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 5

Dynamic Algorithm Configuration and Selection

Exploiting complementarity between algorithms or configurations of algorithms can lead to significant gains in performance, as illustrated in the previous chapter. It is also known that, when solving an optimization problem, different stages of the process require different search behavior. For example, while exploration is needed in the initial phases, the algorithm needs to eventually converge to a solution (exploitation). State-of-the-art optimization algorithms therefore often incorporate mechanisms to adjust their search behavior *while optimizing*, by taking into account the information obtained during the run. These techniques are studied under many different umbrellas, such as *parameter control* [70], *meta-heuristics* [22], adaptive operator selection [162], or *hyper-heuristics* [31]. The probably best-known and most widely used techniques for achieving a dynamic search behavior are the *one-fifth success rule* [201, 56, 211] and the *covariance adaptation technique* that the family of CMA-ES algorithms [97, 98] is built upon. While each of these control mechanisms tackles the problem of balancing performance in different phases of the search in its own way, they are mostly working with a specific algorithm, aiming to tune its performance by changing internal parameters or algorithm modules. This inherently limits the potential of these methods, since different algorithms can have widely varying performances during different phases of the optimization process. By switching between these algorithms during the search, these differences could potentially be exploited to get even better performance. We refer to the problem of choosing which algorithms to switch between, and under which

circumstances, as the *Dynamic Algorithm Selection* (dynAS) problem.

Solving the dynAS problem would be an important milestone towards tackling the more general *dynamic Algorithm Configuration (dynAC)* problem, which also addresses the problem of selecting (and possibly adjusting) suitable algorithm configurations. Specifically, dynAS is limited to switching between algorithms from a discrete portfolio of pre-configured heuristics, whereas for dynAC, the algorithms come with (possibly several) parameters whose settings can have a significant influence on the performance.

While dynamically changing between algorithms or algorithm configurations can be tackled in a variety of ways. For example, in the context of machine learning, there are a variety of works which utilize principles from meta-learning to allow their algorithms to handle data streams which change over time [206] or where choices have to be made at multiple time points [231]. These problems can similarly be tackled using portfolios of algorithms, with bandit algorithms running during the optimization determining how to allocate resources between them [81].

In our work, we focus on a reinforcement-learning-based formulation of the DynAS problem [2]. In particular, we follow the principles outlined in [15] to represent the switching between algorithms as a policy function. This results in the following problem definition:

Definition 5.1 (Dynamic Algorithm Selection (dynAS))**.** Given an algorithm portfolio $\mathcal{A}$, a function $f \in \mathcal{F}$ and a state description $s_t \in \mathbb{S}$ at time step t of an algorithm run. We want to find a policy $\pi : \mathbb{S} \to \mathcal{A}$ which minimizes a performance measure $\text{PERF}(A_\pi, f)$.

Note that this definition can be extended to dynamic algorithm configuration by changing the policy to be $\pi : \mathbb{S} \to (\mathcal{A} \times \Theta_A)$, where Θ_A is the configuration space of algorithm A.

This chapter is based on the following publications: [245, 243, 110, 135, 248]

5.1 Complementarity in Anytime Performance

Before tackling the dynAS problem, we first aim to show the potential of this approach for numerical optimization. We do this by taking a data-driven approach, where we identify the complementarity between algorithms from a large portfolio purely based on their performance profiles, for a simplified version of dynAS where we can switch between algorithms only once during the optimization run. As in previous chapters,

we stick to the BBOB suite from the COCO environment [95], and in particular we make use of its rich collection of algorithm performance data [4].

Our considerations are purely based on a theoretical investigation of the potential, which might be too optimistic for the single-switch dynAS case – most importantly, because of the problem of *warm-starting* the algorithms: since the heuristics are adaptive themselves, their states need to be initialized appropriately at the switch. This may be a difficult problem when changing between algorithms of very different structure. We do not consider, on the other hand, the possibility to switch more than once, so that our bounds may be too pessimistic for the full dynAS setting, in which an arbitrary number of switches is allowed.

Given the above limitations, we therefore also provide a critical assessment of our approach, and highlight ideas for addressing the main challenges in dynAS.

5.1.1 Analysis of Available data

Since the set of available algorithms from the BBOB competitions is quite large, several issues in terms of data consistency arise. When processing the algorithms, we found that a small subset has issues such as incomplete files or missing data. We decided to ignore these algorithms and work only with the ones which were made available within the IOHanalyzer tool [62]. This leaves us with a set of 182 out of 226 possible algorithms to do our analysis.

There are some caveats to this data, mostly related to the lack of a consistent policy for submission to the competitions over the years. For example, the 2009 competition required the submission of 3 runs on 5 instances each, while the 2010 version changed this to 1 run on 15 instances. In theory, the instances should have very little impact on the performance of the algorithms, as they are selected in such a way as to preserve the characteristics of the functions. However, in practice there has been some debate about the impact of instances on algorithm performance, claiming that the landscapes of different instances of the same function can look significantly different to an algorithm [173, 170, 127] (see Chapter 3.2 for more discussion on this topic). In the following, we ignore this discussion and assume that performance is not significantly impacted by the instances.

Another issue with the dataset is the usage of widely inconsistent budgets for the different algorithms. These can be as low as $50D$ and as large as $10^7 D$. However, since we use a fixed-target perspective to study the performance of the algorithms, these differences are not very impactful.

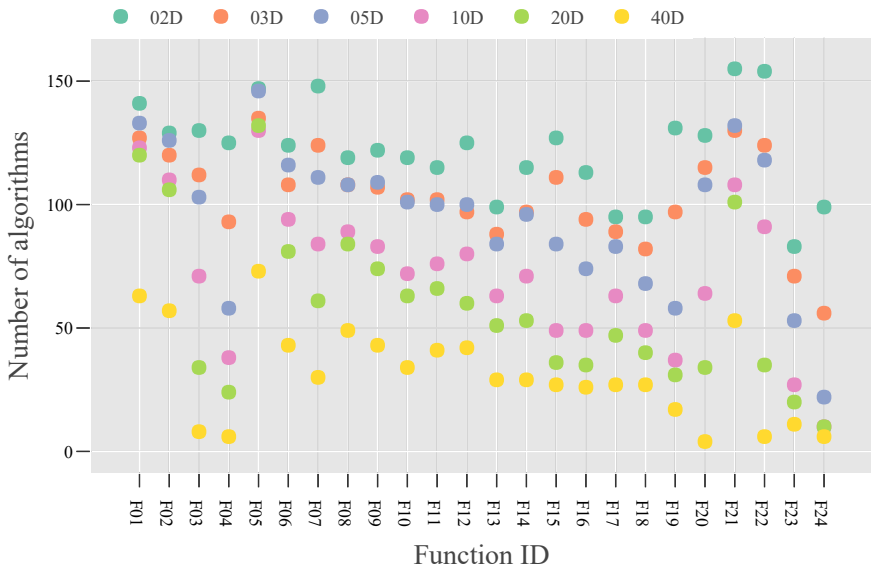


Figure 5.1: Number of algorithms with at least 15 independent runs and at least one of them reaching the target $\phi = 10^{-8}$.

Since the BBOB competitions see an optimizer as having 'solved' an optimization problem when reaching a target precision of 10^{-8}, many of the algorithms will stop their runs after reaching this point to avoid unnecessary computation. Because of this, we will use the same target value in our computations. However, for some of the more difficult functions, this target can be challenging to reach within their budget. To avoid the problem of dealing with algorithms without any finished runs, we only consider an algorithm in our analysis when it has at least 15 runs on the function, of which at least one managed to reach the target 10^{-8}. Figure 5.1 plots the number of algorithms per each function/dimensionality pair that satisfy all the requirements mentioned above. We observe large discrepancies between functions and dimensionalities, with the number of admissible algorithms ranging from 4 to 155, and note that there are no algorithms which are admissible on all functions in all dimensionalities.

5.1.2 DynAS for BBOB-Functions

In this section, we will restrict the dynAS problem on BBOB-functions to using policies which switch algorithms based on the target precisions hit. To get an indication for the amount of improvement which can be gained by dynAC over static algorithm configuration, we use the BBOB-data to theoretically simulate a simple policy which

only implements a single switch of algorithm. We can define this as follows:

Definition 5.2 (Single-Switch dynAS)**.** Let $f^{(d)}$ be a d-dimensional BBOB-function and $\mathcal{A}$ the corresponding portfolio of admissible algorithms. A single-switch policy is defined as the triple $(A_1, A_2, \tau) \in \mathcal{A} \times \mathcal{A} \times \Phi$, where $\Phi = \left\{10^{2-0.2i}) | i \in \{0, \ldots, 50\}\right\}$ is the set of admissible switchpoints. This corresponds to the policy which starts the optimization procedure with algorithm A_1, and run this until target τ is reached, after which the algorithm is changed to A_2.

The performance of this single switch method can then be calculated as follows:

$$\begin{aligned} T(f^{(d)}, A_1, A_2, \tau, \phi) = &\operatorname{ERT}(A_1, f^{(d)}, \tau) \\ &+ \operatorname{ERT}(A_2, f^{(d)}, \phi) - \operatorname{ERT}(A_2, f^{(d)}, \tau) \end{aligned}$$

Here, ϕ is the final target precision we want to reach. For the BBOB-functions, we set $\phi = 10^{-8}$, as noted in Section 5.1.1.

Generally, to assess the performance of an *algorithm selection* method, its performance can be compared to the *Single Best Solver (SBS)*, which can be defined as follows:

Definition 5.3 (Single Best Solver)**.** For each dimensionality $d \in \mathcal{D}$, we have:

$$\operatorname{SBS}_{\text{static}}(\mathcal{F}^{(d)}) = \arg\min_{A \in \mathcal{A}} \sum_{f \in \mathcal{F}} \operatorname{PERF}(A, f^{(d)}, \phi)$$

Often, ERT is used as the performance function, but this value can differ widely between functions, leading to a biased weighting. To avoid this, we can instead use the ranking of ERT per function, to give equal importance to every function. Note that we have final target precision $\phi = 10^{-8}$.

While this SBS has a good average performance, it can easily be beaten by a decent *algorithm selection* technique. As such, a better baseline for performance is needed. This is the theoretically best algorithm selection method, which is called the Virtual Best Solver. This can defined as follows:

Definition 5.4 (Static Virtual Best Solver ($\text{VBS}_{\text{static}}$))**.** For each function $f \in \mathcal{F}$ and dimensionality $d \in \mathcal{D}$, we have:

$$\operatorname{VBS}_{\text{static}}(f^{(d)}) = \arg\min_{A \in \mathcal{A}} \operatorname{PERF}(A, f^{(d)})$$

For the BBOB functions, we use $\operatorname{PERF}(A, f^{(d)}) = \operatorname{ERT}(A, f^{(d)}, \phi)$ with $\phi = 10^{-8}$.

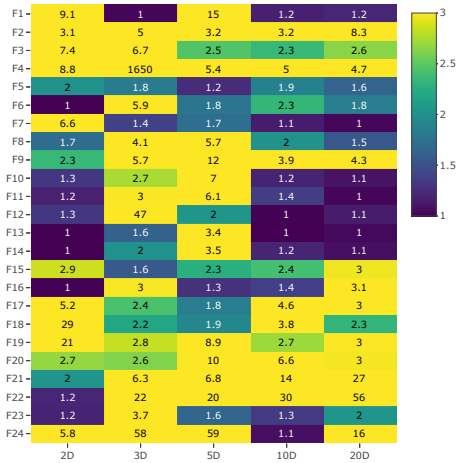


Figure 5.2: Relative ERT of the SBS over the $\text{VBS}_{\text{static}}$. The selected SBS are: Nelder-Doerr (2D), HCMA(3, 10 and 20D) and BIPOP-aCMA-STEP (5D). dimensionality 40 was removed because no algorithm hit the final target on all functions in this dimensionality.

Note that the $\text{VBS}_{\text{static}}$ will always perform at least as good as the SBS, and theoretically gives an upper bound for the performance of any real implementation of algorithm selection techniques. Thus, the difference between SBS and $\text{VBS}_{\text{static}}$ gives an indication of the maximal possible performance gained by algorithm selection. For the BBOB-data, the relative ERT between these two methods is visualized in Figure 5.2. From this, we see that the differences can be extremely large, highlighting the importance of algorithm selection.

Similar to the way we defined $\text{VBS}_{\text{static}}$, we can define a Dynamic Virtual Best Solver, VBS_{dyn}, as follows:

Definition 5.5 (Dynamic Virtual Best Solver)**.** For each BBOB-function $f \in \mathcal{F}$ and dimensionality $d \in \mathcal{D}$, we have:

$$\text{VBS}_{\text{dyn}}(f^{(d)}) = \underset{(A_1, A_2, \tau) \in (\mathcal{A} \times \mathcal{A} \times \Phi)}{\arg\min} T(f^{(d)}, A_1, A_2, \tau, \phi)$$

5.1.3 Results

Since the number of algorithms considered in this paper is relatively large, many of the results are only shown for a subset of functions, dimensionalities or algorithms.

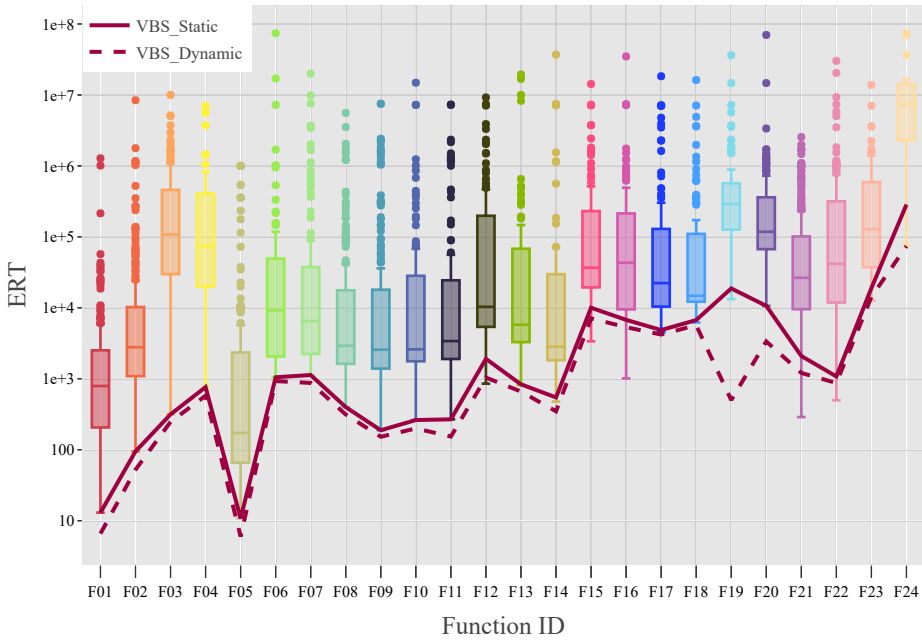


Figure 5.3: Distribution of ERTs among all algorithms for all 24 BBOB-functions in dimensionality 5. Please recall from Figure 5.1 that the number of data points varies between functions. Also shown are the ERTs of the $\text{VBS}_{\text{static}}$ and VBS_{dyn}.

Overall Gain of Single-Switch DynAS

Before investigating the possible improvements to be gained by dynamic algorithm selection, we investigate the performance of the static algorithms from the BBOB-dataset. To achieve this, we look at the distribution of ERTs among the BBOB-functions. For dimensionality 5, this is visualized in Figure 5.3.[1] This figure shows the large differences in performance, both between the algorithms as well as between the different functions. We marked the performance of the $\text{VBS}_{\text{static}}$ and VBS_{dyn}, and see that their differences also vary largely between functions.

To zoom in on the differences between the $\text{VBS}_{\text{static}}$ and VBS_{dyn} we see in Figure 5.3, we can compute for each function, dimensionality and corresponding algorithm portfolio the relative ERT of a the Single-Switch VBS_{dyn} over $\text{VBS}_{\text{static}}$. Specifically, this is calculated as $\frac{\text{ERT}(\text{VBS}_{\text{dynamic}}(f^{(d)}))}{\text{ERT}(\text{VBS}_{\text{static}}(f^{(d)}))}$. This value is shown for each (function, dimensionality)-pair in Figure 5.4. From this figure, we can see that for most func-

[1] Note that for function F05, the linear slope, most algorithms simply move outside the search-space to find an optimal solution, which is accepted by the BBOB-competitions, but leads to a disadvantage to those algorithms which respect the bounds.

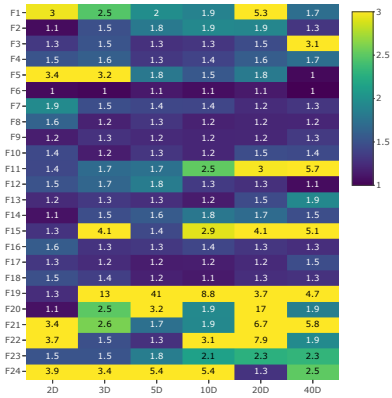


Figure 5.4: Heatmap of the ratio of ERTs between the Virtual Best Static Solver and the Virtual Best Dynamic Solver, for each (function, dimensionality)-pair.

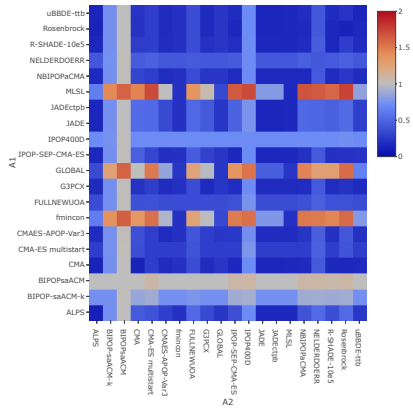


Figure 5.5: Relative ERT of configuration switches relative to $\text{VBS}_{\text{static}}$, for 10-dimensional function 21. The X- and Y-axes indicate algorithms selected as A_2 and A_1 respectively. Larger values (red) indicate better algorithm combinations.

tions, the improvements when using a single configuration change are quite large. Especially for the functions which are traditionally considered more difficult for a black-box optimization algorithm to solve, the possible improvement is massive. In terms of the median over all (function, dimensionality)-pairs, the VBS_{dyn} is 1.49 faster than the $\text{VBS}_{\text{static}}$.

Selected Algorithm Combinations

Since the VBS_{dyn} shows a lot of potential improvement over the classical $\text{VBS}_{\text{static}}$, it makes sense to study its behaviour in more detail. To achieve this, we can zoom in on a single (function, dimensionality)-pair and study the behaviour of the VBS_{dyn} and switching algorithm configurations in general. In Figure 5.5, we show the ERT of the best possible switch between any combination of algorithms in our portfolio $\mathcal{A}$, on function 21 in dimensionality 10. This figure shows some clear patterns in the horizontal and vertical lines. A horizontal line, such as the one for the MLSL-algorithm [147], indicates that an algorithm adds to the performance of most algorithms by being the A_1-algorithm.

This can be interpreted as having a good exploratory search behaviour, but poor exploitation. There are also vertical lines present, which indicate the algorithms which

perform well as A_2-algorithms. These are less pronounced than the horizontal lines, which might indicate that the choice of A_2 algorithms has less impact on the performance than the choice of A_1.

Small Portfolio: Case Study

Since the algorithm space we consider is quite large, it can be challenging to gain insights into the individual algorithms. To show that dynamic algorithm selection is also applicable to smaller portfolio's, we limit ourselves to 5 algorithms. These are representative of some widely used algorithm families: Nelder-Doerr [61], DE-Auto [252], Bipop-aCMA-Step [155], HMLSL [183], and PSO-BFGS [142]. With this reduced algorithm portfolio, we can study the improvements over their respective $\text{VBS}_{\text{static}}$ in more detail, and find interesting algorithms combinations to explore further.

To illustrate the configuration switches which can be considered in this algorithm portfolio, we can zoom in on function 12 in dimensionality 3 and look at the fixed-target curve showing ERT. This is done in Figure 5.6, where we also indicate the best switching points between algorithms. This figure highlights the different behaviors of the algorithms in the portfolio, and thus indicates where switching algorithms would be beneficial. The best possible switch in this function would occur from PSO-BFGS to Nelder-Doerr, at target $10^{-6.4}$, leading to a relative speedup of 1.76 over $\text{VBS}_{\text{static}}$.

To decide which algorithms to use in an algorithm portfolio such as the one used here, two main ways of selecting the algorithms are possible. The first is to use some knowledge about the algorithms to determine which are important. This is useful for initial exploration, but might lead to useful algorithms being ignored. Instead, one can use performance information, such as the I_1 and I_2-values, to provide some initial representation of the usefulness of algorithms to the portfolio. This approach is much more generic, however the choice of measures can be challenging. For example, the I_1 and I_2 measures are hard to extend to more general k-switch dynAS methods. Instead, an extension of marginal contributions [262] and related concepts such as measures building on Shapley values (like those suggested in [79]) would capture algorithm contribution to a portfolio in a much more robust sense, and thus be useful additions to the dynAS setting.

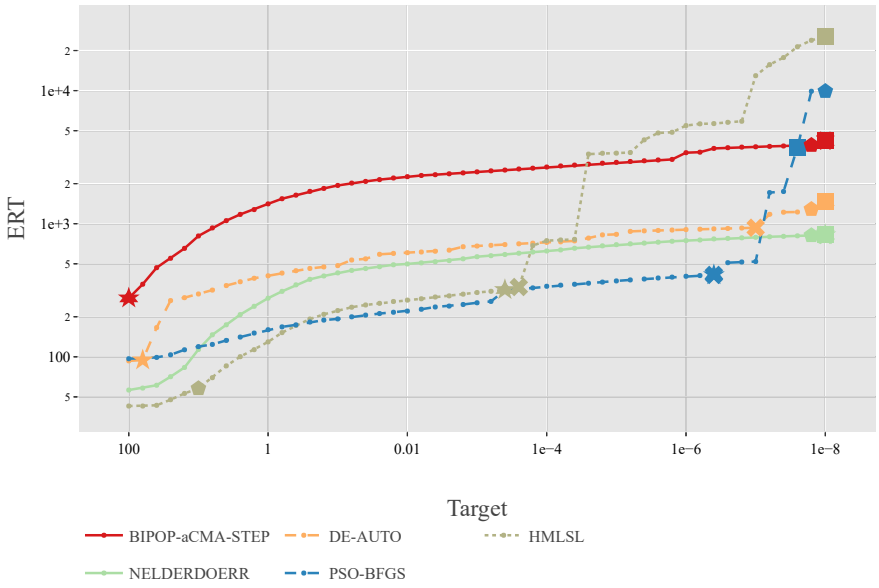


Figure 5.6: ERT-curves for a selected algorithm portfolio of size 5 on F12 in 3D. Markers indicate optimal switch points between algorithms. Their color and symbol indicate the starting and finishing algorithms respectively. (star = Nelder-Doerr, triangle = DE-AUTO, cross = BIPOP-aCMA-STEP, square = HMLSL and pentagon = PSO-BFGS).

5.2 Switching Between Algorithm Variants

To achieve dynAS, we need to tackle the problem of *warmstarting*: initializing the internal state of the secondary algorithm after the first has been terminated. Depending on the used algorithms, this can be an extremely challenging task. To limit the effort needed to warmstart an algorithm, we can ensure all algorithms share the same internal state, as is the case when we limit ourselves to a single modular algorithm framework. In this section, we work within the modCMA framework to implement the single-switch version of dynAS, where we exploit the complementarity between the many module combinations, as was illustrated in Chapter 4. In particular, we aim to switch between different configurations of modCMA (without the local restart module).

5.2.1 Selecting Adaptive Configurations

To determine which switches we should make, we start by gathering benchmark data from all 24 BBOB functions (5-dimensional versions only). We then gather the AHTs for targets $\Phi = \{10^{2-(0.2 \cdot i)} \mid i \in \{0 \dots 50\}\}$. Based on these AHT values, the adaptive configurations suggested in [232] are chosen as follows:

- For each configuration c, each of the 24 BBOB functions f, and each of the 51 target values ϕ, we calculate the AHT over all 25 runs (5 runs for each of the first five instances).
- From this data, we determine the best target value ϕ_{min} for which there exists at least one configuration whose 25 runs all reached this target.
- For every target value $\phi \in \Phi$ satisfying $\phi > \phi_{\text{min}}$ we calculate the best configuration before this target, i.e., we select the configuration c for which $AHT(c, \phi)$ is minimized. We denote this configuration C_1. We then compute the best configuration c from this target until ϕ_{min}, which we denote as C_2, i.e., C_2 is the configuration for which $AHT(f, c, \phi_{\text{min}}) - AHT(f, c, \phi_{\text{min}})$ is minimized. In [232], the theoretical performance (TH for 'theoretical hitting time') is then calculated as $TH(f, C_1, C_2, \phi) = AHT(f, C_1, \phi) - AHT(f, C_2, \phi) + AHT(f, C_2, \phi_{\text{min}})$.
- From this data we compute the target value τ for which the overall performance $TH(f, C_1, C_2, \phi)$ is minimized. This gives us the adaptive configuration (C_1, C_2, τ). We refer to τ as the 'switchpoint' of the adaptive configuration.

5.2.2 Two-Stage Configuration Selection

We introduce a procedure to make the selection process more robust to noise in the performance data. This is based on the finding that the static configurations are not quite stable enough to be used as a baseline. The first step in this process consists of selecting some static configurations for which we should gather more data. The configurations we will consider are made up of two parts. The first part consists of the 50 best-performing static configurations.[2] We then extend this set by looking at the configurations which have been selected to be a part of the 50 theoretically best adaptive configurations. Since this might not be a diverse set of configurations, as one

[2]The best static configurations are determined by their AHT at the final reached target. If fewer than 50 configurations reach this target for a function, we extend these configurations by the ones that have the lowest AHT for the previous target. We repeat this process until we have selected 50 configurations.

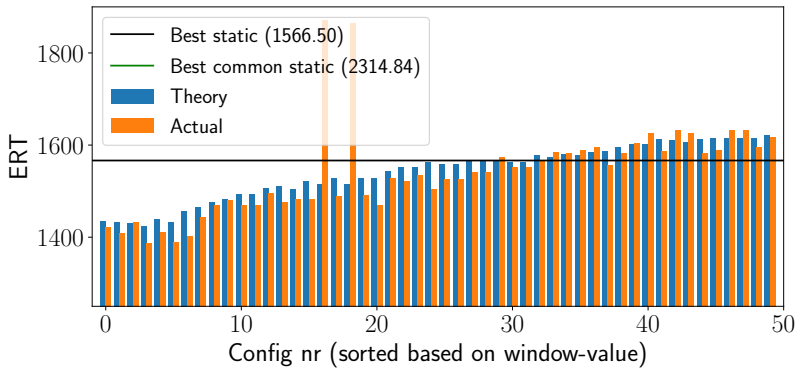


Figure 5.7: F10: ERT of adaptive configurations compared to the best static and "common" static configurations.

configuration might be chosen as C_1 50 times, we decide to limit the number of times a certain configuration can be selected as C_1 and as C_2 to three times each ('limited selection method'). This should give us a more diverse set of configurations which might contribute to good adaptive configurations. We then rerun these configurations using 50 runs on each of the 5 instances, for a total of 250 runs each.

5.2.3 Performance Comparison

The results of the two-stage method are shown in more detail in Figure 5.7 for F10. From this figure, we can see that the fit between theory and practice is quite good, and many of the adaptive configurations manage to outperform the best static configuration by around 10%. Some outliers are present, but the general trend is positive. In this figure, we also note the ERT of the best "common" CMA-ES variant as defined in [234].

An overview of the performance comparison between these groups of configurations can be seen in Figure 5.8. One important point to note is the fact that the best "common" static configuration can outperform the general best static. This is caused by the fact that these common configurations can have (B)IPOP enabled, which is not the case for the best static. In these cases, we assume that this (B)IPOP module is important to finding the optimum, and an adaptive configuration without this module will not be able to perform very well.

Next, we consider the functions for which the best static ERT is lower than that of the common variants. For these functions, we manage to improve upon this best static configuration when using an adaptive configuration. More specifically, we can

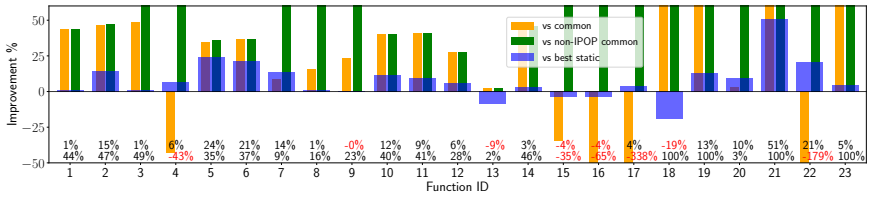


Figure 5.8: Comparison of the best achieved switching ERT relative to the ERTs of the common statics (with and without IPOP; 5×5 runs) and the best non-IPOP static on 5×50 runs. Improvements are cut off at 60% and -50%, respectively. The precise values of the improvements are shown above the x-axis for the improvement relative to the best static (top) and relative to the best common (bottom) configuration.

see that when the best static configuration from the entire configuration space does not have (B)IPOP enabled, we can reliably achieve an improvement when using adaptive configurations.

We also note that when the best static configuration with (B)IPOP significantly outperforms the best rerun configuration, we do not manage to get the same improvements. If we consider the best static configurations to include those with (B)IPOP and compare the performance of the adaptive configurations to those, no improvement is made at all.

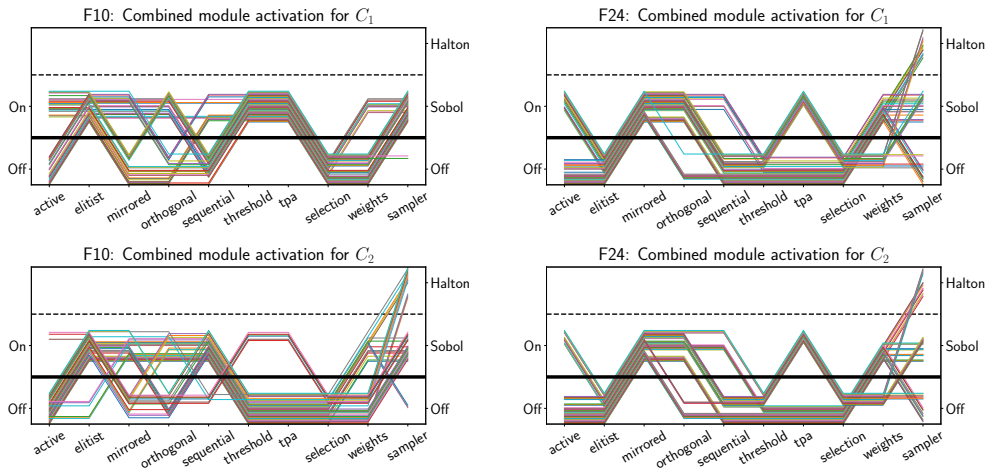
In total, we find performance gains on 18 out of 24 functions of the BBOB benchmark, with stable advantages of up to 23%.

5.2.4 Module Activation Plots

We will now study two functions in more detail. The functions we will analyze are F10, for which we see a decent improvement for most adaptive configurations, and F24, for which we see very negative results.

First, we look at which static configurations have been selected, and how they are used within the adaptive configurations. To do this, we introduce what we call *combined module activation plots*. These plots consist of two parts, corresponding to C_1 and C_2 respectively. In each of these subplots, every line indicates a configuration. The lowest line corresponds with the theoretically best adaptive configuration, increasing from there.

In Figure 5.9a and 5.9b we see these combined module activation plots for the selected adaptive configurations for F10 and F24 respectively. These figures clearly show that for F10 there is a pattern present among the adaptive configurations: the



(a) F10 **(b)** F24

Figure 5.9: Combined module activation plots for the 50 best adaptive configurations for F10 and F24.

modules TPA and threshold start activated and in almost all cases get turned off after the switchpoint. Such patterns are not present in the adaptive configurations for F24. This seems to indicate that for F24 the switches are mostly chosen because of small variances between the different configurations, instead of actual inherent properties of the configurations to perform well at certain points of the search.

5.2.5 Summary of Results

From our experiments, we found large differences in the potential of our approach between functions. For some functions, such as F10, our approach seems quite stable, resulting in improvements of over 10% for several adaptive configurations, as can be seen in Figure 5.7. However, this is not representative of all functions, as for several functions few (or any) adaptive configurations manage to outperform the static configurations.

5.3 Per-run Dynamic Algorithm Selection

Since we have now verified that switching between configurations of a single algorithm can achieve performance gains on some benchmark functions, we now place our focus on the problem of switching between different algorithm families. Our goal here is

to create a switching procedure where the algorithm to switch to is based on information collected during the optimization process, rather than a fixed, predetermined algorithm. We coin this **per-run algorithm selection** to refer to the fact that we make use of information gained by running an initial optimization algorithm ($A1$) during a single run to determine which algorithm should be selected for the remainder of the search. This second algorithm ($A2$) can then be warm-started, i.e., initialized appropriately using the knowledge of the first one. The pipeline of the approach is shown in Figure 5.10.

To extract relevant information about the problem instances, we rely on ELA features computed using samples and evaluations observed by the initial algorithm's search trajectory, i.e., *local* landscape features. Intuitively, we consider the problem instance as perceived from the algorithm's viewpoint. In addition, we make use of an alternative aspect that seems to capture critical information during the search procedure – the algorithm's internal state, quantified through a set of state variables at every iteration of the initial algorithm. To this end, we choose to track their evolution during the search by computing their corresponding *time-series* features.

Using the aforementioned values to characterize problem instances, we build algorithm selection models based on the prediction of the fixed-budget performance of the second solver on those instances, for different budgets of function evaluations. We train and test our algorithm selectors on the BBOB problems, and extend the testing on the YABBOB collection of the Nevergrad platform [200]. We show that our approach leads to promising results with respect to the selection accuracy and we also point out interesting observations about the particularities of the approach.

5.3.1 Data Collection

Problem Instance Portfolio. To implement and verify our proposed approach, we make use of a set of black-box, single-objective, noiseless problems. The data set is the BBOB suite from the COCO platform [95], which is a very common benchmark set within numerical optimization community. This suite consists of a total of 24 functions, and each of these functions can be changed by applying pre-defined transformations to both its domain and objective space, resulting in a set of different instances of each of these problems that share the same global characteristics [96].
Another considered benchmark set is the YABBOB suite from the Nevergrad platform [200], that contains 21 black-box functions, out of which we keep 17. By definition, YABBOB problems do not allow for generating different instances.

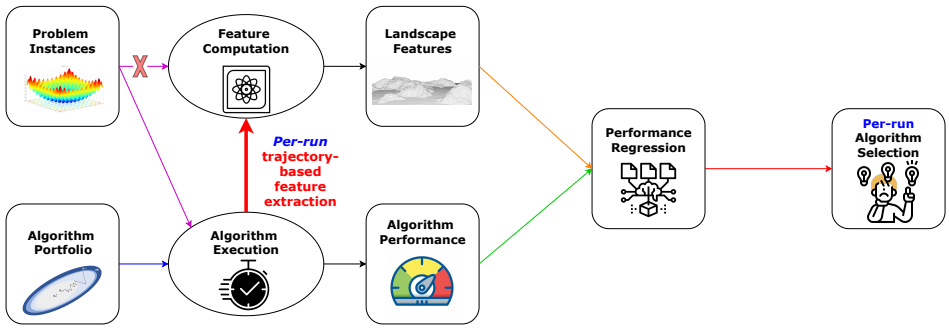


Figure 5.10: Per-run algorithm selection pipeline. The overhead cost of computing ELA features per problem instance is circumvented via collecting information about the instance during the default optimization algorithm run.

Algorithm Portfolio. As our algorithm portfolio, we consider the one used in [110, 210]. This gives us a set of 5 black-box optimization algorithms: MLSL [115, 116], BFGS [29, 77, 85, 212], PSO [123], DE [219] and CMA-ES [88]. Since for the CMA-ES we consider two versions from the modular CMA-ES framework [51] (elitist and non-elitist), this gives us a total portfolio of 6 algorithm variants.

Warm-starting. To ensure we can switch from our initial algorithm (A1) to any of the others (A2), we make use of a basic warm-starting approach specific to each algorithm. For the two versions of modular CMA-ES, we do not need to explicitly warm-start, since we can just continue the run with the same internal parameters and turn on elitist selection if required. The detailed warm-start mechanisms are discussed in [110].**Performance Data.** For our experiments, we consider a number of data collection settings, based on the combinations of dimensionality of the problem, where we use both 5- and 10-dimensional versions of the benchmark functions, and budget for A1, where we use $30 \cdot d$ budget for the initial algorithm. This is then repeated for all functions of both the BBOB and the YABBOB suite. For BBOB, we collect 100 runs on each of the first 10 instances, resulting in 1 000 runs per function. For YABBOB (only used for testing), we collect 50 runs on each function (due to no instances in Nevergrad).

In Figure 5.11, we show the performance of the six algorithms in our portfolio in the 5-dimensional case. Since the A1 budget is $30 \cdot d = 150$, the initial part of the search is the same for all switching algorithms until this point. In the figure, we can see that, for some functions, clear differences in performance between the algorithms appear very quickly, while for other functions the difference only becomes apparent after some more

evaluations are used. This difference leads us to perform our experiments with three budgets for the A2 algorithm, namely $20 \cdot d$, $70 \cdot d$, and $170 \cdot d$.
To highlight the differences between the algorithms for each of these scenarios, we can show in what fraction of runs each algorithm performs best. This is visualized in Figure 5.12. Here we can see that while some algorithms are clearly more impactful than others, the differences between them are still significant. This indicates that there would be a significant difference between a virtual best solver which selects the best algorithm for each run and a single best solver which uses only one algorithm for every run.

5.3.2 Experimental Setup

Adaptive Exploratory Landscape Analysis. As previously discussed, the *per-run* trajectory-based algorithm selection method consists of extracting ELA features from the search trajectory samples during a single run of the initial solver. A vector of numerical ELA feature values is assigned to each run on the problem instance, and can be then used to train a predictive model that maps it to different algorithms' performances on the said run. To this end, we use the ELA computation library named FLACCO [127].
Among over 300 different features (grouped in feature sets) available in FLACCO, we only consider features that do not require additional function evaluations for their computation, also referred to as *cheap features* [10]. They are computed using the fixed initial sample, while *expensive features*, in contrast, need additional sampling during the run, an overhead that makes them more inaccessible for practical use. For the purpose of this work, as suggested in preliminary studies [109, 110], we use 38 cheap features most commonly used in the literature, namely those from *y-Distribution*, *Levelset*, *Meta-Model*, *Dispersion*, *Information Content* and *Nearest-Better Clustering* feature sets.
We perform this per-run feature extraction using the initial $A1 = 30 \cdot d$ budget of samples and their evaluations per each run of each of the first 10 instances of each of the 24 BBOB problems, as well as 17 YABBOB problems (that have no instances) in dimensionalities 5 and 10.
Time-Series Features. In addition to ELA features computed during the optimization process, we consider an alternative – *time-series* features of the internal states of the CMA-ES algorithm. Since the internal variables of an algorithm are adapted during the optimization, they could potentially contain useful information about the

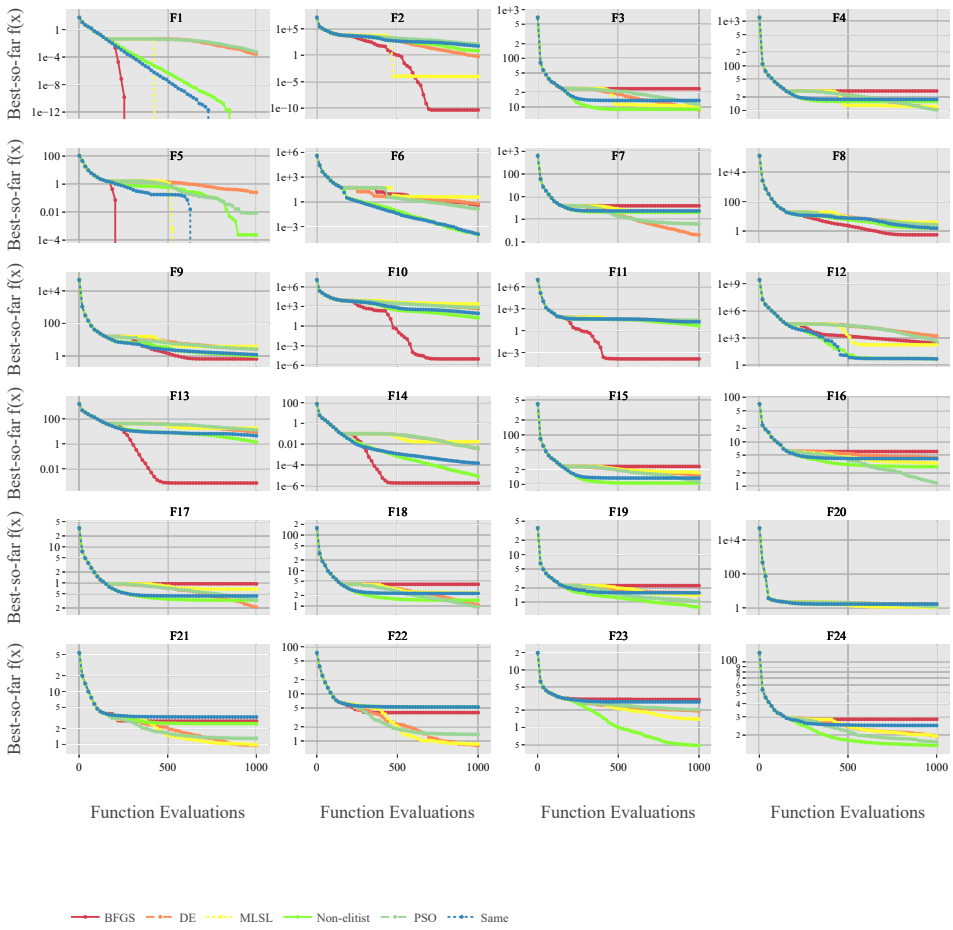


Figure 5.11: Mean best-so-far function value (precision to global optimum) for each of the six algorithms in the portfolio. For computational reasons, each line is calculated based on a subset of 10 runs on each of the 10 instances used, for a total of 100 runs. Note that the first 150 evaluations for each algorithm are identical, since this is the budget used for A1. Figure generated using IOHanalyzer [255].

current state of the optimization. Specifically, we consider the following internal variables: the step-size σ, the eigenvalues of covariance matrix $\vec{v}$, the evolution path $\vec{p}_c$ and its conjugate $\vec{p}_\sigma$, the Mahalanobis distances from each search point to the center of the sampling distribution $\vec{\gamma}$, and the log-likelihood of the sampling model $\mathcal{L}\left(\vec{m}, \sigma^2, \mathbf{C}\right)$. We consider these dynamic strategy parameters of the CMA-ES as a multivariate real-valued time series, for which at every iteration of the algorithm, we compute one data point of the time series as follows: $\forall t \in [L]$:

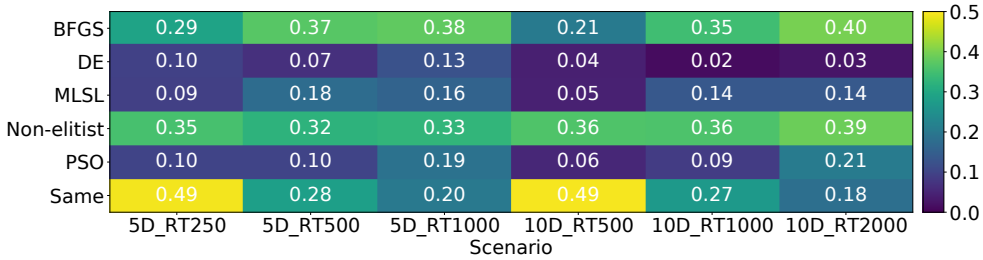


Figure 5.12: Heatmap showing for each scenario (with respect to the dimensionality and A2 budget, encoded in that order in the x-axis labels) in what proportion of runs each algorithm reaches the best function value. Note that these value per scenario can add to more than 1 because of ties.

$\vec{\psi}_t := \left(\sigma, \mathcal{L}(\vec{m}, \sigma^2, \mathbf{C}), ||\vec{v}||, ||\vec{p}_\sigma||, ||\vec{p}_c||, ||\vec{\gamma}||, \text{mean}\,\vec{v}, \text{mean}\,\vec{p}_\sigma, \text{mean}\,\vec{p}_c, \text{mean}\,\vec{\gamma}\right)^\top$,
where L represents the number of iterations these data points were sampled, which equals the A1 budget divided by the population size of the CMA-ES. In order to store information invariant to the problem dimensionality, we compute the component-wise average $\text{mean}\,\vec{x}$ and norm $||\vec{x}|| = \sqrt{\vec{x}^\top \vec{v}}$ of each vector variable.
Given a set of m feature functions $\{\phi_i\}_{i=1}^m$ from TSFRESH [42] (where $\phi_i \colon \mathbb{R}^L \to \mathbb{R}$), we apply each feature function over each variable in the collected time series. Examples of such feature functions are autocorrelation, energy and continuous wavelet transform coefficients. In this paper, we take this entire time series (of length L) as the feature window. We employ all 74 feature functions from the TSFRESH library, to compute a total of 9 444 time-series features per run. After the feature generation, we perform a feature selection method using a Random Forests classifier trained to predict the function ID, for computing the feature importance. We then select only the features whose importance is larger than 2×10^{-3}. This selection procedure yields 129 features, among which features computed on the Mahalanobis distance and the step-size σ are dominant. More details on this approach can be found in [52].
Regression Models. To predict the algorithm performance after the A2 budget, we use as performance metric the target precision reached by the algorithm in the fixed-budget context (i.e., after some fixed number of function evaluations). We create a mapping between the input feature data, which can be one of the following: (1) the trajectory-based representation with 38 ELA features per run (ELA-based AS), (2) the trajectory-based representation with 129 time-series (TS) features per run (TS-based AS), or (3) a combination of both (ELA+TS-based AS), and the target precision of different algorithm runs. We then train supervised machine learning (ML) regression

models that are able to predict target precision for different algorithms on each of the trajectories involved in the training data. Following some strong insights from [107] and subsequent studies, we aim at predicting the logarithm ($\log_{10}$) of the target precision, in order to capture the order of magnitude of the distance from the optimum. In our case, since we are dealing with an algorithm portfolio, we have trained a separate single target regression (STR) model for each algorithm involved in our portfolio. We opt for using a random forest (RF) regression, as previous studies have shown that it provides promising results for automated algorithm performance prediction [108]. To this end, we use the RF implementation from the Python package SCIKIT-LEARN [186].

Evaluation Scenarios. To find the best RF hyperparameters and to evaluate the performance of the algorithm selectors, we have investigated two evaluation scenarios:

(1) Leave-instance out validation: in this scenario, 70% of the instances from each of the 24 BBOB problems are randomly selected for training and 30% are selected for testing. Put differently, all 100 runs for the selected instance will either appear in the training or the test set. We thus end up with 16 800 trajectories used for training and 7 200 trajectories for testing.

(2) Leave-run out validation: in this scenario, 70% of the runs from each BBOB problem instance are randomly selected for training and 30% are selected for testing. Again, we end up with 16 800 trajectories used for training and 7 200 trajectories for testing.

We repeat each evaluation scenario five independent times, in order to analyze the robustness of the results. Each time, the training data set was used to find the best RF hyperparameters, while the test set was used only for evaluation of the algorithm selector.

Hyperparameter Tuning for the Regression Models. The best hyperparameters are selected for each RF model via grid search for a combination of an algorithm and a fixed A2 budget. The training set for finding the best RF hyperparameters for each combination of algorithm and budget is the same. Four different RF hyperparameters are selected for tuning: (1) *n_estimators*: the number of trees in the random forest; (2) *max_features*: the number of features used for making the best split; (3) *max_depth*: the maximum depth of the trees, and (4) *min_samples_split*: the minimum number of samples required for splitting an internal node in the tree. The search spaces of the hyperparameters for each RF model utilized in our study are presented in Table 5.1.

Per-run Algorithm Selection. In real-world dynamic AS applications, we rely on the information obtained within the current run of the initial solver on a particular

Table 5.1: RF hyperparameter names and their corresponding values considered in the grid search.

Hyperparameter	Search space
n_estimators	$[100, 300]$
max_features	$[\text{AUTO}, \text{SQRT}, \text{LOG2}]$
max_depth	$[3, 5, 15, \text{NONE}]$
min_samples_split	$[2, 5, 10]$

problem instance to make our decision to switch to a better suited algorithm. A randomized component of black-box algorithms comes into play here, as one algorithm's performance can vastly differ from one run to another on the very same problem instance.

We estimate the quality of our algorithm selectors by comparing them to standard baselines, the virtual best solver (VBS) and the single best solver (SBS). As we make a clear distinction between per-run and per-instance perspective, in order to compare we need to suitably aggregate the results. Our baseline is the *per-run VBS*, which is the selector that always chooses the real best algorithm for a particular run on a certain problem (i.e., function) instance. We then define VBS_{iid} and VBS_{fid} as virtual best solvers on instance and problem levels, i.e., selectors that always pick the real best algorithm for a certain instance (across all runs) or a certain problem (across all instances). Last, we define the SBS as the algorithm that is most often the best one across all runs.

For each of these methods, we can define their performance relative to the *per-run VBS* by considering their performance ratio, which is defined on each run as taking the function value achieved by the VBS and dividing it by the value reach by the considered selector. As such, the performance ratio for the *per-run VBS* is 1 by definition, and in $[0, 1]$ for each other algorithm selector.

To measure the performance ratio for the algorithm selectors themselves, we calculate this performance ratio on every run in the test-set of each of the 5 folds, and average these values. We point out here that the performance of different AS models are not statistically compared, since the obtained performance values from the folds are not independent [55].

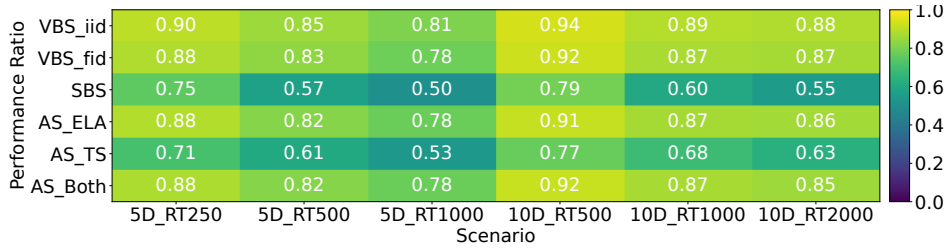


Figure 5.13: Heatmap showing for each scenario the average performance ratio relative to the per-run virtual best solver of different versions of VBS, SBS, and algorithm selectors (based on the per-instance folds). Scenario names show the problem dimensionality and the total used budget.

5.3.3 Evaluation Results: BBOB

For our first set of experiments, we train our algorithm selectors on BBOB functions using the evaluation method described in Section 5.3.2. Since we consider 2 dimensionalities of problems and 3 different A2 budgets, we have a total of 6 scenarios for each of the 3 algorithm selectors (ELA-, TS-, and ELA+TS-based). In Figure 5.13, we show the performance ratios of these selectors, as well as the performance ratios of the previously described VBS and SBS baselines. Note that for this figure, we make use of the *per-instance* folds, but results are almost identical for the *per-run* case.
Based on Figure 5.13, we can see that the ELA-based algorithm selector performs almost as well as the per-function VBS, which itself shows only minor performance differences to the per-instance VBS. We also notice that as the total evaluation budget increases, the performance of every selector deteriorates. This seems to indicate that as the total budget becomes larger, there are more cases where runs on the same instance have different optimal switches.
To study the performance of the algorithm selectors in more detail, we can consider the performance ratios for each function separately, as is visualized in Figure 5.14. From this figure, we can see that for the functions where there is a clearly optimal A2, all algorithm selectors are able to achieve near-optimal performance. However, for the cases where the optimal A2 is more variable, the discrepancy between the ELA and TS-based algorithm selectors increases.

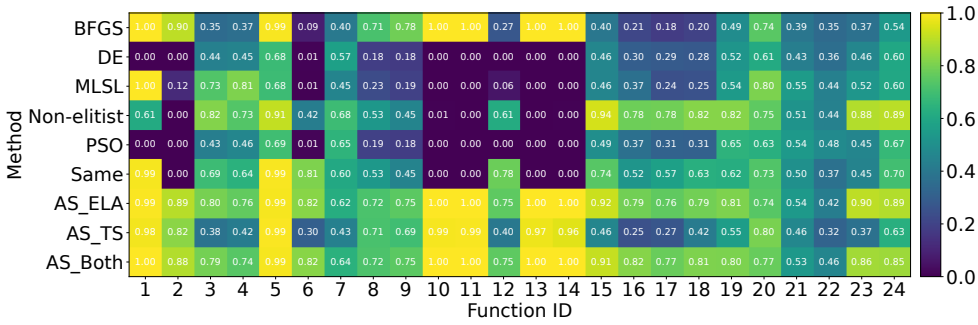


Figure 5.14: Heatmap showing for each 5-dimensional BBOB function the mean performance ratio at 500 total evaluations relative to the per-run virtual best solver, as well as the average performance ratio of each of the 3 algorithm selectors.

5.3.4 Evaluation Results: YABBOB

We now study how a model trained on BBOB problem trajectories can be used to predict the performances on trajectories not included in the training. We do so by considering the YABBOB suite from the Nevergrad platform. While there is some overlap between these two problem collections, introducing another sufficiently different validation/test suite allows us to verify the stability of our algorithm selection models. We recall that for the performance data of the same algorithm portfolio on YABBOB functions, we have target precisions for 850 runs, 50 runs per 17 problems, in all considered A2 budgets.

Training on COCO, testing on Nevergrad. This experiment has resulted in somewhat poorer performance of the algorithm selection models on an inherently different batch of problems. The comparison of the similarity between BBOB and YABBOB problems presented below nicely shows how the YABBOB problems are structurally more similar to one another than to the BBOB ones. To investigate performance flaws of our approach when testing on Nevergrad, we compare, for each YABBOB problem, how often a particular algorithm is selected by the algorithm selection model trained on the BBOB data with how often that algorithm was actually the best one. This comparison is exhibited in Figure 5.15. We observe that MLSL in particular is not selected often enough in the case of a large A2 budget, as well as a somewhat strong preference of the selector towards BFGS. An explanation for these results may be the (dis)similarities between the benchmarks. An analysis of the Pearson correlation between the trajectories on the BBOB and YABBOB suites showed limited correlation between these two suites, which might explain the poor

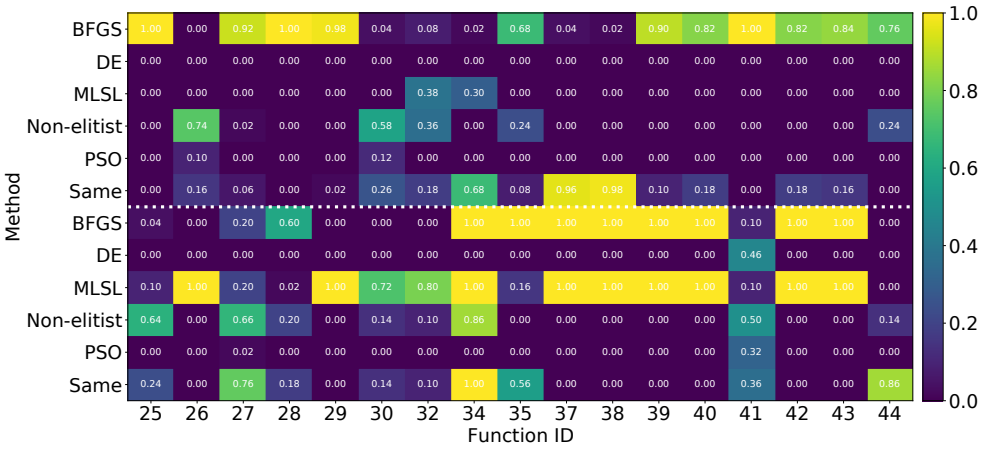


Figure 5.15: Heatmap showing for each 5-dimensional YABBOB/Nevergrad function the fraction of times each algorithm was optimal to switch to when considering a total budget of 500 evaluations (bottom) and how often each of these algorithm was selected by the algorithm selector trained on BBOB/COCO (top). Note that the columns of the bottom part can sum to more than 1 in case of ties.

generalization results [135].

5.4 When to Switch?

In the previous section, we have illustrated a way in which we can perform a single switch between optimization algorithm by using information collected during the search. While this information was only used to determine which algorithm should be switched to, we can extend this usage by not just deciding what to switch to, but whether to switch at all. In this final section of the DynAS chapter, we look at whether the search trajectories contain sufficient information to predict how beneficial a switch would be in the near future. Such a predictive model would be a first step towards a truly dynamic switching algorithm, as the model can be applied consistently during the search to detect whether switching is useful, without being restricted to a single pre-determined switching point.

5.4.1 Algorithm Portfolio

Since the potential of switching between algorithms seems to be highly dependent on the set of algorithms considered in the used portfolio [245], we consider a set of 5

algorithms:

- Covariance Matrix Adaptation Evolution Strategy *CMA-ES* [97] (implementation from the modCMA package [51])
- Differential Evolution *DE* [219] (implementation from nevergrad [200])
- Particle Swarm Optimization *PSO* [123] (implementation from nevergrad)
- Success-History based Adaptive Differential Evolution *SHADE* [223] (implementation from pyade [198])
- Constrained Optimization By Linear Approximation *Rcobyla* [190] (implementation from nevergrad)

We show the performance of these 5 algorithms on the 10-dimensional BBOB problems from the fixed-budget perspective in Figure 5.16. We see that there are significant differences in the performances of these algorithms, with no algorithm consistently dominating all others.

In addition to the algorithms, we implement warm-starting mechanisms to be able to switch between them. For the Nevergrad-based algorithms, we make use of the built-in *ask-not-told* functionality, which adapts the state of the algorithm based on a set of observations $(\{x, f(x)\})$. For starting the CMA-ES we use the warmstarting mechanism proposed in [210], which sets the center of mass and stepsize based on the 3 best solutions found so far. For switching to SHADE, we initialize the population as the last N points seen by the previous algorithm, where N is the population size.

To illustrate the usability of these warmstarting mechanisms, we investigate the performance achieved by switching from each algorithm to itself, using the described warm-starting mechanism. Since each of these warmstarting mechanisms inherently loses some information about the search process, we assume the warm-started versions will have slightly worse performance than their equivalent non-warmstarted runs. The results of running each of the 5 algorithms with 5 different points at which they are warm-started, are visualized in Figure 5.17. From this figure, we see that the performance loss from warm-starting is relatively minor, indicating that most of the relevant information is passed to the second part of the search.

5.4.2 Finding use cases using irace

To identify whether the selected portfolio can benefit from dynamically switching between algorithms, we view the problem of dynamic algorithm selection from the

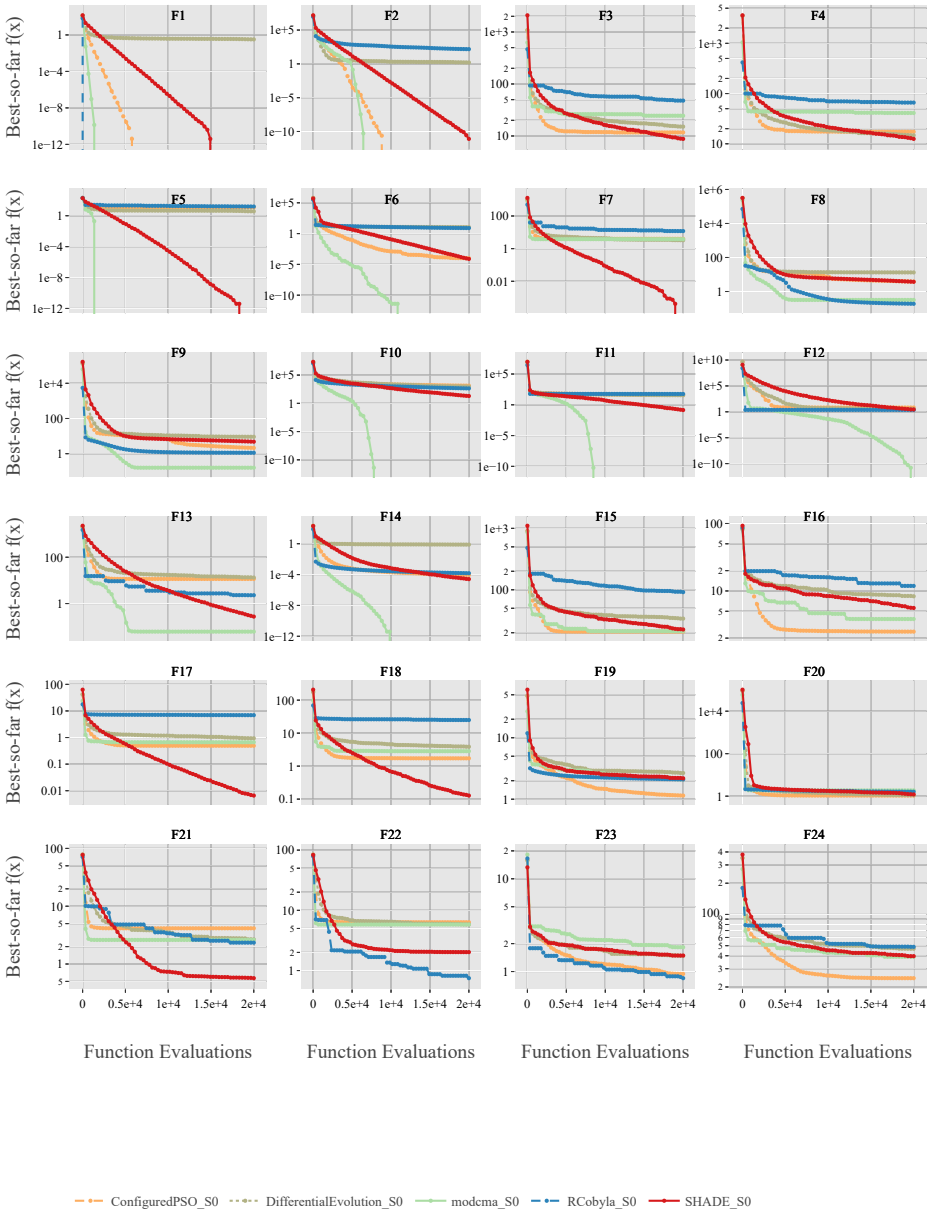


Figure 5.16: Mean function value reached, relative to the used budget, for 24 BBOB functions. Figure generated using IOHanalyzer [255]. Data available for interactive visualization at iohanalyzer.liacs.nl (IOHanalyzer dataset source 'DynAS_EvoStar23').

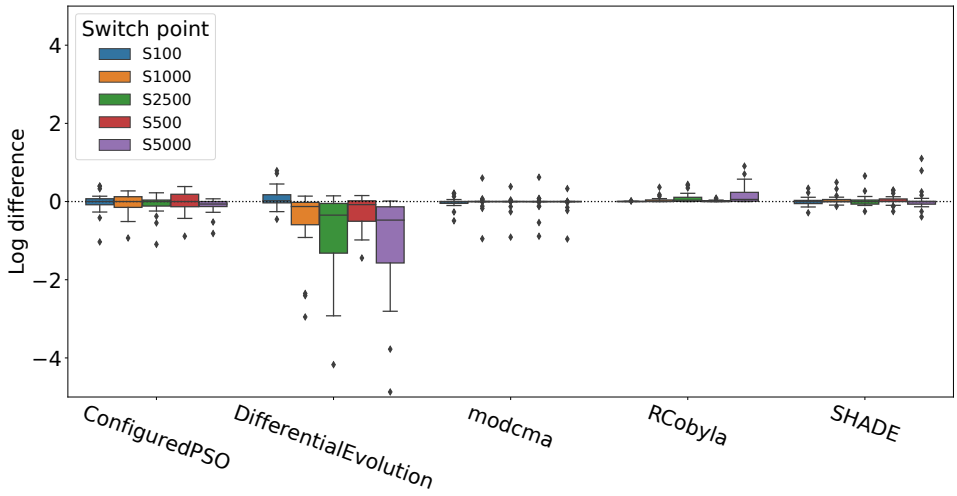


Figure 5.17: The log-distance between geometric mean of function value reached after 10 000 evaluations (limited to 10^{-8}). Differences are computed as mean with restart minus mean without restart, so negative values indicate restarts improve performance. Each box represents 24 10-dimensional BBOB problems, for each of the 5 algorithms in the portfolio for the set of 5 tested switching points.

perspective of hyperparameter tuning. We consider the dynamic algorithm to consist of three distinct parts: the first algorithm, the point at which to switch, and the second algorithm. We use irace [152] to find the configurations which reach the best function value after 5 000 function evaluations. Since irace is inherently stochastic, we perform 5 independent runs, and for each of the sets of elite configurations we perform 250 verification runs (50 runs on 5 instances). The performance of these configurations is then compared to the best static algorithm in the portfolio for each function (virtual best solver). This relative measure is visualized in Figure 5.18.

From this figure, we can see that on most problems, there are sets of configurations which seem to outperform the static algorithms. However, for some cases we see deterioration in performance compared to the VBS, indicated by negative values. This can be explained partly by the stochasticity of the algorithms: the performance observed by irace is based on a limited number of runs, and by selecting based on these limited samples can be sub-optimal when looking at the true performance distribution [247]. Additionally, there might be some cost associated with the warmstarting when the samples are collected from an initial algorithm which is not the same as the algorithm being switched to.

Since we see that there are some cases where a switch between algorithms appears

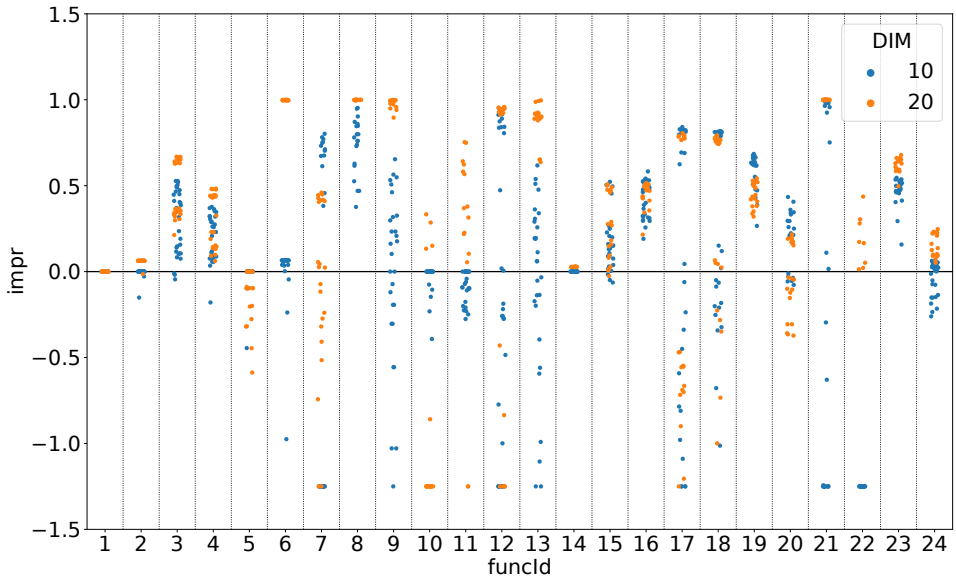


Figure 5.18: Relative improvement in terms of geometric mean of final function value of the elite configurations of irace against the virtual best solver (best static algorithm per function/dimensionality). Negative improvements are capped at -1.25 for visibility.

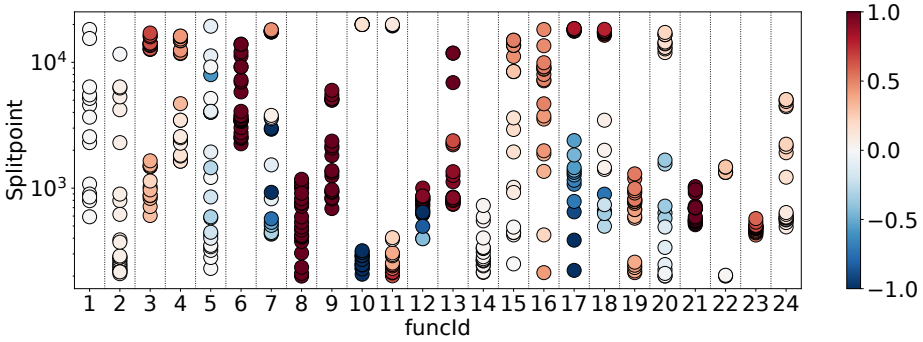


Figure 5.19: Distribution of the switch point in the elite configurations found by irace, for the 20-dimensional versions of the BBOB functions. The color of the dots corresponds to the relative improvement over VBS as shown in Figure 5.18

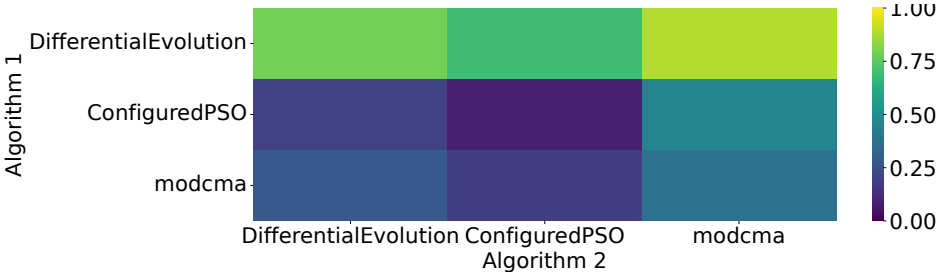


Figure 5.20: Fraction of cases in which a switch from algorithm 1 (y-axis) to algorithm 2 (x-axis) is beneficial.

beneficial, we can delve deeper into the configurations which show these benefits. In particular, we can look at the distribution of the used switch point and its correlation to the relative performance improvement, as is shown in Figure 5.19. Here, we observe that the switch points are fairly widely distributed, and that multiple different switching points can lead to similar improvements in performance.

5.4.3 Predicting Benefits of Switching

While the setup as described in Section 5.4.1 allows us to investigate the dependence of performance of a dynamic algorithm selection on the time at which the switch occurs, it does not provide directly usable insights into how this switch might be detected during the search. In order to investigate this online detection, we require a set of data where multiple switching points are attempted, such that we are able to identify on a per-run basis how beneficial each decision is. In addition, we collect features at each decision point, which can then be used to create a model to predict the observed benefits.

5.4.4 Setup

To achieve these insights into the impact of the switching point, we set up a large-scale experiment collecting the performance data for a reduced portfolio of 3 algorithms (CMA-ES, PSO and DE) on all 24 10-dimensional BBOB problems. This reduction is done to reduce computation costs. We collect 5 runs on each of the first 5 instances, and collect the full trajectory of the static algorithm up to 10000 evaluations. Then, for all switch points linearly spaced from 50 to 9500, we collect the performance data achieved when switching to each of the 3 algorithms (so we include a switch to the selected algorithm to itself) in the portfolio after another 500 evaluations.We then consider the

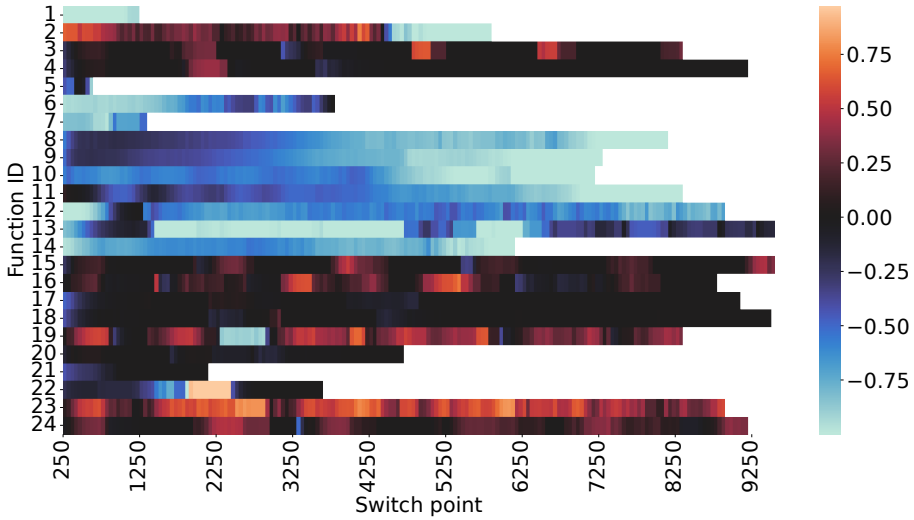


Figure 5.21: Mean relative benefit of switching from CMA-ES to DE at each of the selected switching points, for each of the 24 BBOB functions.

best fitness value reached in these 500 evaluations as the achieved performance of the dynamic algorithm. This short time-window is used to allow for the eventual creation of a dynamic switching regime which can perform more than one change during the optimization process.

In Figure 5.20 we show the fraction of cases in which a switch provides benefit over continuing the first algorithm in these 500 evaluations. From this, we see that switching is often beneficial, particularly in the case of switching to CMA. This matches our observations from Section 5.4.1, where we saw that our chosen version of DE often benefits from restarts, while the CMA-ES is the best preforming algorithm overall.

To enable an easier comparison between the algorithms, we define the target value for our model to be the relative benefit of switching after 500 evaluations, which is defined as follows:

$$r(a_s, a_r) = \left(1 - \frac{\min(a_s, a_r)}{\max(a_s, a_r)}\right)(2 \cdot \mathbb{1}_{a_s < a_r} - 1) \tag{5.1}$$

where a_s is the performance when a switch is performed, and a_r is the performance when no switch occurs. This measure takes values in $[-1, 1]$, where positive values correspond to situations where switching is beneficial, while a negative value indicates

detrimental effect of the switch.

To highlight the overall importance of the switching point, we can visualize the mean relative benefit of switching at each point in a heatmap, as is done in Figure 5.21 for the case of switching from CMA-ES to DE. Here, we see that even though the individual algorithm performance from Figure 5.16 showed that CMA-ES dominates DE in most problems, and Figure 5.20 showed that this combination is not the most promising overall, there are still many cases where a switch would still be beneficial for the performance in the next 500 evaluations. In particular, we see some clear distinctions between functions where switching is detrimental and some functions where benefits are observed, although not for all possible switching points.

In order to predict the benefit of switching at each decision point, we train a random forest model for each switch combination which outputs the relative benefit of performing the switch. The input for this model consists of the ELA features calculated on the trajectory of the first algorithm during the last $\{50, 150, 250\}$ evaluations before the switching point. We exclude the ELA features that require addition sample points, e.g., the so-called cell mapping features, resulting in 68 features in total.

This set is extended by including the diversity in the samples, both the mean component-wise standard deviation of the full set of samples ($pop\_\ div$) and the standard deviation from their corresponding fitness values ($fit\_\ div$).

Features which are constant for all samples or give NaN values for more than 90% of samples are removed from consideration. Features are then normalized (to zero mean and unit variance).

The random forest models use the default hyperparameters from sklearn [186]. Their performance is evaluated using the leave-one-function out strategy, where we train on the data from 23 BBOB functions and use the remaining one for testing. This is repeated for each function, and the results shown in this section are always on this unseen function. For our accuracy measure, we make use of the mean square error.

5.4.5 Results

In Figure 5.22, we show the overall model quality per decision point, aggregated over the algorithm which is being switched to. This aggregation allows us to gain an overview of the potential to learn the relative benefit of switching from data, which illustrates significant differences among test functions and the choice of the first algorithm. From this figure, we can see that some settings lead to very poor MSE values.

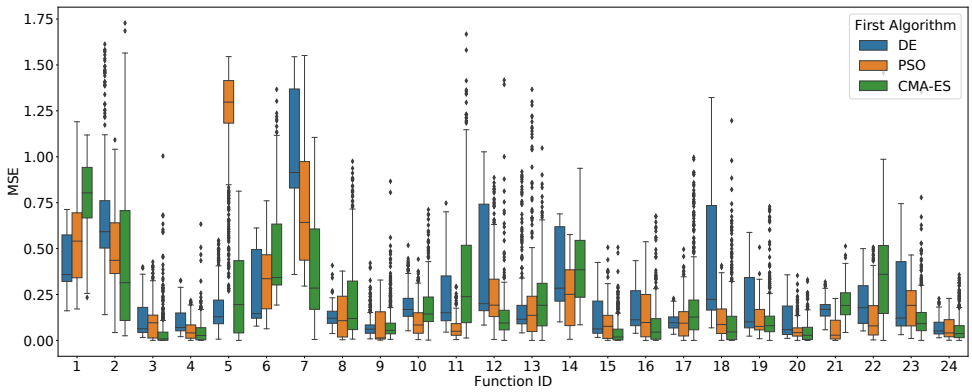


Figure 5.22: Distribution of model quality (Mean Square Error) for each function, colored according to the algorithm from which the switch occurs. Aggregated across the secondary algorithms and switch points.

This can either indicate that the model is not able to extract the needed information from the training features, or that the set of features seen on the validation-function is not consistent with the ones in the training set. For the former, it could be attributed by highly noisy feature values coming from the randomness of the first algorithm; For the latter, it is very likely that the landscape (hence the ELA features) of the test function is dissimilar to the ones in the training set. Further analyses per function/algorithm pair (Figure 5.23) aims to investigate these two possible factors. This could in part be an artifact of the leave-one-function-out validation, since the BBOB function have been originally created such that each function has distinct high-level properties [96]. However, we should note that the features we consider are trajectory-based, and are thus not necessarily as different between functions as the global version of the same features would be.

Figure 5.23 show this dependence on F7 and F15. In the top subfigure (DE to PSO on F7) we see that the actual switch (blue dots) is mostly detrimental, while the predicted value is somewhat positive, which is also reflected by quite high MSE scores of the model. Note that, the relative benefit values are not considerably noisy from the chart as the majority the sample concentrates at the very bottom, which should be learnable if the RF model were trained on this function. Hence, in this case, we conclude that, in our leave-one-function-out procedure, the model fails to generalize to function F7.

In contrast, in the bottom part of Figure 5.23 (PSO to CMA on F15), we see that

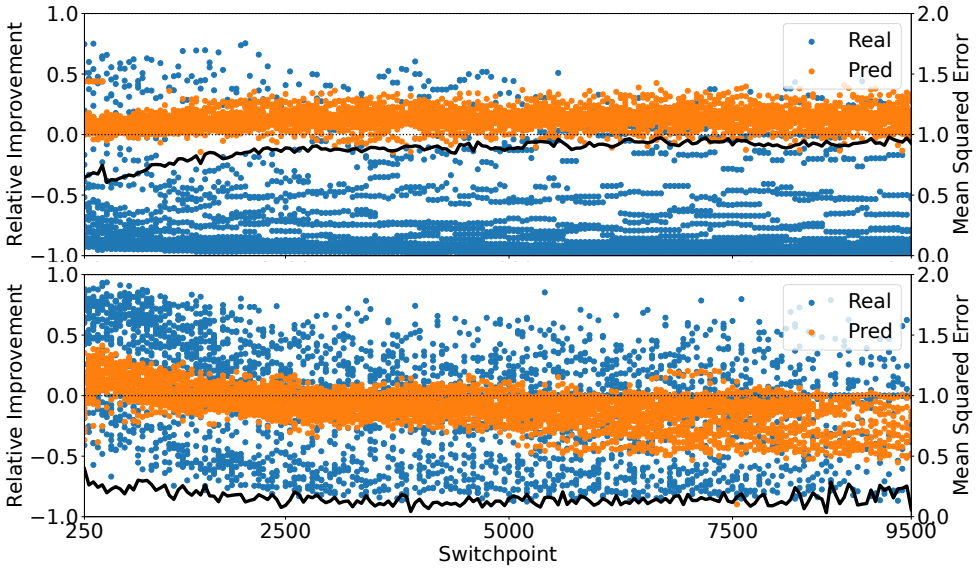


Figure 5.23: Relation between improvement and point at which the switch occurs in, for both the real improvement and the improvement predicted by the RF model. Top: switching from DE to PSO on F7. Bottom: switching from PSO to CMA on F15. The thick black line shows the MSE of the model evaluated on the selected switch point only. X-axis is shared between the two subfigures.

the overall behavior of benefit decreasing as the search continues is quite well captured by the predictions. There are two interesting aspects of the results: (1) the model seems to yield unbiased predictions of the relative benefit, which is strong support that the model generalizes well to F15; (2) The variance of the predictions are much smaller than that of the actual values, implying the possibility of a substantially large random noise when measuring the relative benefits (this observation matches with previous studies on the intrinsic large stochasticity of iterative optimization heuristics [247]). The impact of this noise might be reduced in future by performing the switch multiple times from the same switching point, leading to more stable training data.

5.4.6 Impact of Features

In addition to considering the accuracy of the trained models, we can also use the models themselves to get insights into the underlying structure of the local landscapes as seen by the algorithms. In particular, we make use of Shapley additive explanations (SHAP [157]) to gain insight into the contribution of the ELA features to the final

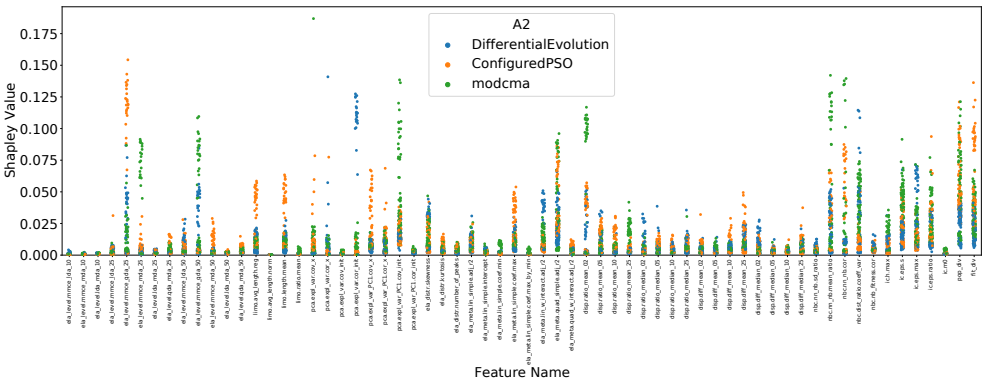


Figure 5.24: Shapley values of the features in each of the models which predict the real-valued improvement of the switch. Each dot corresponds to one model, trained on 23 functions, where the SHAP-values are calculated on the function which has been left out. Since we have 3 A1 algorithms, this leads to a total of 72 data points per feature.

predictions. Since we consider a multitude of models, we consider the distributions of Shapley values of each feature, aggregated across functions and algorithms. This is visualized in Figure 5.24.

Since Figure 5.24 is colored according to the algorithm being switched to, we can observe some interesting differences. Specifically, we see that the largest Shapley values are clearly present for different features depending on the A_2 algorithm considered. This seems to indicate that the state of the local landscape has a different effect on each algorithm. Thus, the models are indeed taking into account some specific information about the potential performance of the specific algorithm combination on which it is trained, rather than only identifying whether continuing with the current algorithm is useful in general.

By considering the local landscape features themselves without taking the models into account, we can perform dimensionality reduction to judge whether there are any patterns present in the landscape which could potentially be exploited. We make use of UMAP [163], and visualize the features obtained during the runs of CMA-ES in Figure 5.25. While this figure shows some clear clusters of similar values of the relative benefit of switching, there exist some regions where this distinction is not as clear. Based on this observation, it seems likely that the model quality can be further improved, although it is still limited by the inherent stochasticity in the dynamic algorithm selection task.

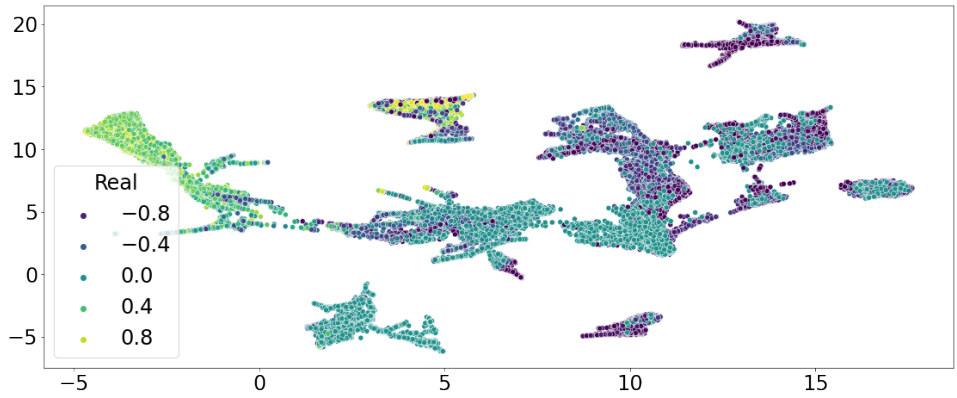


Figure 5.25: UMAP embedding of all datapoints from the CMA to DE model, where the color corresponds to the relative benefit of performing the switch.