

Modular Differential Evolution

Diederick Vermetten

Leiden Institute for Advanced Computer Science
Leiden, The Netherlands
d.l.vermetten@liacs.leidenuniv.nl

Anna V. Kononova

Leiden Institute for Advanced Computer Science
Leiden, The Netherlands
a.kononova@liacs.leidenuniv.nl

Fabio Caraffini

Swansea University
Swansea, United Kingdom
fabio.caraffini@swansea.ac.uk

Thomas Bäck

Leiden Institute for Advanced Computer Science
Leiden, The Netherlands
t.h.w.back@liacs.leidenuniv.nl

ABSTRACT

New contributions in the field of iterative optimisation heuristics are often made in an iterative manner. Novel algorithmic ideas are not proposed in isolation, but usually as an extension of a preexisting algorithm. Although these contributions are often compared to the base algorithm, it is challenging to make fair comparisons between larger sets of algorithm variants. This happens because even small changes in the experimental setup, parameter settings, or implementation details can cause results to become incomparable. Modular algorithms offer a way to overcome these challenges. By implementing the algorithmic modifications into a common framework, many algorithm variants can be compared, while ensuring that implementation details match in all versions.

In this work, we propose a version of a modular framework for the popular Differential Evolution (DE) algorithm. We show that this modular approach not only aids in comparison, but also allows for a much more detailed exploration of the space of possible DE variants. This is illustrated by showing that tuning the settings of modular DE vastly outperforms a set of commonly used DE versions which have been recreated in our framework. We then investigate these tuned algorithms in detail, highlighting the relation between modules and performance on particular problems.

CCS CONCEPTS

• **Theory of computation** → **Design and analysis of algorithms**;
Bio-inspired optimization.

KEYWORDS

Differential Evolution, Benchmarking, Modular Algorithms, Algorithm Configuration

1 INTRODUCTION

Science is an iterative process. This is easily seen in action in the evolutionary computation community. Most of the contributions made to the state-of-the-art are incremental modifications on a set of core algorithms. This process of researchers building upon the work of others is highly beneficial to the community as a whole, allowing further specialisation of algorithms. However, there are some obstacles to the integration and comparison of many of these proposed modifications.

One particular issue is 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 [1, 6]. Issues such as these have raised questions regarding the reproducibility of research in computer science as a whole, and evolutionary computation is no exception [29]. 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 which can easily be swapped out.

The ideas behind modular algorithms have been around for decades [7, 31, 32], but although they have been shown to be extremely useful [17], 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 [14, 39] and the Particle Swarm Optimisation framework [8]. These works highlight the benefits of modular algorithms not only for fair comparisons, but also hint at the potential to study interactions between modules.

In this work, we propose a first step towards a modular version of Differential Evolution (DE), a heuristic originally introduced in [33, 37] to optimise a single-objective real-valued fitting problem, and whose design took into consideration elements from evolutionary algorithms and swarm intelligence optimisation (see [11, 41] 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 [13]. 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.

2 DIFFERENTIAL EVOLUTION

The original DE framework offers a somewhat modular structure by design, where DE variants are characterised by the notation DE/x/y/z-SDIS. Here, we stress the importance of considering the SDIS (Strategy for Dealing with Infeasible Solutions) as a non-optional operator of a heuristic algorithm [24]. To understand this notation and describe the algorithm in a compact way, some DE jargon described below is needed.

DE is a population-based algorithm where one iteration (i.e. a generation) is completed by perturbing candidate solutions in the population one at a time - from the first to the last individual. This process starts from some initial population and is iterated for multiple generations depending on the available computational budget. The order in which individuals are selected to undergo the variation operators is maintained. In this context, a selected solution is referred to as the target individual x_t . This individual undergoes a crossover operator z , which first requires the preparation of a second individual, called the mutant individual x_m , to produce an offspring x_o called the trial vector. This solution might contain infeasible components inherited from mutant and therefore must be fed to a SDIS operator. Immediately after generating a feasible trial solution, target and trial compete to enter the next population (note that the swap takes place only after the generation cycle is complete in the original DE framework - i.e., after each individual has produced a trial solution).

DE mutation operator x/y is executed to obtain a mutant solution. This is the operator that gives the name to the algorithm, as it is based on the idea of ‘moving’ a point (usually a randomly chosen solution from the population, i.e. $x=\text{rand}$, or the best so far individual, i.e. $x=\text{best}$, or a combination of individuals that can include the target - see [34] and Section 3.2) by adding a number y of scaled ‘difference’ vectors. This results in a linear combination of individuals because the difference vectors are obtained by subtracting two randomly chosen individuals. Note that all solutions involved in x/y must be distinct and feasible individuals or combinations of individuals. Furthermore, in the case where x is chosen as a vector originating from one individual to another, for example rand-to-best or current-to-rand , some recombination is already present at the mutation level, leaving the option of dropping the crossover operator. To function, this simple algorithmic structure only requires setting a population size λ , a scale factor $F \in [0, 2]$ for the mutation operator, and a crossover rate $CR \in (0, 1]$ for the crossover. A detailed pseudocode of this framework can be found in [24] and for more details and examples of DE operators, we refer to [13].

3 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 [9]. In this paper, 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 1.

3.1 Initialisation

To create the initial population, we implemented several sampling strategies (Sampler, see Table 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 [36] (Opposition), where each time we generate an individual for the initial population, we also generate its mirror image around the origin.

3.2 Mutation

The mutation operator has been the focus of many modifications of DE, see, e.g. [10, 13, 18, 23, 44]. To capture the most established mutation variations of the kind x/y (as described in Section 2), 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 $x = \text{Base}$. Conversely, when Ref is one of the admissible reference solutions displayed in Table 1, a scaled version of the vector directed from target to the reference point is generated and added to Base, i.e. $\text{Base} + F(\text{Ref} - \text{target})$. Therefore, when Ref is not none, one obtains any of the classic strategies of the kind $x = \text{target-Refs}$, 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 [4].

One more modification makes use of an archive of external solutions, as done, e.g., in [44], 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.

3.3 Crossover

The classical studies in DE generally consider two types of crossover: binomial ($z=\text{bin}$) and exponential ($z=\text{exp}$) [34], 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 [20], 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 [16] to calculate the eigenvalues and eigenvectors. These

Table 1: Available modules and parameters, 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, ..., 200} (4+ $\lfloor (3 \log(D)) \rfloor$)
Parameter	Scale factor	F	r	[0, 2] (0.5)
Parameter	Crossover rate	CR	r	[0, 1] (0.5)

Table 2: 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	[38] Base : target, Ref : pbest Base : target, Ref : pbest	adaptive		$18 \cdot D$	LPSR : true, Archive : true, AdaptF : shade, AdaptCR : shade
SHADE		adaptive		$10 \cdot D$	Archive : true, AdaptF : shade, AdaptCR : shade
DAS1	[12] Base : target, Ref : best	0.8	0.9	$10 \cdot D$	
DAS2		0.8	0.9	$10 \cdot D$	
Qin1	[35] Ref : best Ref : best, Diffs : 2	0.9	0.9	50	
Qin2		0.5	0.3	50	
Qin3		0.5	0.3	50	
Qin4		0.5	0.3	50	
Gamperle1	[19] Ref : best, Diffs : 2 Ref : best, Diffs : 2	0.45	0.4	$2 \cdot D$	
Gamperle2		0.6	0.9	$2 \cdot D$	
jDE	[3]	adaptive		100	AdaptF : jDE, AdaptCR : jDE

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`.

3.4 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 [2, 25]. For the proposed modular DE framework, we selected a varied set of 10 strategies for box-constrained problems (such as all problems in

the BBOB test suite) which are fully described and analysed in such articles, and we refer to [40] for implementation details.

3.5 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 [5]. For F and CR , we implement the adaptation mechanisms of SHADE and jDE [3, 38]. 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 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].

4 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 2, 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 [27].

To benchmark this portfolio, we use the BBOB suite from the COCO platform [22]. This suite contains 24 noiseless, single-objective, continuous optimisation problems, each of which can be instantiated with different transformations. These instances aim to provide slight deviations from the original function while preserving the properties of the global landscape. For each DE variant, we collect performance data on all 24 BBOB problems, using IOHexperimenter [15] for data collection. We perform 50 runs per function, spread over 10 instances (5 independent runs per instance). We repeat this for dimensions $D \in \{5, 10, 20\}$, 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 the *Area Over the ECDF Curve* (AOC) as an *anytime performance* measure [21]. Note that we make use of AOC instead of the more common Area Under the Curve (AUC) to keep the interpretation of minimizing the performance metric. To calculate the ECDF and the corresponding AOC, we use a set of 81 targets logarithmically scaled between 10^8 and 10^{-8} . These targets are based on precision (difference between best-so-far $f(x)$ found and the f -value of the global optimum) to allow aggregation across instances. As a last step, we normalise the AOC to lie in $[0, 1]$ by dividing the computed value by the total budget. For *interpreting the AOC values*, we should keep in mind that when all targets are hit immediately, we get a value of 0, which is optimal, while 1 indicates that none of the selected targets have been hit throughout the optimisation run.

Experiment 2 For our second set of experiments, we use the algorithm configuration tool *irace* [30] 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 the f -test version of *irace*, with a first-test value of 5, with the other parameters set to their default values.

We perform 10 independent runs of *irace* on each function from the BBOB suite, for dimensions $D \in \{5, 10, 20\}$, 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 [43]. In addition

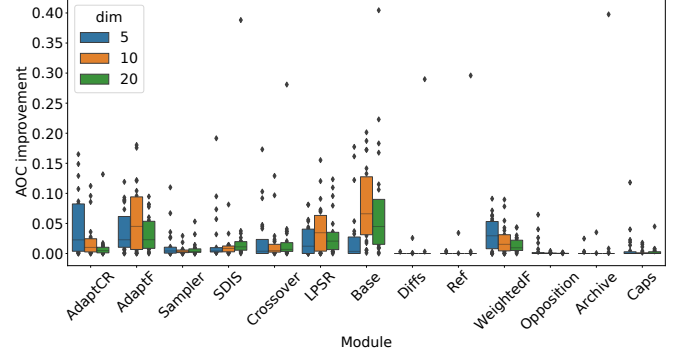


Figure 1: Improvement in AOC over the default setting when selecting the best-performing option for each module.

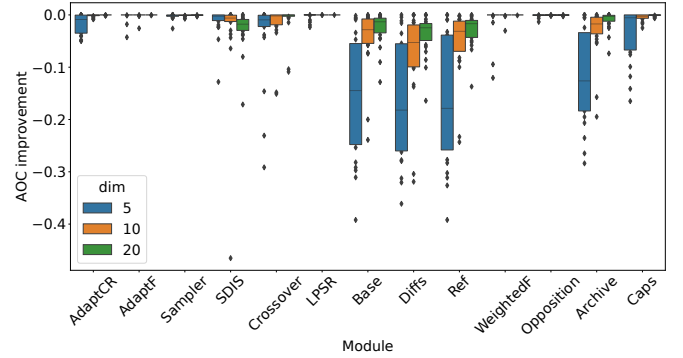


Figure 2: Reduction in AOC over the default setting when selecting the worst-performing option for each module.

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.

Reproducibility: To ensure the reproducibility of our results, the complete set of scripts used for these experiments have been uploaded to a Zenodo repository [40]. This repository also contains the resulting *irace* logs, table with elite configurations, and verification runs in IOHanalyzer format. The notebooks used for analysis and visualisation are also provided. A set of additional figures that could not be included in this paper has been added to Figshare [40].

5 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 1) to the variant with the

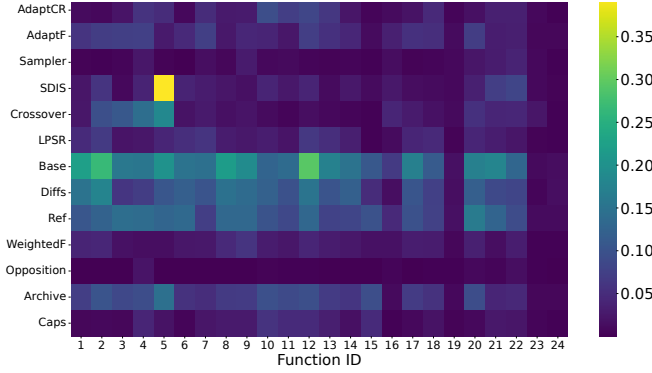


Figure 3: Importance of each module to the AOC on each of the 24 BBOB functions, aggregated over the used dimensions. Importance is calculated as the sum of absolute values from Figures 1 and 2.

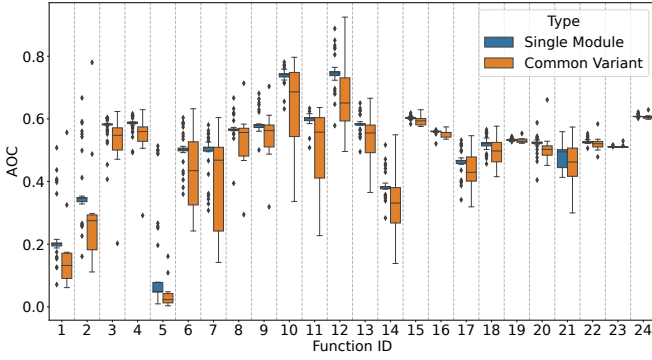


Figure 4: Performance distribution (AOC) of the 30 *single-module* DE variants and the 11 *common* DE variants from Table 2, for the 10-dimensional BBOB problems.

identified best options enabled for each module. The resulting distribution of improvements is shown in Figure 1.

From Figure 1, 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 2. The combination of these two figures gives an overall importance of each module, in the sense that if only 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 3. 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

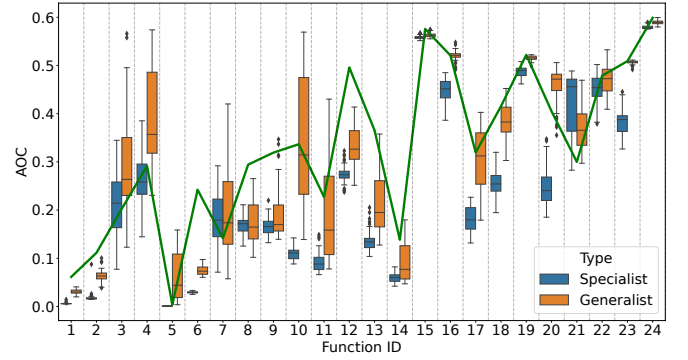


Figure 5: Performance distribution (AOC) of the configurations tuned for an individual function (*specialist*) and the configurations tuned for the full BBOB suite (*generalist*), for the 10-dimensional BBOB problems. The green line shows the best DE version from the union of *single-module* DE and *common* DE variants from Table 2.

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 is rather small. This might indicate that these settings work best when combined with other modules or more specific parameter settings. Also 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 [25]. All other BBOB functions are known not to have optima in the relative vicinity of domain boundaries [28].

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 visualized in Figure 4. 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.

6 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 5. From this figure, we can see that generally, both 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

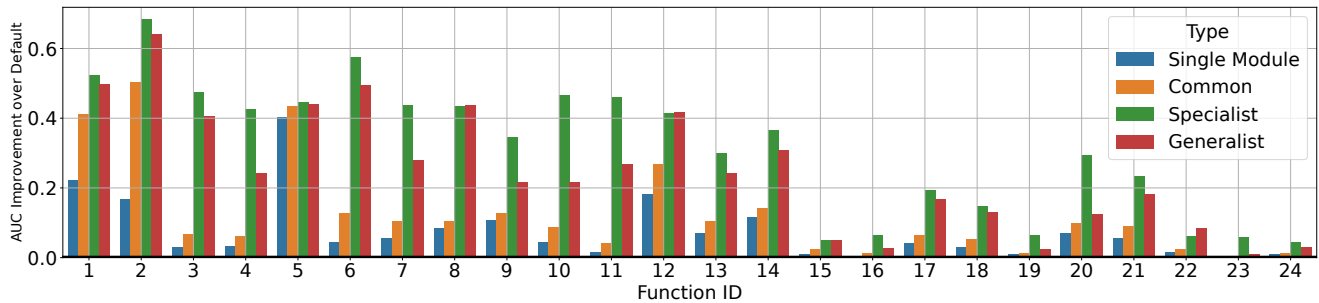


Figure 6: Relative improvement in AOC value between the best configuration of each type and the default setting, in 20D.

in Figure 6. 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 5, which in turn are slightly larger than those achieved on the 5D version of these problems. The figures for these other settings can be found on our Figshare repository [40].

One more important note from Figure 5 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 [42].

This 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, 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 might be able to provide some additional benefits over the current random sampling.

Since the variance in performance between *generalist* configurations is large, it would be worthwhile to investigate the correlation between functions, based on the performance of the elite configurations. This is visualised in Figure 7. From this figure, we see that several groups of problems seem to appear. This grouping might indicate that several of these functions could be removed from the tuning set. Since the algorithm configuration process works on the basis of ranking, testing on multiple instances where the ranking of configurations is almost equivalent might not be very beneficial. Removing some functions from the training set would allow for a better representation of the function space to be used in each individual race, resulting in potentially improved *generalist* configurations.

7 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 dimensions.

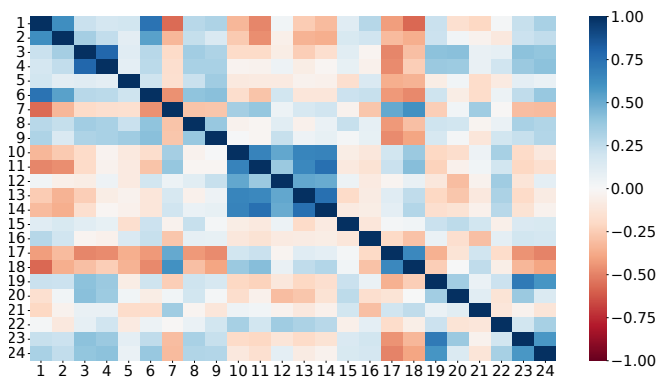


Figure 7: Kendall Tau correlation between the 24 functions of the BBOB suite, measured based on the performance of the set of elites which were tuned on the full BBOB suite in 10D.

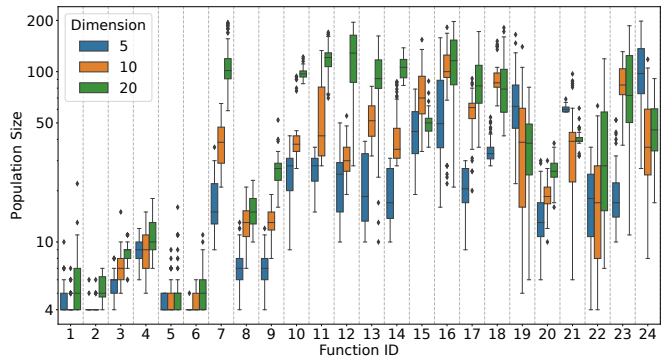


Figure 8: Distribution of population size per dimension in elite configurations (*specialist*).

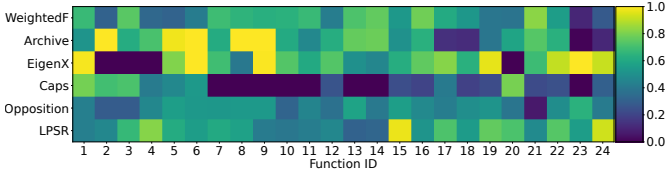


Figure 9: Fraction of elite configurations in which the specified binary module is on (aggregated across dimensions).

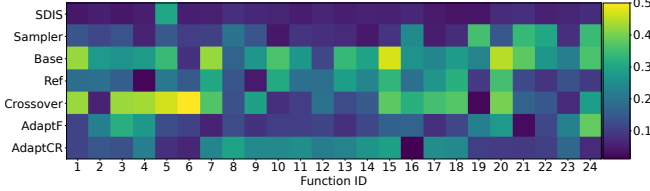


Figure 10: Standard deviation of module frequencies for selected modules.

In Figure 8, we show how the population size changes with respect to functions and dimensionality. In particular, we observe that for the more simple problems, such as the sphere (F1) and ellipsoid (F2), lower population sizes are preferred. In contrast, the highly multimodal problems with medium or low global structure (F15-F24) use a comparatively much higher population size. As would be expected, for most functions, the used population size increases as dimensionality grows. For some functions, this trend is not maintained, which might indicate that the selected limit (200) was too low. This shows that the default population size in Table 1 is indeed not ideal, and higher values should be considered instead.

Next to the parameter values, we can also consider patterns in the modules themselves. For the binary modules, this analysis results in Figure 9, where we show the fraction of times a given module was activated in the elite configuration for each BBOB function. While values close to 0.5 do not provide much direct insight, the more extreme values are interesting to consider. In particular, we notice that the capping mechanism for mutation as used in JSO (Caps) is rarely enabled, indicating that it appears to be detrimental to performance in most problems.

In addition to these binary modules, we can also analyse the other modules. This can be done by considering the standard deviation in the activation frequencies for each module. A low deviation corresponds to no changes from the mean, indicating that all modules are selected equally often, while high values (0.5 as maximum) indicate that one module is selected every time. These deviations, aggregated across dimensions, are visualised in Figure 10. From this figure, we see that the bound correction (SDIS) is distributed relatively uniformly, with the only exception being F5. This matches our previous observations in Figure 3, indicating that irace correctly identified the importance of this module during tuning. Furthermore, the mutation base (Base) seems to be a critical module, with large deviations from the initial uniform distribution.

While Figure 10 gives an overview of the distribution of modules, it does not show which options of these modules are selected more

often. For this, we can instead consider the module activations of a single function and visualise them as a parallel coordinate plot. Figure 11 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 [28]. Because of this, a Gaussian initialisation will significantly outperform any uniform or low-discrepancy initialisation strategy.

In Figure 11 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 $\text{Diff}s = 1$.

We can compare these observations with those in Figure 12, where we show the same visualisation for F2 in 10D. One significant difference between these two settings is the spread of performance. For F19, the differences between the best and worst configurations are greater than 0.1, while for F2 these differences are at most 0.005. This is partly explained by considering that F2 is a unimodal problem and, as such, the performance variability on this function is inherently lower than the one for multimodal F19.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a modular framework for Differential Evolution and illustrate how it can help us gain insight into the differences between the many variations of DE proposed throughout the years. We show the flexibility of this framework by recreating a set of 11 *common* versions of DE. Although these hand-crafted versions seem to outperform the default settings and other variations we create by changing only a single module, we see that the wide range of available options provides a wide potential to improve overall performance.

By utilising irace to tune the modules and parameters of this modular DE, we find consistent improvements over a set of *common* DE variants. When adding more modules, we can repeat this process to gain an understanding of the newly created interaction with existing modules [14]. We illustrated that the results from such a tuning process also help to gain further insight into the benefits of using different versions of DE for different problems.

While the set of *common* DE variants we used here by no means covers the full spectrum of hand-crafted DE versions that have been published, it shows another potential application of the proposed modular framework. If we can recreate an even larger set of previously benchmarked DE variants, we can take a large step towards a fair comparison of their individual contributions to the state-of-the-art. A large-scale benchmarking study using a modular framework would remove many aspects of inconsistency, resulting in a potentially more unbiased comparison. Such a study could

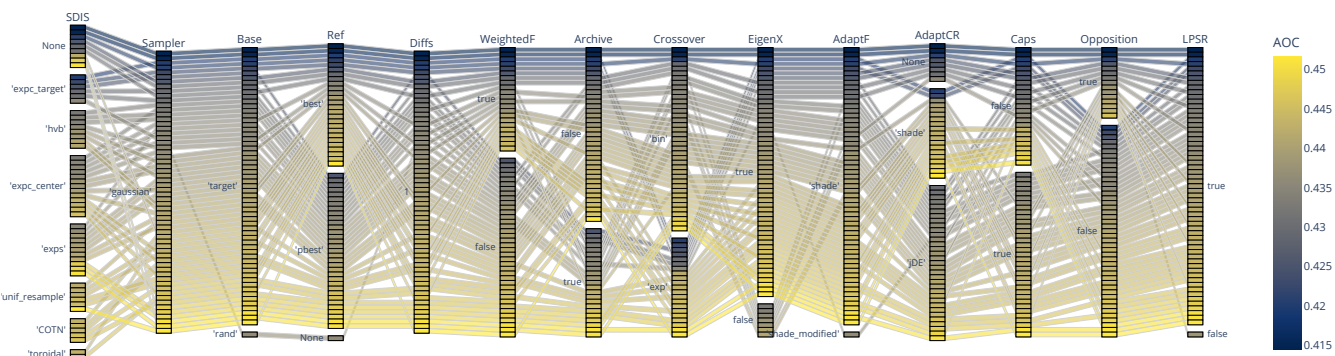


Figure 11: Parallel coordinate plot showing the modules activated in the elite configuration found across 10 runs of irace, on F19 in 5D. Configurations are colored based on normalized AOC.

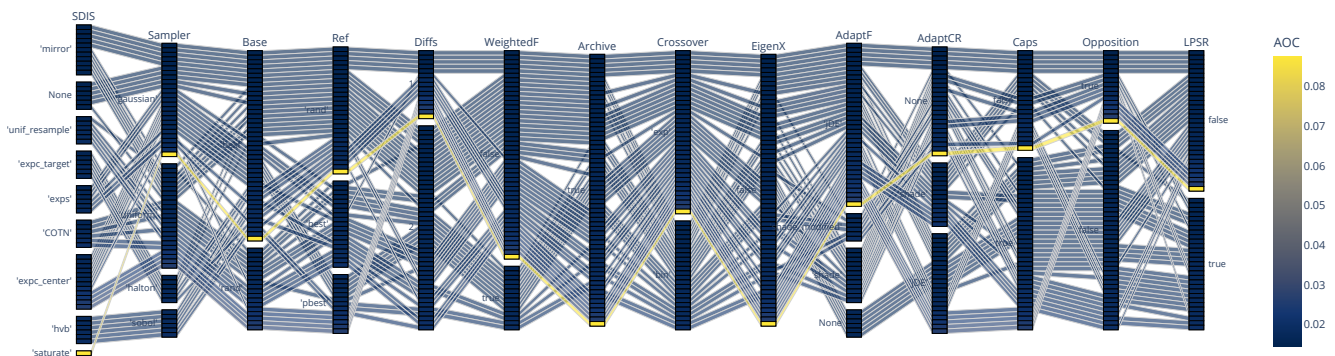


Figure 12: Parallel coordinate plot showing the modules activated in the elite configuration found across 10 runs of irace, on F2 in 10D. Configurations are coloured based on normalized AOC.

show to what extent the published results in DE depend on factors such as the implementation and choice of the benchmark set-up.

The modular DE we propose here is clearly not a complete framework that encapsulates all variations of DE. It might be a lofty goal to create one framework which covers this wide range of options, but striving towards coverage of more algorithm variants seems to be worthwhile. Although more modules increase the complexity of finding good configurations, the modular structure provides opportunities to use techniques such as knowledge graph embedding to create models that can effectively predict the performance of previously unused module combinations [26].

REFERENCES

- [1] Rafał Biedrzycki. 2021. Comparison with State-of-the-Art: Traps and Pitfalls. In *2021 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 863–870.
- [2] Rafał Biedrzycki, Jarosław Arabas, and Dariusz Jagodziński. 2019. Bound constraints handling in Differential Evolution: An experimental study. *Swarm and Evolutionary Computation* 50 (2019), 100453. <https://doi.org/10.1016/j.swevo.2018.10.004>
- [3] Janez Brest, Sao Greiner, Borko Boskovic, Marjan Mernik, and Viljem Zumer. 2006. Self-Adapting Control Parameters in Differential Evolution: A Comparative Study on Numerical Benchmark Problems. *IEEE Transactions on Evolutionary Computation* 10, 6 (2006), 646–657. <https://doi.org/10.1109/TEVC.2006.872133>
- [4] Janez Brest, Mirjam Sepesy Maučec, and Borko Bošković. 2017. Single objective real-parameter optimization: Algorithm jSO. In *2017 IEEE congress on evolutionary computation (CEC)*. IEEE, 1311–1318.
- [5] Janez Brest and Mirjam Sepesy Maučec. 2008. Population size reduction for the differential evolution algorithm. *Applied Intelligence* 29, 3 (2008), 228–247.
- [6] Dimo Brockhoff. 2015. A Bug in the Multiobjective Optimizer IBEA: Salutary Lessons for Code Release and a Performance Re-Assessment. In *Evolutionary Multi-Criterion Optimization - 8th International Conference, EMO 2015, Guimarães, Portugal, March 29 -April 1, 2015. Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9018)*, António Gaspar-Cunha, Carlos Henggeler Antunes, and Carlos A. Coello Coello (Eds.). Springer, 187–201. https://doi.org/10.1007/978-3-319-15934-8_13
- [7] Sébastien Cahon, Nordine Melab, and E-G Talbi. 2004. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of heuristics* 10, 3 (2004), 357–380.
- [8] Christian L Camacho-Villalón, Marco Dorigo, and Thomas Stützle. 2021. PSO-X: A component-based framework for the automatic design of particle swarm optimization algorithms. *IEEE Transactions on Evolutionary Computation* 26, 3 (2021), 402–416.
- [9] Felipe Campelo and Moisés Botelho. 2016. Experimental investigation of recombination operators for differential evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. 221–228.
- [10] Guogang Cao, Cong Cao, Qing Zhang, and Wenju Li. 2018. Differential Evolution Improved with Intelligent Mutation Operator Based on Proximity and Ranking. In *2018 11th International Symposium on Computational Intelligence and Design (ISCID)*, Vol. 02. 196–201. <https://doi.org/10.1109/ISCID.2018.10146>
- [11] Fabio Caraffini, Anna V. Kononova, and David Corne. 2019. Infeasibility and structural bias in differential evolution. *Inf. Sci.* 496 (2019), 161–179. <https://doi.org/10.1016/j.ins.2019.05.019>
- [12] Swagatam Das, Ajith Abraham, Uday K Chakraborty, and Amit Konar. 2009. Differential evolution using a neighborhood-based mutation operator. *IEEE transactions on evolutionary computation* 13, 3 (2009), 526–553.
- [13] S. Das, Sankha Subhra Mullick, and P.N. Suganthan. 2016. Recent advances in differential evolution – An updated survey. *Swarm and Evolutionary Computation* 27 (2016), 1 – 30. <https://doi.org/10.1016/j.swevo.2016.01.004>
- [14] Jacob de Nobel, Diederick Vermetten, Hao Wang, Carola Doerr, and Thomas Bäck. 2021. Tuning as a means of assessing the benefits of new ideas in interplay with existing algorithmic modules. In *Proceedings of the Genetic and Evolutionary*

- Computation Conference Companion*. 1375–1384.
- [15] Jacob de Nobel, Furong Ye, Diederick Vermetten, Hao Wang, Carola Doerr, and Thomas Bäck. 2021. IOExperimenter: Benchmarking Platform for Iterative Optimization Heuristics. *arXiv preprint arXiv:2111.04077* (2021).
 - [16] James Demmel and Krešimir Veselić. 1992. Jacobi's Method is More Accurate than QR. *SIAM J. Matrix Anal. Appl.* 13, 4 (1992), 1204–1245.
 - [17] Johann Dreo, Arnaud Liefvooghe, Sébastien Verel, Marc Schoenauer, Juan J Merelo, Alexandre Quemy, Benjamin Bouvier, and Jan Gmys. 2021. Paradiseo: from a modular framework for evolutionary computation to the automated design of metaheuristics: 22 years of Paradiseo. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 1522–1530.
 - [18] Michael G. Epitropakis, Dimitris K. Tasoulis, Nicos G. Pavlidis, Vassilis P. Plagianakos, and Michael N. Vrahatis. 2011. Enhancing Differential Evolution Utilizing Proximity-Based Mutation Operators. *IEEE Transactions on Evolutionary Computation* 15, 1 (2011), 99–119. <https://doi.org/10.1109/TEVC.2010.2083670>
 - [19] Roger Gämperle, Sibylle D Müller, and Petros Koumoutsakos. 2002. A parameter study for differential evolution. *Advances in intelligent systems, fuzzy systems, evolutionary computation* 10, 10 (2002), 293–298.
 - [20] S. M. Guo and C. C. Yang. 2015. Enhancing Differential Evolution Utilizing Eigenvector-Based Crossover Operator. *IEEE Transactions on Evolutionary Computation* 19, 1 (Feb 2015), 31–49.
 - [21] Nikolaus Hansen, Anne Auger, Dimo Brockhoff, and Tea Tušar. 2022. Anytime Performance Assessment in Blackbox Optimization Benchmarking. *IEEE Transactions on Evolutionary Computation* 26, 6 (2022), 1293–1305.
 - [22] Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tušar, and Dimo Brockhoff. 2020. COCO: A platform for comparing continuous optimizers in a black-box setting. *Optimization Methods and Software* (2020), 1–31.
 - [23] Sk. Minhazul Islam, Swagatam Das, Saurav Ghosh, Subhrajit Roy, and Ponnuthurai Nagarathnam Suganthan. 2012. An Adaptive Differential Evolution Algorithm With Novel Mutation and Crossover Strategies for Global Numerical Optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 42, 2 (2012), 482–500. <https://doi.org/10.1109/TSMCB.2011.2167966>
 - [24] Anna V Kononova, Fabio Caraffini, and Thomas Bäck. 2021. Differential evolution outside the box. *Information Sciences* 581 (2021), 587–604.
 - [25] Anna V Kononova, Diederick Vermetten, Fabio Caraffini, Madalina-A Mitran, and Daniela Zaharie. 2022. The importance of being constrained: dealing with infeasible solutions in Differential Evolution and beyond. *arXiv preprint arXiv:2203.03512* (2022).
 - [26] Ana Kostovska, Diederick Vermetten, Sašo Džeroski, Panče Panov, Tome Eftimov, and Carola Doerr. 2023. Using Knowledge Graphs for Performance Prediction of Modular Optimization Algorithms. *arXiv preprint arXiv:2301.09876* (2023).
 - [27] J. Liu and J. Lampinen. 2005. A Fuzzy Adaptive Differential Evolution Algorithm. *Soft Computing* 9, 6 (01 Jun 2005), 448–462. <https://doi.org/10.1007/s00500-004-0363-x>
 - [28] Fu Xing Long, Diederick Vermetten, Bas van Stein, and Anna V Kononova. 2022. BBOB Instance Analysis: Landscape Properties and Algorithm Performance across Problem Instances. *arXiv preprint arXiv:2211.16318* (2022).
 - [29] Manuel López-Ibañez, Juergen Branke, and Luis Paquete. 2021. Reproducibility in evolutionary computation. *ACM Transactions on Evolutionary Learning and Optimization* 1, 4 (2021), 1–21.
 - [30] Manuel López-Ibañez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. 2016. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives* 3 (2016), 43 – 58. <https://doi.org/10.1016/j.orp.2016.09.002>
 - [31] Manuel Lopez-Ibanez and Thomas Stutzle. 2012. The automatic design of multi-objective ant colony optimization algorithms. *IEEE Transactions on Evolutionary Computation* 16, 6 (2012), 861–875.
 - [32] Martin Lukasiwycz, Michael Glaß, Felix Reimann, and Jürgen Teich. 2011. Opt4J: a modular framework for meta-heuristic optimization. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. 1723–1730.
 - [33] K. Price and R. Storn. 1997. Differential evolution: A simple evolution strategy for fast optimization. *Dr. Dobb's J. Software Tools* 22, 4 (1997), 18–24.
 - [34] Kenneth V. Price, Rainer Storn, and Jouni Lampinen. 2005. *Differential Evolution: A Practical Approach to Global Optimization*. Springer. <https://doi.org/10.1007/3-540-31306-0>
 - [35] A Kai Qin, Vicky Ling Huang, and Ponnuthurai N Suganthan. 2008. Differential evolution algorithm with strategy adaptation for global numerical optimization. *IEEE transactions on Evolutionary Computation* 13, 2 (2008), 398–417.
 - [36] S. Rahnamayan, H. R. Tizhoosh, and M. M. Salama. 2008. Opposition-Based Differential Evolution. *IEEE Transactions on Evolutionary Computation* 12, 1 (2008), 64–79.
 - [37] R. Storn and K. Price. 1995. *Differential Evolution - a Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces*. Technical Report TR-95-012. ICSI.
 - [38] Ryoji Tanabe and Alex Fukunaga. 2013. Success-history based parameter adaptation for Differential Evolution. In *2013 IEEE Congress on Evolutionary Computation*. 71–78. <https://doi.org/10.1109/CEC.2013.6557555>
 - [39] Sander van Rijn, Hao Wang, Matthijs van Leeuwen, and Thomas Bäck. 2016. Evolving the structure of Evolution Strategies. In *SSCI*. 1–8. <https://doi.org/10.1109/SSCI.2016.7850138>
 - [40] Diederick Vermetten, Fabio Caraffini, Anna V. Kononova, and Thomas Bäck. 2023. Reproducibility files and additional figures. (Feb. 2023). Code and data repository: <https://doi.org/10.5281/zenodo.7624677> Figure repository: <https://figshare.com/s/2885a458ec5270d9c0ce>.
 - [41] Diederick Vermetten, Bas van Stein, Anna V Kononova, and Fabio Caraffini. 2022. Analysis of structural bias in differential evolution configurations. In *Differential Evolution: From Theory to Practice*. Springer, 1–22.
 - [42] Diederick Vermetten, Hao Wang, Manuel López-Ibañez, Carola Doerr, and Thomas Bäck. 2022. Analyzing the impact of undersampling on the benchmarking and configuration of evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 867–875.
 - [43] Furong Ye, Carola Doerr, Hao Wang, and Thomas Bäck. 2022. Automated configuration of genetic algorithms by tuning for anytime performance. *IEEE Transactions on Evolutionary Computation* 26, 6 (2022), 1526–1538.
 - [44] Jingqiao Zhang and Arthur C Sanderson. 2009. JADE: adaptive differential evolution with optional external archive. 13, 5 (2009), 945–958.