the difference in true means between configurations is relatively large.