Universiteit Leiden

Master Computer Science

Integrating Algorithm Selection and Configuration:
A Proof of Concept for the Modular CMA-ES

Name:	Diederick Vermetten
Student ID:	1603094
Date:	16/10/2019
Specialisation:	Computer Science and Advanced Data Analytics
1st supervisor:	Thomas Bäck
2nd supervisor:	Hao Wang
External Supervisor:	Carola Doerr (Sorbonne Université, France)

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

Abstract

When faced with a specific (class of) optimization problem, choosing which algorithm to use is always a tough choice. Not only is there a vast variety of algorithms to select from, but these algorithms often contain many hyperparameters, which need to be tuned in order to achieve the best performance possible. Usually, this problem is separated into two parts: *algorithm selection* and *algorithm configuration*. In this thesis, we aim to combine this into a single approach for choosing (and tuning) the best out of the $4,608$ variants of the Covariance Matrix Adaptation Evolution Strategies (CMA-ES) provided by the modEA framework [vRWvLB16].

Building upon the promising results presented in [VvRBD19], we first explore the effect of hyperparameter tuning on the performance of the CMA-ES variants. To this end, study how different hyperparameter tuning methods, MIP-EGO and irace, compare on the *algorithm configuration* problem. Based on these results, we find that a simple sequential execution of *algorithm selection* and hyperparameter tuning will not always result in finding the best performance, and doing so requires an exponentially increasing amount of data as the algorithm space grows.

We also consider an alternative approach, which integrates algorithm selection and configuration into a single problem. We use irace and MIP-EGO, to optimize both the hyperparameters and the algorithm variant at the same time. Following [Bel17], the tuning of three chosen hyperparameters which are present in all CMA-ES variants within the modEA framework. We find that this integrated approach is promising, achieving better performance than the sequential approach, while requiring significantly fewer function evaluations.

1 Introduction

Optimization is an important sub-domain of both Mathematics and Computer Science. Over the years it has seen an increasing amount of interest from many researchers. This caused an enormous increase in the amount of optimization algorithms which are available to solve any kind of problem. These algorithms keep being developed further, leading to an incredible variety of slight variations, each of which with its own benefits and detriments on specific classes of problems. However, it has long been known that no single algorithm can outperform all others on all functions, an observation commonly referred to as the *no-free-lunch theorem* [WM97]. This fact leads to a new challenge for researchers and practitioners alike: How to choose which algorithm to use for which problem? Even when limiting ourselves to just a single class of algorithm, the number of variants can be daunting, leading practitioners to resort to a few standard versions, which might not be well suited to their problems.

In this work, we study a class of Covariance Matrix Adaptation Evolution Strategies (CMA-ES). Even tough it is a variant on the general Evolution Strategy in itself, many different variations on CMA-ES have been created. Through experimentation, it has been shown that no single variant outperforms all others [Han09, BFK13]. This gives rise to even more practical considerations: which algorithm, and which variant of that algorithm, is best suited to a particular class of problems? This problem is called the *algorithm selection* problem, and has been studied in many different fields. However, the variant of CMA-ES does not completely define how well it performs on different kinds of functions. The hyperparameters of the algorithm also play a large role in its performance [Bel17]. The selection of which hyperparameter settings to use for an algorithm on a certain kind of function is called the *algorithm configuration* problem, which we also study in detail.

In this work, we use the modular CMA-ES framework, modEA, as introduced in [vR18], to create variations of the CMA-ES. This framework implements a standard CMA-ES skeleton, with the option to enable or disable many modules, each of which implements some modifications to the standard CMA-ES. This framework is used to create the CMA-ES variants discussed in this work. We extend previous work done in [VvRBD19] by looking at hyperparameter optimization of the modular CMA-ES. To combine algorithm selection with algorithm configuration, we use a one-search-space approach, which optimizes algorithm variant and corresponding hyperparameters at the same time.

The remainder of this thesis is structured as follows: Section 2 introduces the previous work which relates to our topic: the modEA framework, our test-bed and the used performance methods. This section also introduces hyperparameter tuning and explains the methods used in the experiments, for which experimental setup is introduced in Section 3. The results are split into two parts: Section 4 describes the experiments which focus on hyperparameter tuning, while Section 5 covers the integration of algorithm selection and configuration into a single approach. This work is concluded in Section 6 and future work is discussed in Section 7.

2 Related Work

2.1 The Modular CMA-ES

We build our work upon the modular CMA-ES framework (modEA), which was first introduced in [vRWvLB16]. The modEA package, implemented in python, is freely available at [vR18]. This framework implements a general structure for CMA-ES, with the option to customize the algorithm by selecting modules to turn on or off. These modules represent previously introduced variations on the default CMA-ES, such as elitism, active update, etc. In total, 11 modules have been implemented, as shown in Table 1. Of these modules, 9 are binary and 2 are ternary, allowing for a total of $4{,}608$ different possible CMA-ES variants, which we will from now on refer to as **configurations**. In this thesis, the configurations will often be referred to by their ID instead of their vector of module activations. These representations can be translated using the formula in Appendix D.

The modules available in modEA can be combined to create many popular variants of CMA-ES, such as the set of commonly used configurations are shown in Table 2. However, since they can be activated in any combination, many new CMA-ES configurations can be created, which have the potential to improve significantly over the common CMA-ES variants [vRWvSB17]. Next to these modules, the modEA framework also allows the option to change many different hyperparameters, such as population size, learning rates etc. These can also have a big impact on the performance of the algorithm, and are discussed in more detail in Section 2.5

#	Module name	0	1	2
1	Active Update [JA06]	off	on	-
2	Elitism	(μ, λ)	$(\mu+\lambda)$	-
3	Mirrored Sampling [BAH+10]	off	on	-
4	Orthogonal Sampling [WEB14]	off	on	-
5	Sequential Selection [BAH+10]	off	on	-
6	Threshold Convergence [PMEVBR+15]	off	on	-
7	TPA [Han08]	off	on	-
8	Pairwise Selection [ABH11]	off	on	-
9	Recombination Weights [AJT05]	$\log(\mu+\frac{1}{2})-\frac{\log(i)}{\sum_j w_j}$	$\frac{1}{\mu}$	-
10	Quasi-Gaussian Sampling	off	Sobol	Halton
11	Increasing Population [AH05, Han09]	off	IPOP	BIPOP

Table 1: Overview of the CMA-ES modules available in the used framework. The entries in row 9 specify the formula for calculating each weight w_i.

Variant	Representation
CMA-ES	00000000000
Active CMA-ES	10000000000
Elitist CMA-ES	01000000000
Mirrored-pairwise CMA-ES	00100001000
IPOP-CMA-ES	00000000001
Active IPOP-CMA-ES	10000000001
Elitist Active IPOP-CMA-ES	11000000001
BIPOP-CMA-ES	00000000002
Active BIPOP-CMA-ES	10000000002
Elitist Active BIPOP-CMA-ES	11000000002

Table 2: Common CMA-ES Variants. A selection of ten common CMA-ES variants is listed here, taken from [vRWvLB16].

2.2 Test-bed: the COCO framework

For the benchmarking of the algorithm configurations produced by modEA, we previously used the COCO/BBOB suite of noiseless benchmarking functions [HAB$^+$16]. This suite contains 24 functions, defined in a continuous search space ($f : \mathbb{R}^d \to \mathbb{R}$), of which we use the 5D versions. An overview of some global properties of these functions is shown in Table 3. Each function can be transformed in both objective and variable space, resulting in separate instances with similar fitness landscapes. A large part of our analysis is built on data from [VvRBD19], which uses the first 5 instances of all functions, for which 5 independent runs were performed on each instance, for each configuration and each function. This data is available at [VvRBD].

2.3 Performance Measures

Since we are interested in the performance of the CMA-ES configurations, we need to define exactly which performance measures we are interested in. To be consistent with previous work, such as [VvRBD19], we decide to focus hitting times, particularly AHT and ERT, which we defined in this section.

In general, there are two distinct approaches which can be taken to analyze the performance of an optimization algorithm. The first is the fixed-budget approach, which looks at the best $f(x)$ value found after a certain budget B has been used. The second approach is fixed-target, which focuses on how many function evaluations are needed by an algorithm to reach a certain target function value ϕ for the first time. In this work, we use the fixed-target approach to analyze and compare the configurations created by modEA.

For the fixed-target analysis, a clear way to define targets across all used benchmark functions needs to be defined. The targets are set as precisions to the optimal values. We use the precision to the optimal value as a shorthand, so $\phi = 10^{-8}$ is hit when $|f_{\text{opt}} - f_{\text{best-so-far}}| \leq \phi$. When we use this shorthand, the hitting time of target ϕ is the first function evaluation for which this target is hit. We can write this as $t_i(c, f, \phi)$ to signify the hitting time of target ϕ of run i of configuration c on function f. If target ϕ is not hit, we define $t_i(c, f, \phi) = \infty$.

In essence, the hitting time is an integer-valued random variable, which we denote as $T(c, f, \phi)$. The observed hitting times $t_i(c, f, \phi)$ are then sampled from this distribution. To determine

	Function	multim.	gl.-struc.	separ.	scaling	homog.	basins	gl.-loc.
1	Sphere	none	none	high	none	high	none	none
2	Ellipsoidal separable	none	none	high	high	high	none	none
3	Rastrigin separable	high	strong	none	low	high	low	low
4	Bueche-Rastrigin	high	strong	high	low	high	med.	low
5	Linear Slope	none	none	high	none	high	none	none
6	Attractive Sector	none	none	high	low	med.	none	none
7	Step Ellipsoidal	none	none	high	low	high	none	none
8	Rosenbrock	low	none	none	none	med.	low	low
9	Rosenbrock rotated	low	none	none	none	med.	low	low
10	Ellipsoidal high-cond.	none	none	none	high	high	none	none
11	Discus	none	none	none	high	high	none	none
12	Bent Cigar	none	none	none	high	high	none	none
13	Sharp Ridge	none	none	none	low	med.	none	none
14	Different Powers	none	none	none	low	med.	none	none
15	Rastrigin multi-modal	high	strong	none	low	high	low	low
16	Weierstrass	high	med.	none	med.	high	med.	low
17	Schaffer F7	high	med.	none	low	med.	med.	high
18	Schaffer F7 mod. ill-cond.	high	med.	none	high	med.	med.	high
19	Griewank-Rosenbrock	high	strong	none	none	high	low	low
20	Schwefel	med.	deceptive	none	none	high	low	low
21	Gallagher 101 Peaks	med.	none	none	med.	high	med.	low
22	Gallagher 21 Peaks	low	none	none	med.	high	med.	med.
23	Katsuura	high	none	none	none	high	low	low
24	Lunacek bi-Rastrigin	high	weak	none	low	high	low	low

Table 3: Classification of the noiseless BBOB functions based on their properties (multi-modality, global structure, separability, variable scaling, homogeneity, basin-sizes, global to local contrast). Predefined groups are separated by horizontal lines. Table taken from to [MBT+11].

the mean of T, the most straightforward method is to use the sample mean, also referred to as the Average Hitting Time (AHT), which is defined as follows:

Definition 2.1 ((Penalized) Average Hitting Time)**.**

$$\tilde{T}(c, f, \phi) = \frac{1}{n} \sum_{i=1}^{n} \min\{t_i(c, f, \phi), P\}$$

When a run does not succeed in hitting target ϕ, we have $t_i(c, f, \phi) = \infty$. In this case, a penalty $P \geq B$ is applied. Usually, this penalty is set to ∞, in which case this value is the called the AHT. Otherwise, it is commonly referred to as penalized AHT.

The AHT is a very simple estimator for the mean of the hitting time T, but it is not a consistent unbiased estimator. For a consistent, unbiased estimator of the mean of the hitting time T, the so-called Expected Running Time (ERT) is used. This is defined as follows:

Definition 2.2 (Expected Running Time)**.**

$$\mathrm{ERT}(c, f, \phi) = \frac{\sum_{i=1}^{n} \min\{t_i(c, f, \phi), B\}}{\sum_{i=1}^{n} \mathbb{1}\{t_i(c, f, \phi), < \infty\}}$$

Previous work [AH05] has shown ERT to be a consistent, unbiased estimator of the mean of the distribution hitting times T. It is also good to note that ERT and AHT are equivalent in the case where the configuration c manages to hit target ϕ in all runs.

2.4 Overview of terminology

To remain consistent with previous work, we use the following terminology:

- **Module**: A single option within a configurable algorithm. For example: elitism in modEA.
- **Configuration**: An instantiation of a configurable algorithm according to a complete set of module settings. For example: the common configurations in Table 2.
- **Target**: A predefined value indicating a precision to the optimal value, which can be reached during an optimization run. A target is hit when $|f_{\text{opt}} - f_{\text{best-so-far}}| \leq \tau$. We use targets between 10^2 and 10^{-8}.
- **Instance**: A specific instantiation of a function. For example: instance 1 of BBOB-function 21.
- **Run**: A single execution of a configuration on a single instance of a function.

2.5 Hyperparameter Tuning

Almost every possible algorithm contains at least some amount of free choice from the user in term of parameter settings. However, these parameters can often be hidden to the user and set to a predetermined value. Often times, these default values are not the best ones for the users given problem, and their optimal setting depends on the scenario in which the algorithm will be applied. If some internal parameters are not controlled by the algorithm itself, and they have to be set from an external source, and they have an impact on the adaptation of other parameter values, these are typically called hyperparameters. To get the best performance out of any given algorithm, the value of these hyperparameters is crucial. Because of this, hyperparameter tuning has long been an important aspect of algorithm design. The simplest form of hyperparameter tuning is done by having a human expert try some different values for the available hyperparameters and choosing those best suited to their problem. However, as the amount of available computational resources is ever increasing, algorithms for hyperparameter tuning have become more and more popular as time went on.

Hyperparameter tuning can in itself be seen as an optimization problem, and thus many different techniques for automatic hyperparameter tuning have been proposed over the years. For a hyperparameter setting to be evaluated, the entire underlying algorithm needs to be evaluated. Hence, hyperparameter tuning can be viewed as an expensive evaluation function, i.e. a function for which we have a very limited budget of evaluations. Because of this, efficient global optimization techniques, such as Bayesian optimization, are popular in this domain.

2.5.1 Hyperparameters

Before deciding on which hyperparameter tuning approach to take, we first need to define which parameters we want to include in our optimization. Previous work on hyperparameter tuning in CMA-ES [Bel17] has considered the parameters c_1, c_c and c_μ to be the most interesting to study. We will follow this suggestion, and focus on these three parameters in particular. In this work, we only set the hyperparameters once, so we are performing offline hyperparameter tuning.

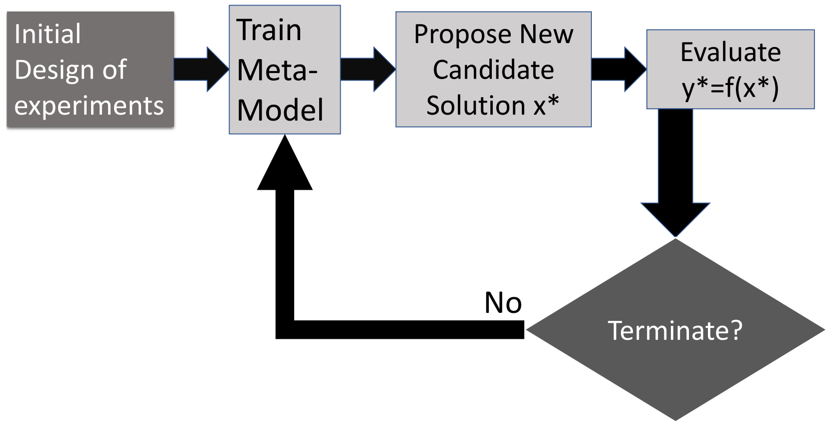

Initial Design of experiments
Train Meta-Model
Propose New Candidate Solution x*
Evaluate y*=f(x*)
Terminate?
No

Figure 1: Basic Flowchart for EGO

Since the modular CMA-ES is considered here, we have many additional parameters to be tuned when certain modules are enabled or disabled. We can even view the selection of a configuration as a hyperparameter tuning problem, and optimize both the configuration and its corresponding parameters at once. We will refer to this as the one-search-space approach, and investigate it in detail in Section 5.2. This approach requires the hyperparameter tuning method to be able to use discrete variables in the tuning.

2.5.2 Tuning methods

In this work, we consider two methods for hyperparameter optimization. Since we deal with both continuous and discrete parameters, we need an algorithm which can deal with mixed-integer input. We decided on using two different tools. The first method is Mixed Integer Parallel Efficient Global Optimization (MIP-EGO), which was presented in [WEB18] to test cooling strategies for moment-generating functions. We compare this to irace [LIDLC$^+$16], a state-of-the-art tuning method. These techniques will be described in detail in Sections 2.5.3 and 2.5.4 respectively.

2.5.3 MIP-EGO

MIP-EGO [WEB18, WvSEB17] builds upon EGO, and can deal with mixed-integer search-spaces. Because EGO is designed to deal with expensive function evaluations, and this variant has the ability to deal with continuous, discrete and categorical parameters, it is also well suited to the hyperparameter tuning task.

The general working principle of Efficient Global Optimization (EGO) algorithms is shown in Figure 1. EGO starts by sampling an initial set of solution candidates from some user-specified distribution, e.g. random or so-called quasi-random distributions (such as Latin hypercube sampling). When domain or problem specific knowledge is available, more complex distributions can be used. This process is usually referred to as the design of experiment. Based on the evaluation of these initial points, a meta-model is constructed. This is usually done by Gaussian process regression, but random forests are also often used, especially when the search space contains a mixture of real, ordinal and nominal variables. Based on this model, a new point (or set of points) is proposed according to some metric, called the acquisition function. This can be as simple as selecting the point with the largest *probability of improvement* (PI) or the largest

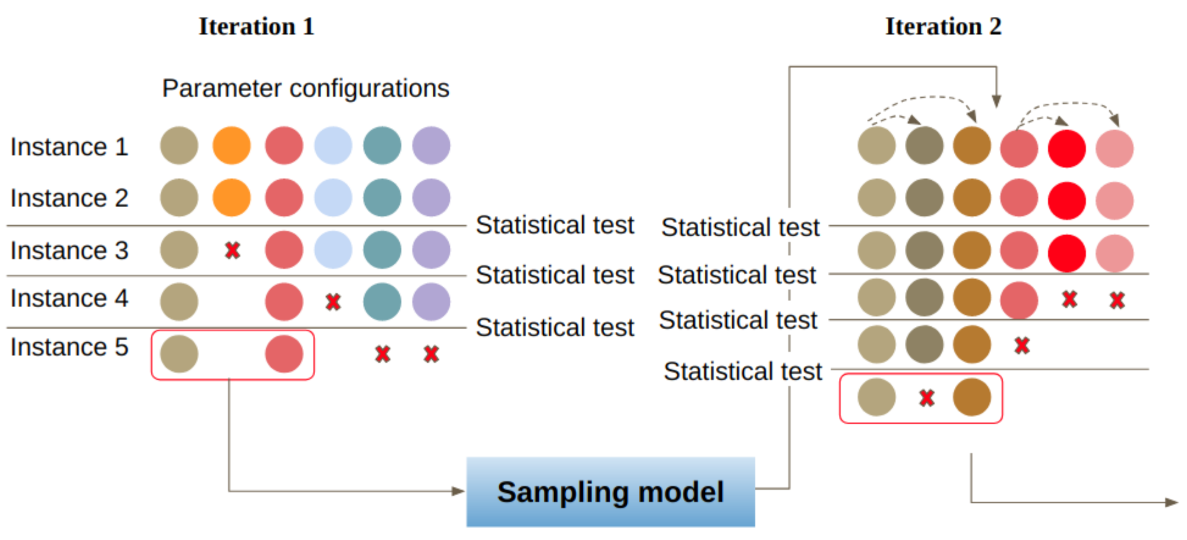


Figure 2: Illustration of basic iterated racing.

expected improvement (EI). More recently, acquisition functions based on moment-generating function of the improvement have also been introduced [WvSEB17]. After selecting the point (or points) to evaluate, the meta-model is updated according to their quality. The process is repeated until a termination criterion is met.

MIP-EGO works by first sampling an initial set of candidate points using Latin hypercube sampling, evaluating them and then building a surrogate model using Random Forests. We use the expected improvement (EI) as the acquisition function, and maximize this using a mixed-integer evolution strategy.

2.5.4 Irace

Irace is an algorithm designed for hyperparameter optimization, which is implemented in R (available at [LIPC]), and implements an iterated racing procedure [LIDLC$^+$16, LIDLSB11]. The main principle of iterated racing is shown in Figure 2. It starts by randomly sampling an initial population, and then iteratively running them on single instances. The distributions of hitting times are then compared using statistical tests, e.g. a two-sample T-test, and the candidates which are significantly worse (in the statistical sense) are eliminated. After a set number of iterations or if only a set number of candidates remain, new candidates are generated according to some sampling model and a new race is started.

For our experiments, we use the elitist version of irace with adaptive capping. This works by first sampling a set of candidate parameter settings, which can be any combination of discrete, continuous, categorical or ordinal variables. Then these parameters are used to run on some problem instances, after which a statistical test (in this thesis we use the standard t-test, but this can be improved upon in further research) is run to determine which parameter settings to discard. The remaining parameter settings are then run on more instances and continuously tested every l iterations until either only a minimal number of remaining settings or the budget of the current iteration is exhausted. The surviving candidates with the best rank, as sorted by their averaged hitting times, are selected as the elites.

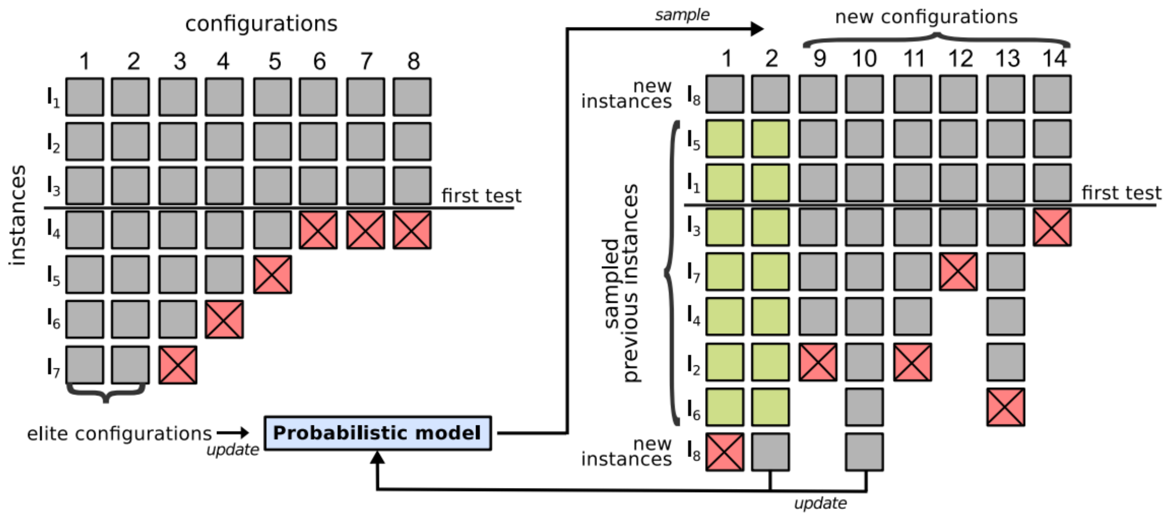


Figure 3: Illustration of elitist iterated racing as implemented in the irace-package. Figure taken from [CLIHS17].

After the racing procedure, new candidate parameter settings are generated by selecting a parent from the set of elites and mutating them. The mutation procedure re-samples all continuous parameter values from a distribution with mean at the value in the parent and set variance which decreases after each race to promote exploitation, as described in detail in [LIDLC$^+$16]. Similar mutation procedures are used for the other parameter types. After generating the new set of candidates, a new race is started with these new solutions, combined with the ones that survive from the previous race. Since we use an elitist version of irace, the elites are not discarded until the competing candidates have been evaluated on the same instances which the elites have already seen. This is done to prevent the discarding of candidates which perform well on the previous race based on only a few instances in the current race. A visual representation of the working mechanism of irace is shown in Figure 3.

Apart from using elitism and statistical tests to determine when to discard candidate solutions, we also use another recently developed extension of irace, the so called adaptive capping [CLIHS17] procedure. Adaptive capping helps to reduce the number of evaluations spent on candidates which will not manage to beat the current best. Adaptive capping enables irace to stop evaluating a candidate once it reaches a mean value which is worse than the median of the elites, indicating that this candidate is unlikely to be better than the current best parameter settings.

2.5.5 Summary

We note that we decided on using MIP-EGO and irace because they approach the problem in fundamentally different ways. MIP-EGO is designed to be efficient at balancing exploration and exploitation in low-budget scenarios, but it is not explicitly built to deal with stochastic problems. In contrast, irace is built specifically with the algorithm configuration problem in mind, so it can deal with stochasticity efficiently, but its candidate generation might be less efficient than the procedures used in EGO. These two methods both have their distinct properties, so comparing and contrasting them might give us some interesting insights into the nature of our configuration and hyperparameter space. Other commonly used hyperparameter tuning methods, such as hyperband [LJD$^+$16] and SMAC [HHLB11], were not used in this work.

3 Experimental setup

To run our experiments, the COCO BBOB [HAB$^+$16] framework is used. This is a well-established collection of benchmark problems for black-box optimization, as described in more detail in Section 2.2. More specifically, the benchmark we use is the set of 24 noiseless, 5-dimensional functions as shown in Table 3. These are the same functions as were used in [VvRBD19], for which a large amount of data is available. This data is used as the basis for the algorithm selection in Section 4.

3.1 Research questions

In these experiments, we aim to answer a few key research questions to lead us to a better understanding of the impact of hyperparameter tuning in the many-algorithm context created by modEA, as well as some insight into the properties of the hyperparameter tuning methods mentioned in Section 2.5. Specifically, we aim to answer the following questions:

- What is the general impact of hyperparameter tuning on the performance of CMA-ES configurations created by modEA? How stable is this performance?
- Is the naive approach, i.e. sequential execution of algorithm selection followed by hyperparameter tuning, effective at finding the best *(configuration, hyperparameters)*-pair? Are there configurations which perform poorly with default hyperparameters which can improve to the point where they beat the best static configurations after hyperparameter tuning?
- Are the expected values given by the hyperparameter tuning run reliable? Is the sample size large enough to get an accurate estimate of ERT?
- How well do the default hyperparameter settings perform? Are there significant differences to the optimized hyperparameter settings?
- Is there a significant difference in performance between MIP-EGO and irace when using them for hyperparameter tuning?
- Can we predict the ERT of a configuration using some basic models?
- Can we use the hyperparameter tuning methods to optimize both the configuration and the hyperparameter settings at the same time? How do MIP-EGO and irace differ on this task?
- What are the main differentiating factors between MIP-EGO and irace?

3.2 Data repository

All data produced during the experiments described in the next sections, as well as the code used to run the experiments themselves, will be made available on Github at [VWBD]. Data used from [VvRBD19], including the ERTs from 5×5 runs on all BBOB-functions, is also available on Github, at [VvRBD].

4 Sequential approaches

In this section, the algorithm selection problem and hyperparameter tuning are considered to be two completely separate problems. We will use the available 5×5 hitting time data for the algorithm selection, and focus on the hyperparameter tuning part, using both MIP-EGO and irace as hyperparameter tuning methods.

The first experiments aim to identify the potential performance gains (in terms of ERT) which can be achieved by hyperparameter tuning. To achieve this, MIP-EGO is used to optimize the hyperparameters of a large set of configurations. As mentioned in Section 2.5.1, three hyperparameters have been selected to be tuned: c_1, c_c and c_μ. MIP-EGO is given a budget of 200 evaluations, each consisting of 5×5 runs, split into an initial random sampling size of $19 + 1$ and a further set of 180 iterations. The goal for the optimization is to minimize ERT over 5 runs on 5 instances of the selected function. To ensure MIP-EGO can find a ‘reasonable’ parameter setting, the default values of the hyperparameters are always included in the initial population. This removes the possibility for negative improvements, given we ignore the effects of variance.

To explore the effect of hyperparameter tuning on the performance of the CMA-ES configurations, we select two functions to investigate in more detail. These are F12 and F21. These functions were selected because previous research [VvRBD19] showed that they are quite distinct. Where F12 is relatively easy to solve for a CMA-ES, F21 is much more challenging, leading to an inability to reach target 10^{-8} consistently. Because of this, the target to reach is set to match the one used in [VvRBD19], shown here in Table 5. For both of these functions, we select a set of interesting configurations on which to perform the hyperparameter tuning.

4.1 Algorithm Selection

To select the sets of configurations on which to perform the hyperparameter tuning, the ERTs based on 25 avaialbe runs can be used. For F21, we select the 50 best modular CMA-ES configurations, add a set of 6 common configurations (from Table 2) to it. This leads to a set of 56 distinct configurations, on which MIP-EGO is run. We then select the best hyperparameter setting evaluated by MIP-EGO and denote this as the tuned hyperparameter setting. The resulting ERT from this hyperparameter tuning will be shown in Figure 4a, which shows promising results. However, these ERTs might not be reliable, since previous work [VvRBD19] found that ERTs based on 25 runs are not always reliable. Because of this, these results are investigated in detail in Section 4.2.

Similarly, for F12, a set of configurations on which we perform hyperparameter tuning is selected. These configurations can be split into three distinct groups:

- Group 1: Those selected using the process of the two-stage approach as described in [VvRBD19]. This approach selects the 50 best static configurations and combines this with the configurations which are used in the 50 best switching configurations.
- Group 2: The static configurations ranked 200-256 (ranked on ERT at target 10^{-8}, of the set of configurations with the (B)IPOP module disabled).

- Group 3: A set of random configurations, with the restriction that the (B)IPOP module is not active.

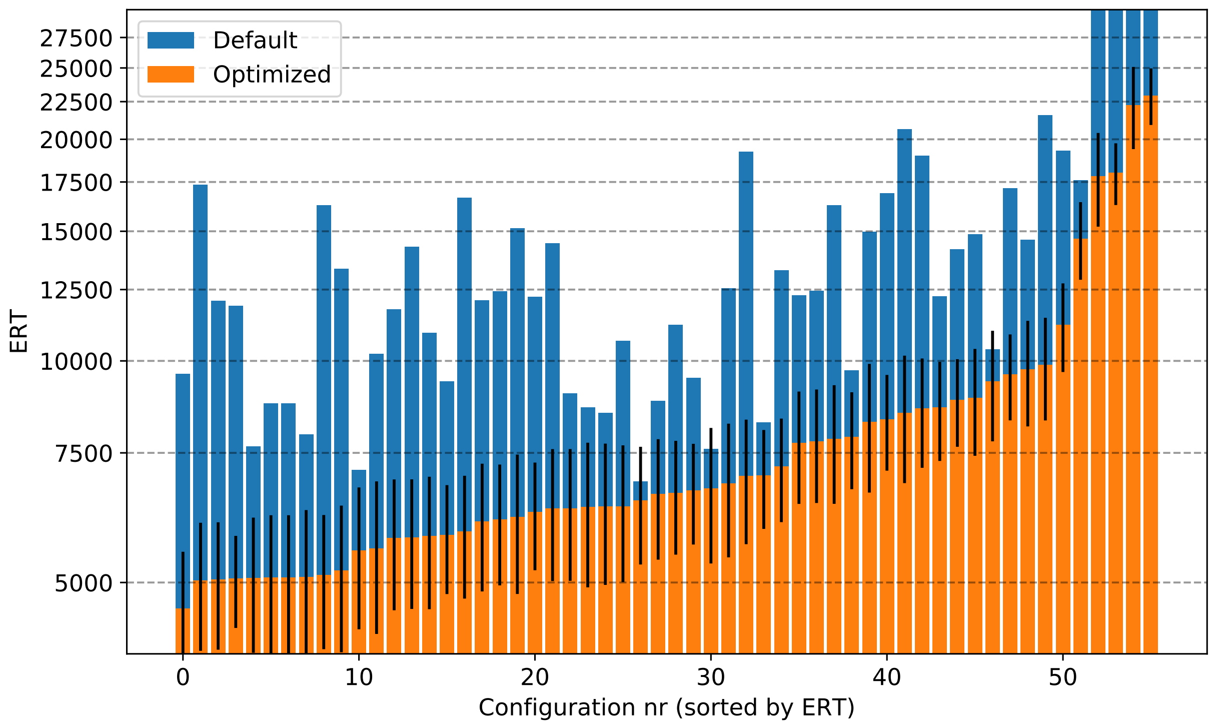
Note that this set only contains configurations without the (B)IPOP module to allow the re-using of data gathered in [VvRBD19]. For each of these configurations, MIP-EGO is run with a budget of 200 evaluations to tune the hyperparameters (c_1, c_c, c_μ).

4.2 The curse of high variance

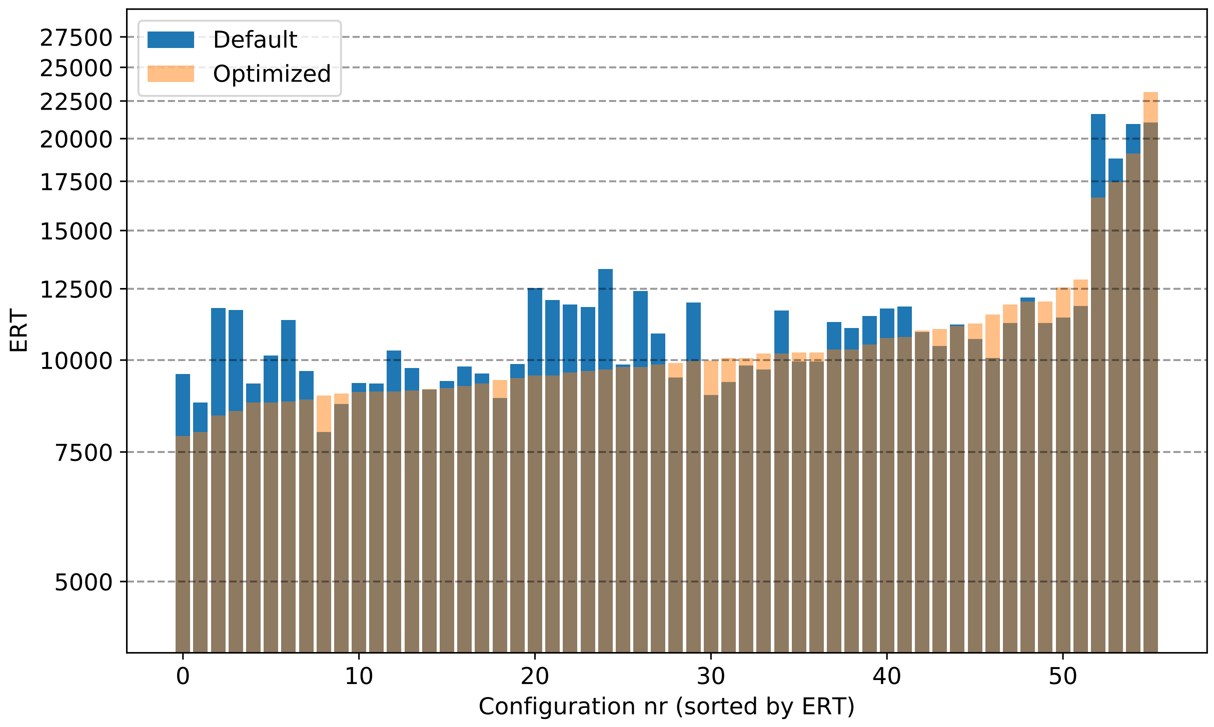
The previous section explained the algorithm selection procedure. On the selected configurations, MIP-EGO is run to find the best hyperparameter settings. However, MIP-EGO tries to optimize ERT based on 25 runs, which has been shown to be unreliable at times [VvRBD19]. Because of this, the configurations are always verified on 250 runs (50 runs on 5 instances each), both with their default hyperparameters and the tuned hyperparameters as found by MIP-EGO.
The results from the hyperparameter on F21 are shown in Figure 4. From Figure 4a, it can be see that optimizing hyperparameters initially seems to have a large impact on the found ERT. Among these configurations, the average improvement found is around 45%, which is very significant. These results can then be compared to their more robust counterpart, shown in Figure 4b. From this figure, we can see that the improvements which we initially saw in Figure 4a have almost completely disappeared, from an average of around 45% to only 5%. This seems to indicate that the large improvement which were initially shown, might have been caused largely by the high variance present in the hitting times on this function.

The same procedure is performed on F12, first running MIP-EGO on all selected configurations and then rerunning the configurations 250 times to verify the resulting ERTs. The results from this are shown in Figure 5. In Figure 5a, we show the ERT with default hyperparameters vs the best ERT found in the MIP-EGO run. We see distinct differences between the groups, with the largest improvements visible for group 3, which indicates that configurations which initially perform poorly can become much better with optimized hyperparameters, while configurations which already perform quite well tend to see less benefit. If we take a look at the relative improvement obtained (from 25 runs), which is shown in Figure 6, we see this pattern even more clearly. While improvements of over 40% are rare occurrences for groups 1 and 2, they are the norm for group 3. This is caused by the fact that most configurations in group 3 have difficulty reaching the set target, thus getting large penalties to their ERT. If a better hyperparameter setting manages to reach the target value slightly more often, even if caused solely by the inherent stochasticity of the algorithm, the improvement in ERT is very large.

From Figure 5b, it can be seen that, when rerunning the tuned hyperparameter settings, the improvement over the default hyperparameters has become a smaller than those seen in Figure 5a, and more erratic. Figure 6 shows the difference in relative ERT improvement based on 25 runs against 250 runs. This figure visualizes the reduction in relative improvement achieved by the 250-run verification. From this figure, we also see that, for some configurations, there is a negative improvement from the optimized hyperparameters compared to the default ones. To get a more robust idea of this difference, the hitting time distributions are shown as a split violin plot in Figure 7. From this figure, we can see that for the first three configurations, a negative improvement in ERT is present, but the distributions are almost exactly the same. A

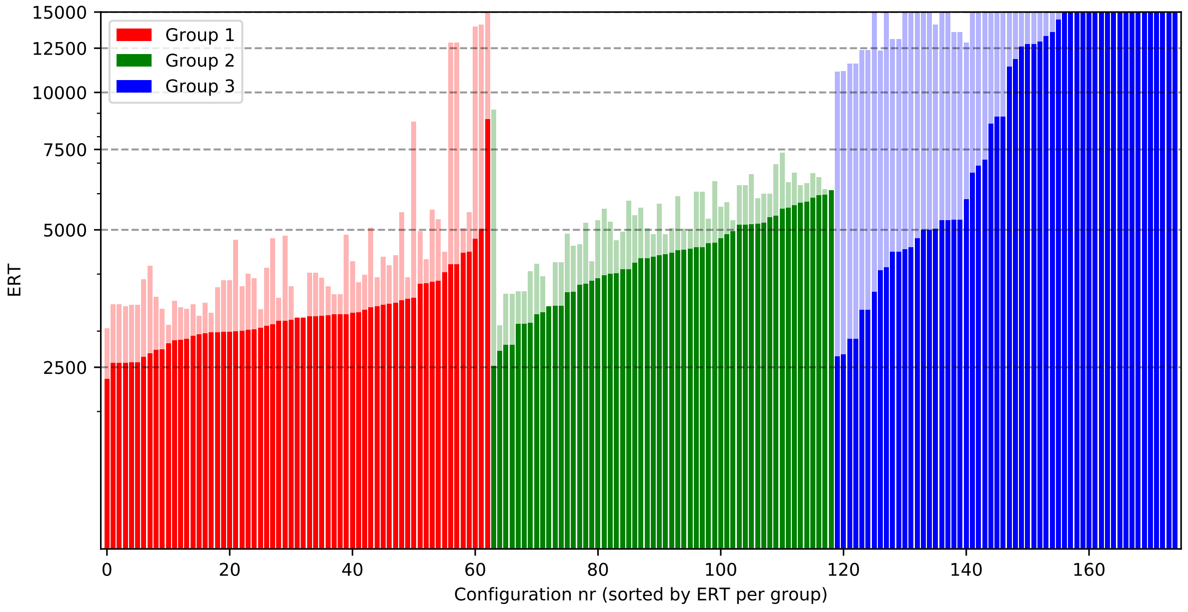


(a) ERTs based on 5 runs of 5 instances. Errorbars are shown only for the optimized hyperparameter values and indicate the 95% confidence interval of the mean. The configurations are sorted according to the optimized ERT after hyperparameter tuning.

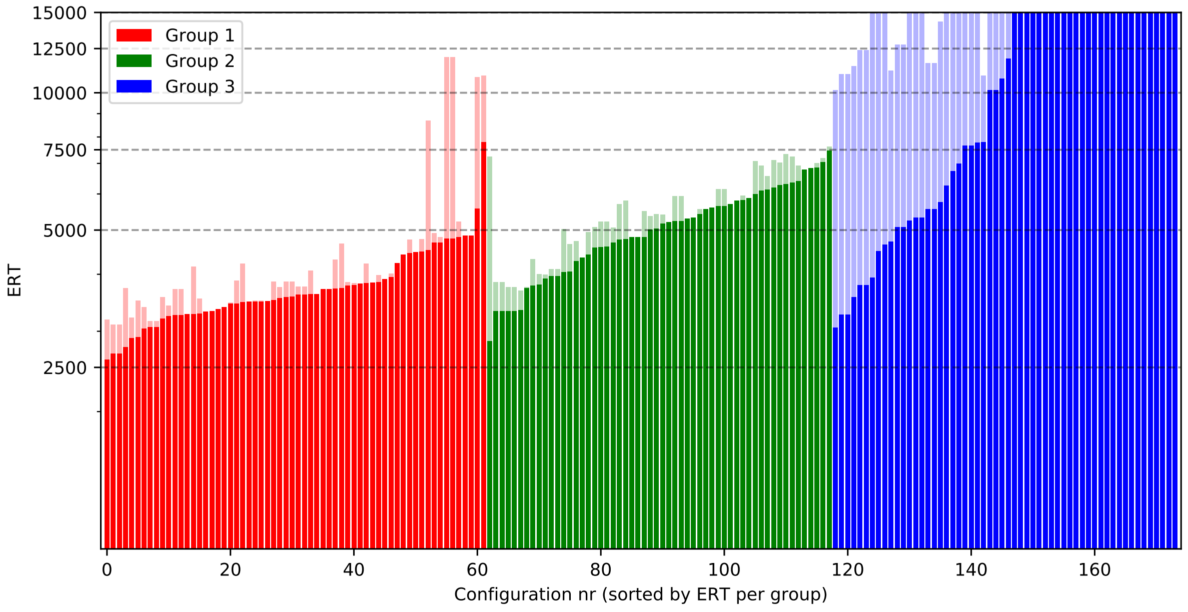


(b) Rerun for 50 runs on 5 instances, both for the default hyperparameter settings as well as the tuned ones. The configurations are sorted according to the ERT with tuned hyperparameters.

Figure 4: ERT of default hyperparameter settings vs. those found by MIP-EGO, for 56 different configurations on benchmark problem F21.



(a) F12: ERT of default hyperparameter settings (lighter colour) vs. those found by MIP-EGO (darker colour). Configurations sorted according to optimized ERT within their group. All data comes from 5 runs of 5 instances.



(b) Rerun of configurations and hyperparameters from Figure 5a, for 50 runs on 5 instances.

Figure 5: ERT of default hyperparameter settings vs. those found by MIP-EGO, for 174 different configurations on benchmark problem F12.

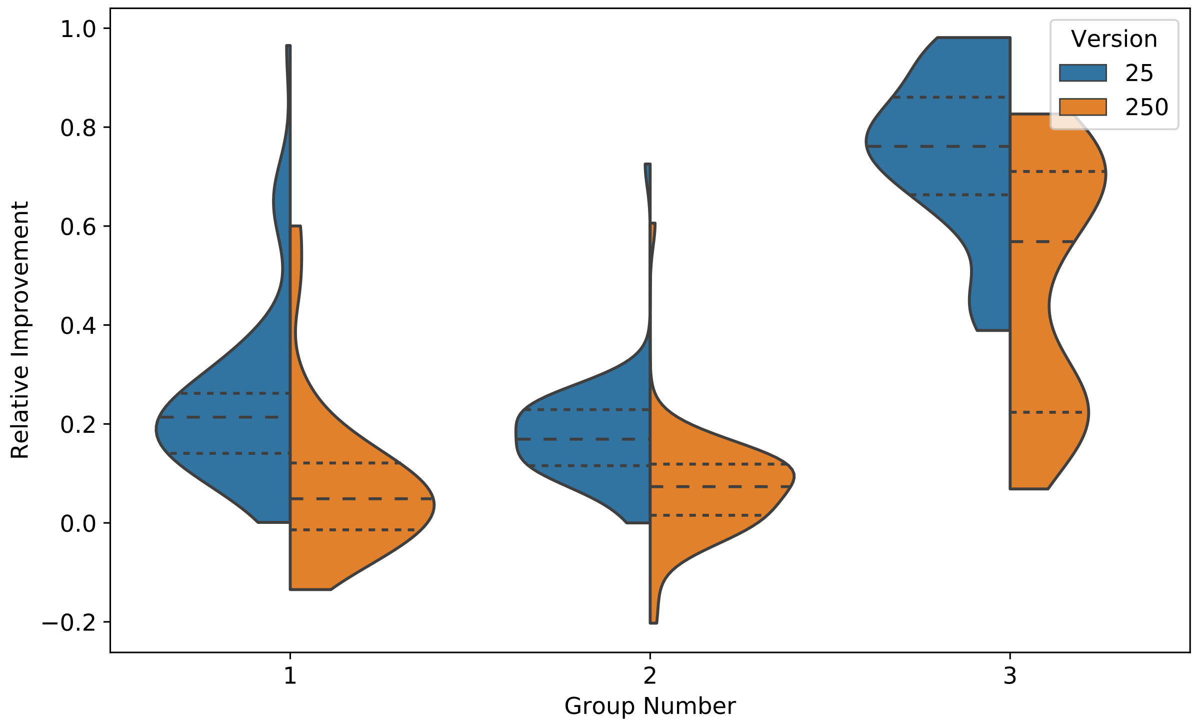


Figure 6: Distribution of relative improvements in ERT between default and tuned hyperparameters for 174 different configurations, split in 3 groups as described in the previous section. On benchmark function F12.

Mann-Whitney U test confirms that, with the exception of configuration $1{,}341$ there are no statistically significant differences between the two distributions for the configurations with see negative improvement ($\alpha = 0.01$). However, even for the configuration for which the distributions are different, the probability that a random sample of 5×5 runs from the distribution with the optimized hyperparameters has a lower ERT than another random sample from the distribution with default hyperparameters, is around 26.6%.

Another detail to notice in Figure 5b is that, while for most configurations, the large improvements in ERT become much smaller, there are several configurations for which the large improvements remain. This indicates that these configurations have much more to gain from hyperparameter tuning. Some investigation of the module activations in these configurations shows that they all have the active update module turned on. When this module is active, it changes the value of the c_c parameter to $\frac{2}{(D+\sqrt{2})^2} = 0.0486$, which seems to be too low for this function, thus there can be a large improvement when changing the value of this hyperparameter.

For most configurations on which hyperparameter tuning was performed, the improvement shown in Figures 5a and 4a is not stable over 250 runs. To get a better understanding of how this could happen, a small experiment was performed to simulate how a simplified hyperparameter optimization might work. For this experiment, we assume that for a given configuration around 10% of tested hyperparameter values have the same hitting time distribution as the optimal hyperparameter value found by MIP-EGO. If we then sample k runs on each instance from the set of 50 runs whose hitting times we have gathered, we end up with 20 sets of $5 \cdot k$ hitting times, from which we then take the one with the best ERT. We can then compare its

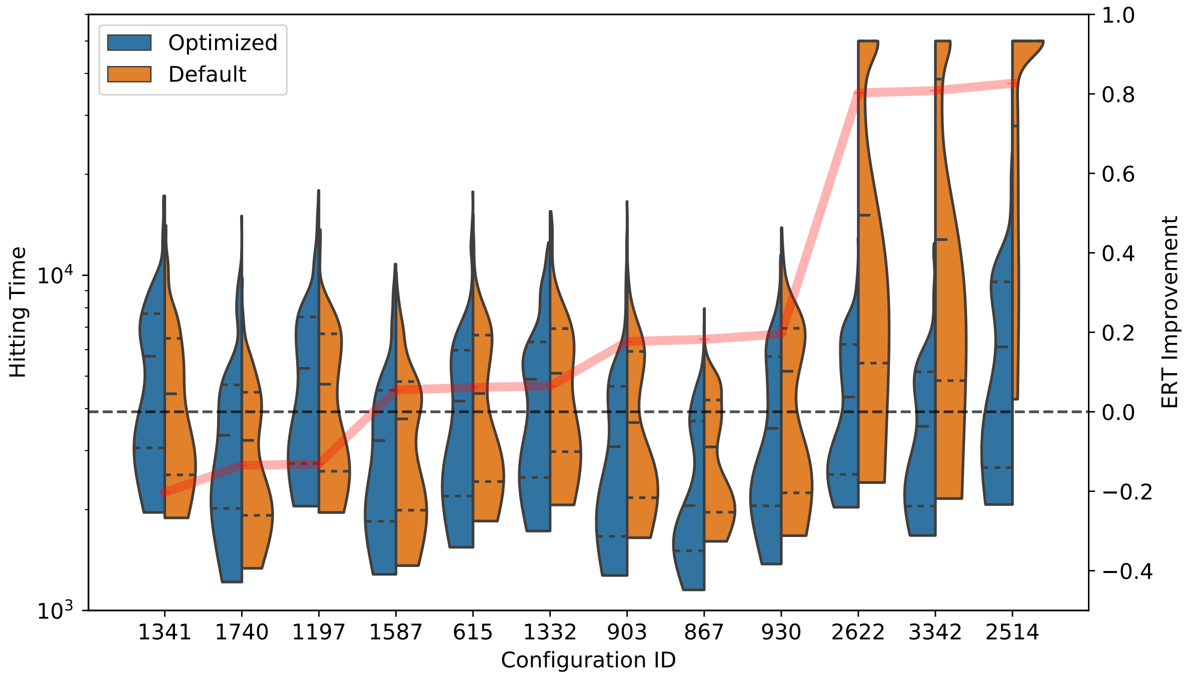


Figure 7: Distribution of hitting times (left y-axis) from 250 runs for 12 different configurations on F12, comparing the runs with optimized hyperparameter settings to those with default hyperparameters. The red line (right y-axis) indicates the amount of improvement in ERT observed for the configuration.

ERT to that of the original sample to see how much improvement we can find. The results from this sampling experiment are shown in Figure 8, as well as the actual improvement on F21 between the ERT found from the 25 runs observed during the running of MIP-EGO and the 250 verification runs performed afterwards.

From Figure 8, it can be see that the observed worsening of the ERTs of the configurations with tuned hyperparameters on F21 can be explained by this sampling. In other words, during the running of MIP-EGO, we might sample from many similar distributions, and because only only the best candidate is selected at the end, there is a large possibility of over-fitting on the used seeds. This indicates that care needs to be taken in interpreting the raw ERT results. This demonstrates the need to always repeat experiments on unseen seeds to verify generalizability of the selected hyperparameters. For F12, the difference between ERT from 25 runs during MIP-EGO and achieved ERT on 250 runs can be explained the same way. When assuming only 5% of tested hyperparameters have the same distribution of hitting times, an improvement of around 16% can be expected, which matches the observed 15.3%.

For a final measure of the impact of variance on performance, the differences in ERT-based ranking of the configurations is studied in more detail. For F21, the changes from the ranking based on the 25-run ERT and the ranking based on the ERT from the 250 verification runs is shown in Figure 9. In this figure, it can be seen that the changes in rankings are quite significant, which magnifies the importance of the robust verification procedure. To more precisely state the correlation between the two rankings, we can calculate the Kendall rank correlation coefficient (which we will refer to as the Kendall coefficient), which is defined as follows:

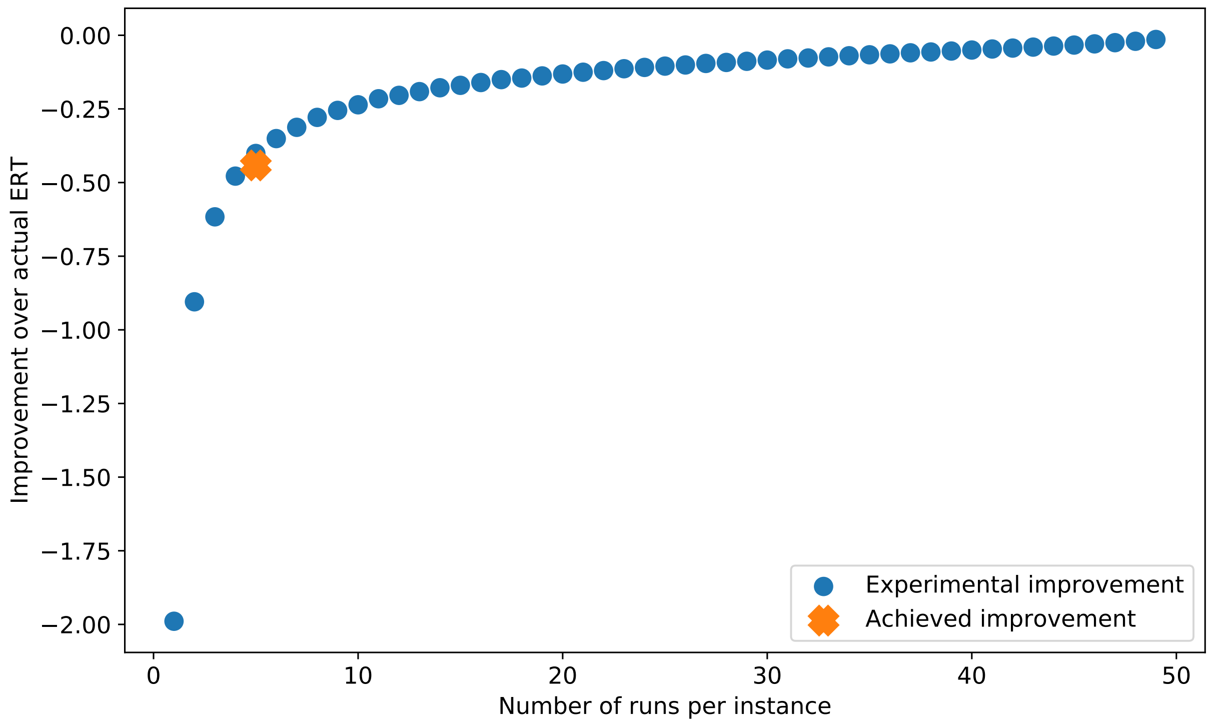


Figure 8: Average improvement of ERT obtained from 250 runs vs the value obtained after running MIP-EGO (25 runs) in orange, vs experimental improvement. Experimental improvement obtained over 100 repetitions of selecting k samples per instance for each configuration and calculating their respective ERT.

Definition 4.1. Kendall rank correlation coefficient The Kendall rank correlation coefficient between two ranks of data-points X is defined as follows:

$$K(r_1, r_2) = \frac{2 \cdot \sum_{x,y \in X} \operatorname{sign}(r_1(x) - r_1(y)) \cdot \operatorname{sign}(r_2(x) - r_2(y))}{|X| \cdot (|X| - 1)}$$

Where we set $\operatorname{sign}(0) = 0$ to deal with ties.

For the rankings in Figure 9, the Kendall coefficient is 0.40, which is quite low, indicating the the correlation is indeed not very strong. This leads to the finding that the algorithm selection itself might not be very robust. To make sure this effect is not solely based on the used function or the configurations, the same analysis is performed on benchmark function F12, with a larger set of configurations. This is done in Section 4.3.

In this section, the impact of variance on the hyperparameter tuning has been investigated. It has been shown that variance plays a big role in the hyperparameter tuning process, and thus care needs to be taken when selecting which configurations and which hyperparameters to use. Another important finding from this section is the fact that the configuration which performs the best with default hyperparameters does not necessarily correspond to the configuration which performs the best after hyperparameter tuning. The next section will take a closer look at the differences in ranking between ERTs of default and tuned hyperparameter values.

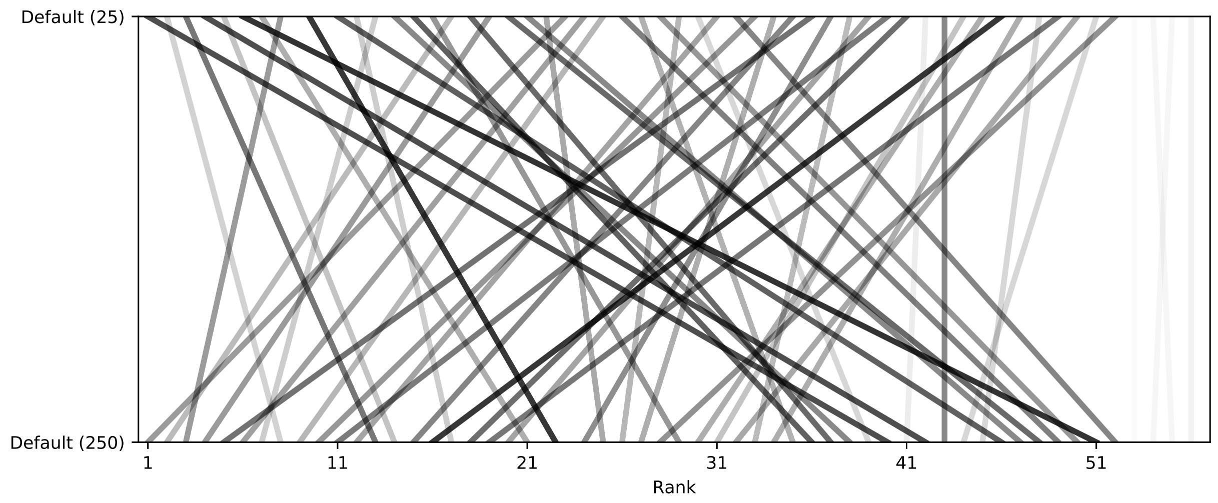


Figure 9: Evolution of ERT-based ranking (lower rank is better) of 56 configurations on F21. Default refers to the realized ERT of the default hyperparameters during the run of MIP-EGO (default hyperparameters are always included in the initial sampling of the search space). Optimized is the best ERT found by MIP-EGO. Darker lines correspond to larger changes in ranking.

4.3 Impact of hyperparameter tuning on configuration ranking

The previous section explored the effects of variance on the relative ERT improvements for either a single configurations or a set of configurations. If this effect would be constant, one might simply ignore it and use the 25 run data and apply the correction at the end. However, the impact of variance changes per configuration. Thus, when ranking the configurations by ERT, this ranking might differ depending on whether 25 or 250 runs were used. Visualizations of the changes in ERT-based ranking for F21 and F12 are shown in Figures 9 and 10 respectively.

As discussed previously, the changes in ranking seen in Figure 9 are quite large. This indicates that algorithm selection will not be very robust, so care should be taken when determining which configurations to perform hyperparameter tuning on. However, it should also be noted that the hyperparameter tuning itself might have an impact on the ranking of the configurations, This is shown in Figure 10, where several large differences in ranking between default and tuned hyperparameter settings are noticeable. These correspond to the previously mentioned configurations with active update, which improved significantly with hyperparameter tuning. Overall, the other configurations seem relatively stable. The Kendall coefficient for the rankings from Figure 10 is 0.70. This signifies a positive correlation, although it is not as high as initially expected (for comparison, the correlation between the ERT rankings with default hyperparameters for 25 and 250 runs is 0.89). This indicates that just selecting the top x configurations and tuning their hyperparameters might not be enough to find the best possible ERT.

In this section, the effects of hyperparameter tuning on the ERT-based ranking of configurations has been studied. This gives rise to the question of whether the default hyperparameter values are close to, or far from, optimal, and for which configurations this is the case. To get an idea of how different the tuned hyperparameters are from the default ones, the next section investigates the distribution of the hyperparameter values

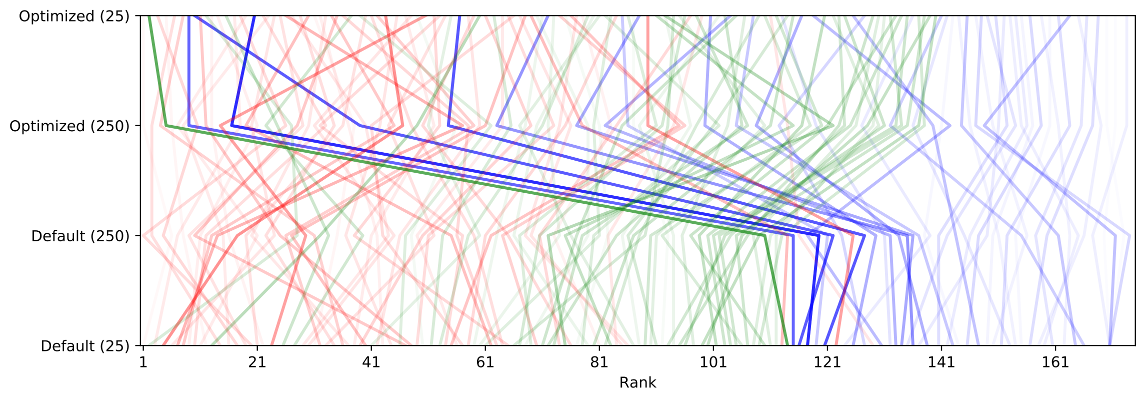


Figure 10: Evolution of ERT-based ranking (lower rank is better) of 174 configurations on F12. Default refers to the ERT using the default hyperparameters while optimized is the best ERT using the tuned hyperparameters as found by MIP-EGO. Color-scheme as in Figure 5a. Darker lines correspond to larger changes in ranking.

4.4 Distribution of parameter values

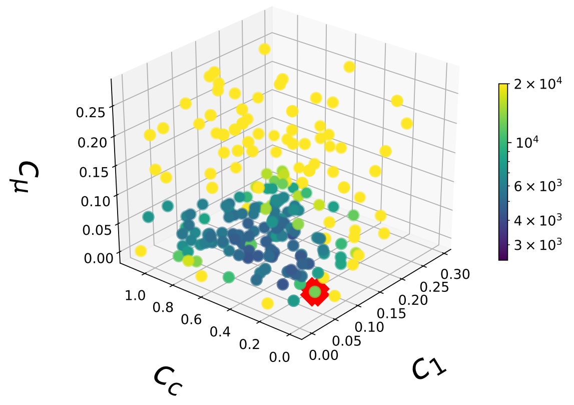
Next, we will take a more in-depth look at the actual values of the tuned hyperparameters. Since we only optimize three hyperparameters, the entire hyperparameter space can be visualized in a 3D-plot. First, we show the distribution of the ERT for all hyperparameter settings tested during the MIP-EGO run (200 points), for two separate configurations. This is done in Figure 11. From this figure, we notice that, while the differences between the two configurations are quite large, they do have an area in common where the ERTs are quite close to the optimal one found by MIP-EGO. We also notice that the default hyperparameter configuration does not necessarily lie in this area.

To get a clearer picture of this common region of interest, we perform the following procedure to create the visualization shown in Figure 12:

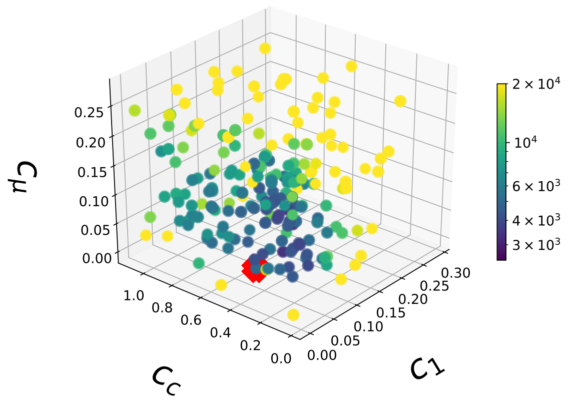
- Create a grid of points evenly split among the parameter space
- For each point and each configuration, add the mean value of its 3 nearest-neighbor points which were evaluated
- Divide by the number of configurations considered to get an average nearest-neighbor value among all configurations

This gives us a visual representation of the common region of interest where the ERT is lowest among all configurations. To prevent a large influence of poorly-performing configurations, we only take into account the top 50 configurations (ranked based on the best ERT achieved during the MIP-EGO run).

From Figure 12, we can see that even thought the optimal hyperparameter values are quite spread out, there still is a clear area of common good performance. It is important to note that most default hyperparameter settings do in fact lie within this common area, indicating that they are quite well chosen already. However, this also stresses the importance of per-configuration hyperparameter tuning, since the optimal ERT might be achieved with a very different hyperparameter setting.



(a) Configuration nr. 4,374



(b) Configuration nr. 918

Figure 11: Distribution of ERT-values for the hyperparameter settings evaluated during the MIP-EGO runs of two configurations. Darker colors correspond to lower ERT. The default hyperparameter setting is marked in red.

This experiment showed that the optimal hyperparameter setting can vary largely between different configurations. For some configurations, the default hyperparameter settings might be close to optimal, leading to a small potential gain in ERT from hyperparameter tuning. However, some other configurations have poorly performing default hyperparameter settings, and can see relatively large performance improvements. This gives rise to a natural question: can the performance of a configuration after hyperparameter tuning be predicted based solely on the configuration itself?

4.5 ERT-prediction using random forests

For the previous experiments, we focused on the hyperparameter tuning and its results on the performance of individual configurations. However, this was done using a brute-force approach, where we gathered 5×5 run data. In total, the sequential approach using MIP-EGO requires $25 \cdot 4,608 + 200 \cdot 25 = 120,200$ total function evaluations. And, as we saw in Figures 10 and 9, simply choosing the top x configurations based on ERT with default hyperparameters might not always suffice to find the configuration with the best ERT after hyperparameter tuning. So even tough a huge number of function evaluations have been used, there is no guarantee that the best (configuration, hyperparameters)-pair will be found.

In this section, we explore a way to predict the ERT, both with default and optimized hyperparameters, without having to run MIP-EGO on each configuration. This should allow for a limited number of MIP-EGO runs to gather data, after which we have a single configuration for which the hyperparameters can be tuned. This can be achieved by training a Random Forest (RF) model on the 250-run data for a subset of configurations, and using this to predict the ERTs for the unseen ones.

To determine the viability of this method, we first try to predict the ERT of some configurations on F12 for which we have already gathered the necessary data to verify the predictions. We use

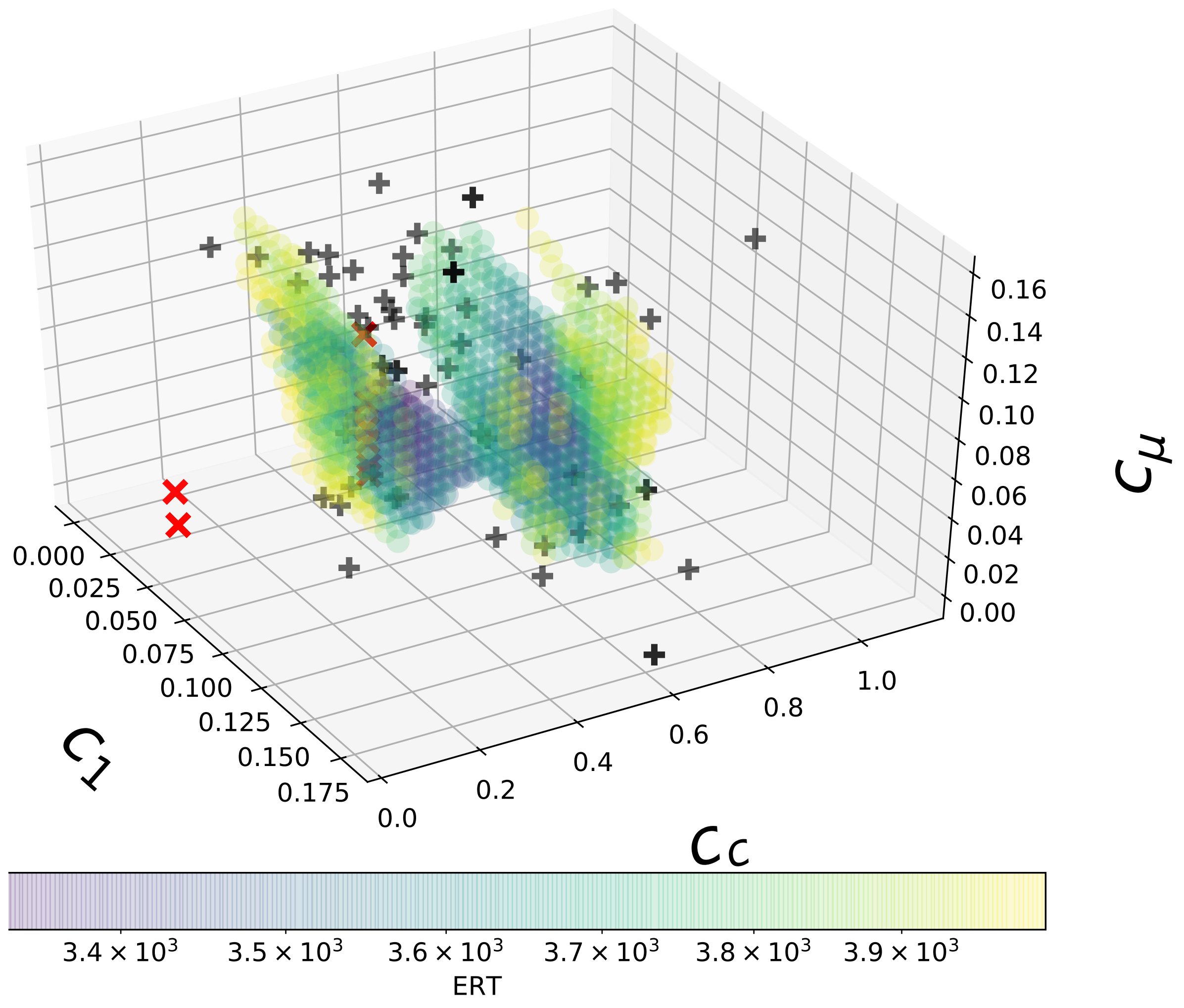


Figure 12: Aggregated ERT among 50 best configurations: averaged nearest-neighbor ERT value among top 50 configurations, gathered by splitting the search-space into a grid and for each grid point averaging among all configurations the average ERT of the grid points 3 nearest neighbors evaluated during the MIP-EGO run. Red points indicate locations of default hyperparameters, while black '+'-sings indicate locations of optimized hyperparameters found during MIP-EGO runs. Only points with averaged nearest-neighbor ERT of less than 4000 are shown.

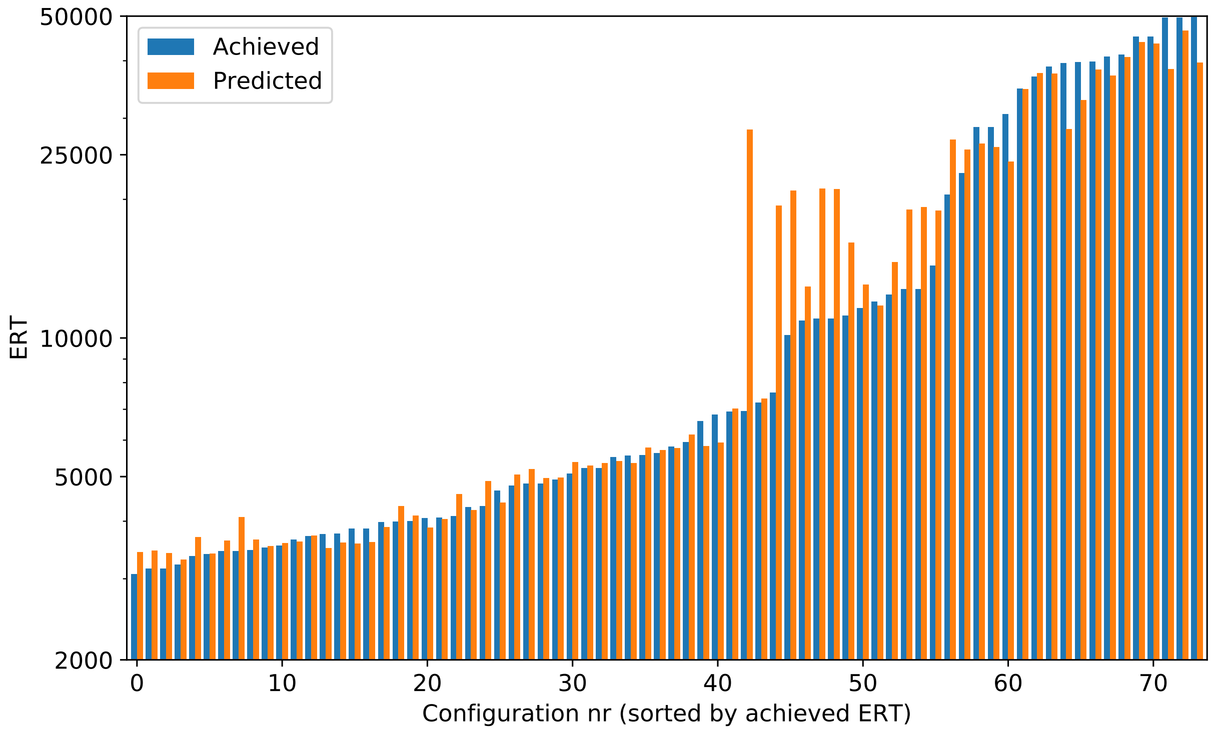


Figure 13: ERT predicted by random forest model vs achieved ERT (250 runs), for a testset of 74 different configurations.

the configurations mentioned in the beginning of this section, and split them up in a training-set of size 100 and a test-set with the remaining configurations, with the goal of predicting the ERT with default hyperparameters. In Figure 13, we show the differences between predicted and achieved ERT, which can be shown in terms of prediction error, which is defined as follows:

Definition 4.2 (Prediction error)**.**

$$\text{PredictionError}(X) = 1 - \frac{\min(X_{\text{pred}}, X_{\text{real}})}{\max(X_{\text{pred}}, X_{\text{real}})}$$

From Figure 13, we see that the fit between predicted and achieved is quite good, with an average error of close to 10%. We can also look at which configurations the model predict to perform well, and run these to verify the results. We take the 11 configurations which the model predicted to perform well, and show the results of running them in Figure 14. From this, we see that there are a few outliers present for which the model is completely wrong, but for all other configurations the prediction is as accurate if not more so than the ERT predicted by looking at the 25-run data.

Since this is only one realization of the model, it might not be representative of the actual predictive power. To get some more robust results, we need to repeat these experiments. We are also interested in knowing how many samples are needed to train the model well, so we perform a test with varying training-set size. We compare the prediction of the ERT with default hyperparameters to the ERT with optimized hyperparameters, as well as a baseline of random normally distributed values with the same mean an variance as the ERTs with default hyperparameters. The results of this experiment are shown in Figure 15. From this, we see

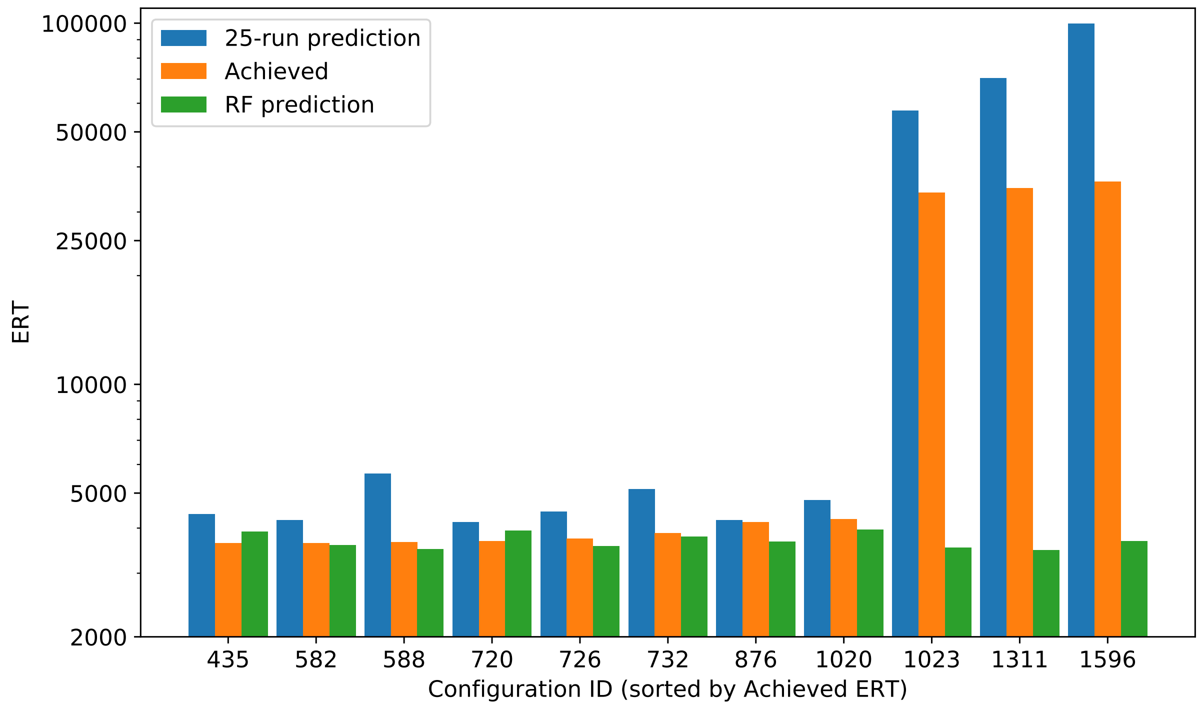


Figure 14: ERT predicted by random forest model vs achieved ERT (250 runs), for the set of the 11 best predicted configurations.

that predicting the ERTs of the default hyperparameter values is slightly easier than predicting the ERT after hyperparameter optimization. However, both predictions are substantially better than random, indicating that the model is at least learning something about the configurations.

Even tough this approach seems promising, we have to keep in mind that the configurations we use to train and test on are not chosen randomly. This bias in the available configurations might be a large factor in explaining the predictive strength of our model. To verify this, we perform a similar experiment on F6 by gathering ERT data (default hyperparameters) for 112 random configurations[1]. We split this in a testset of size 90, and test on the remaining configurations. When repeated $1{,}000$ times, we get an average difference between predicted and actual ERT of 61.2%, indicating that way we choose the configurations is indeed the major reason why we saw good predictions for F12.

4.6 MIP-EGO vs irace

For all of our previous experiments, we used MIP-EGO for the hyperparameter tuning. While this method is very well suited to expensive optimization and by extension hyperparameter tuning, it is not adapted to deal with noisy problems. For this reason, we compare it to irace, which as described in Section 2.5.4, should be able to handle noisy data more effectively. Since we ran MIP-EGO for 200 evaluations, which consist of 25 runs each, we give irace a total evaluation budget of $5{,}000$ to split among its evaluations, and task it with minimizing the penalized averaged hitting time, with a penalty for unfinished runs of $4 \cdot B = 100{,}000$. We then run irace on 5 different configurations. We used the configurations with rank 0, 1, 2,

[1]Chosen as a multiple of 14 since 14 nodes were available to run this experiment

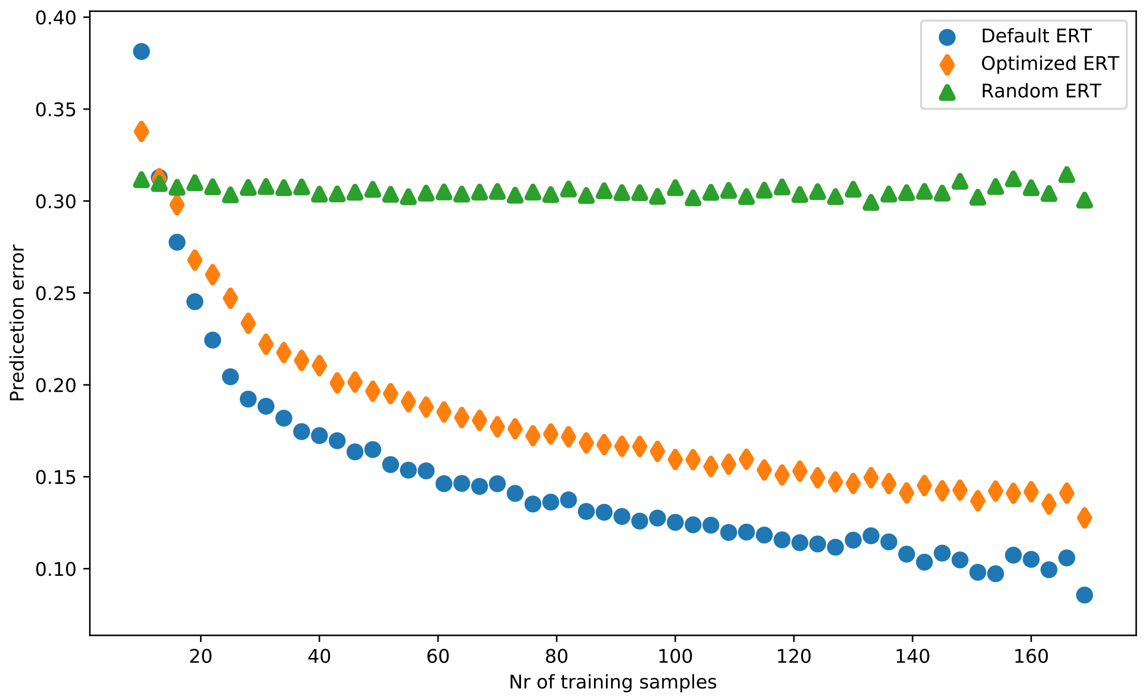


Figure 15: Absolute relative difference between predicted and actual ERT of Random Forest model trained on different training sizes (out of a set of 174 samples). Results are shown for benchmark function 12 for predicting ERT values (based on 250 runs) of default and tuned configurations. Random ERT refers to random samples from a normal distribution with the same mean and variance as the Default ERT samples.

25 and 50 based on ERT of 250 runs with MIP-EGO optimized hyperparameters. The results from running irace on these configurations is shown in Figure 16.

From Figure 16, we can see that the difference between the two methods seems negligible. If we keep in mind the biased way we used to select which configurations to use, we can say that irace performs just as well as MIP-EGO on these configurations for this function. An interesting point to note is the fact that the underestimation of ERT during the optimization also happens in irace, although slightly less than MIP-EGO. This effect would likely become smaller the more budget irace is given, since it would allocate more runs to the best hyperparameter settings to get a more robust ERT. When using MIP-EGO, the only way to achieve a better prediction would be to allocate more runs per iteration, which would mean that poorly performing hyperparameter settings will also take up a lot more evaluations.

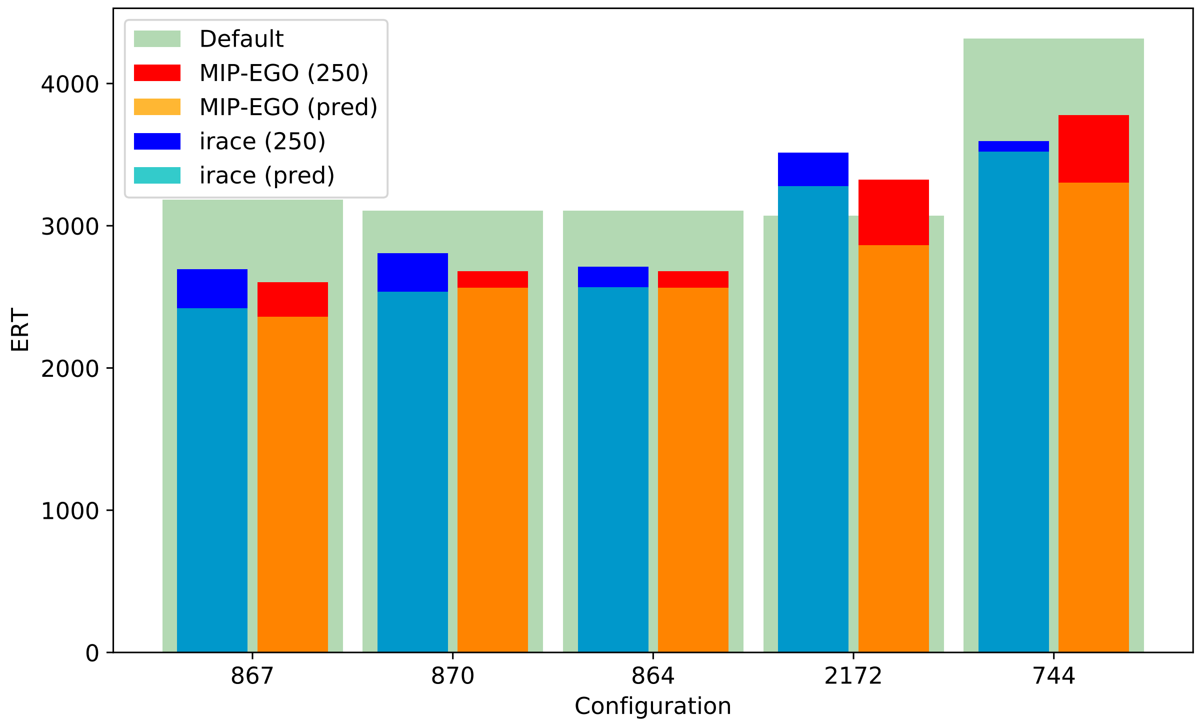


Figure 16: Comparison of irace and MIP-EGO on hyperparameter optimization of 5 configurations on F12. In the legend, 'pred' refers to the ERT found during the optimization run, while all other ERTs are based on 250 runs.

4.7 Impact of hyperparameter tuning per function

To get some more insight into the differences between the available functions, a smaller version of the experiment described in Section 4 is performed for all functions. A set of 30 configurations is selected for each function. These configurations can be split into three distinct groups:

- Group 1: The 10 configurations with the best ERT on the 25-run data
- Group 2: The configurations ranked 200-210 based on the 25-run ERT
- Group 3: The 10 commonly used configurations as seen in Table 2.

Note that groups 1 and 2 are distinct for each function, as opposed to group 3 which is static. For each of these configurations, we run MIP-EGO to optimize (c_1, c_c, c_μ) using a budget of 200 evaluations of 25 runs each. The resulting (configuration, hyperparameters)-pairs are then run it 50 times on 5 instances. This is done both with default and tuned hyperparameters. From this, the average gain from hyperparameter tuning can be calculated. This is shown in Figure 17.

In Figure 17, the differences between functions are clearly visible. For some of the easier functions, the relative improvements are all close together, indicating that the tuning of hyperparameters has a similar effect on most configurations. However, for most of the functions, this is not the case, as the differences in relative improvement span a much wider range. This is consistent with our previous finding on F12, where we saw that for some configurations,

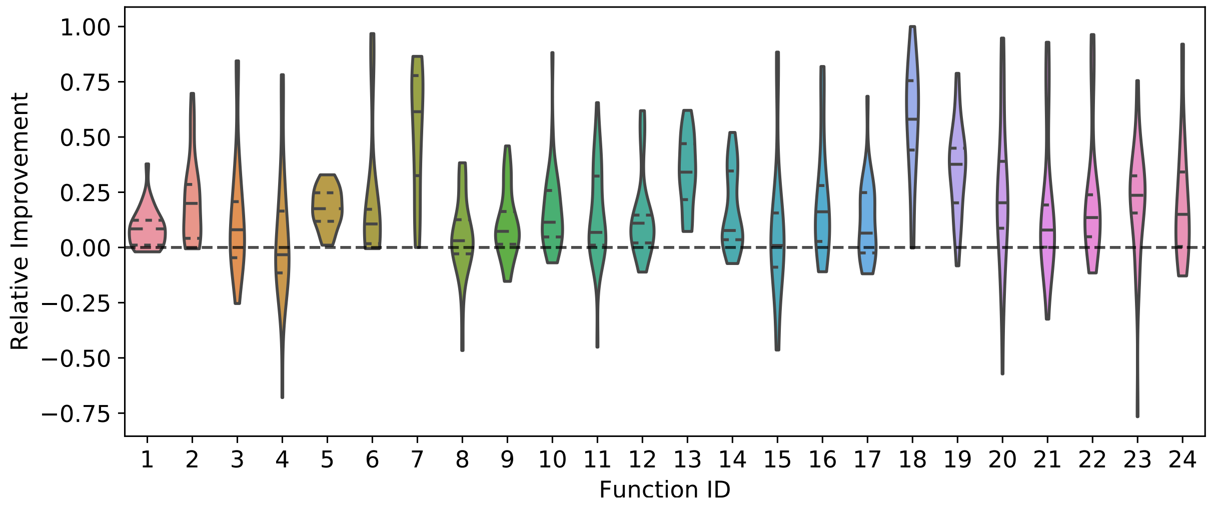


Figure 17: Distribution of relative improvement in ERT between the default and tuned hyperparameters. For each function, 30 configurations (as described in the first paragraph of Section 4.7) are tuned with MIP-EGO, and the resulting (configuration, hyperparameters)-pairs are rerun 250 times to validate the results. The same is done for the default hyperparameters, and then the relative improvement in ERT is calculated.

there was a large improvement possible over the default hyperparameters, while for others the default hyperparameters are close to optimal. There are also some negative improvements shown, which might be caused by a large prediction error.

Using the data gathered in this experiment, the prediction error for MIP-EGO can be studied in more detail. For each function, 30 configurations are available on which MIP-EGO was run. This gives us 30 prediction errors per function. Based on the experiments in Section 4.2, it can be assumed that larger variance will lead to larger prediction errors. Figure 18 shows both the distribution of prediction errors as well as the average standard deviation of hitting times over all 30 configurations. In this figure, the relation between the prediction error and the variance is clearly visible.

From this experiment, we can also extract the overall amount of improvement gained per function, in terms of the best tuned configuration relative to the best configuration with default hyperparameters. This can also be compared to the results of the ‘naïve’ approach of sequential algorithm selection and configuration, i.e. tuning the hyperparameters of the best configuration. These comparisons are shown in Figure 19. From this, we see that, while the default hyperparameter tuning manages to find some improvement over the default hyperparameter settings, the best (configuration, hyperparameters)-pair manages to find a significantly larger improvement.

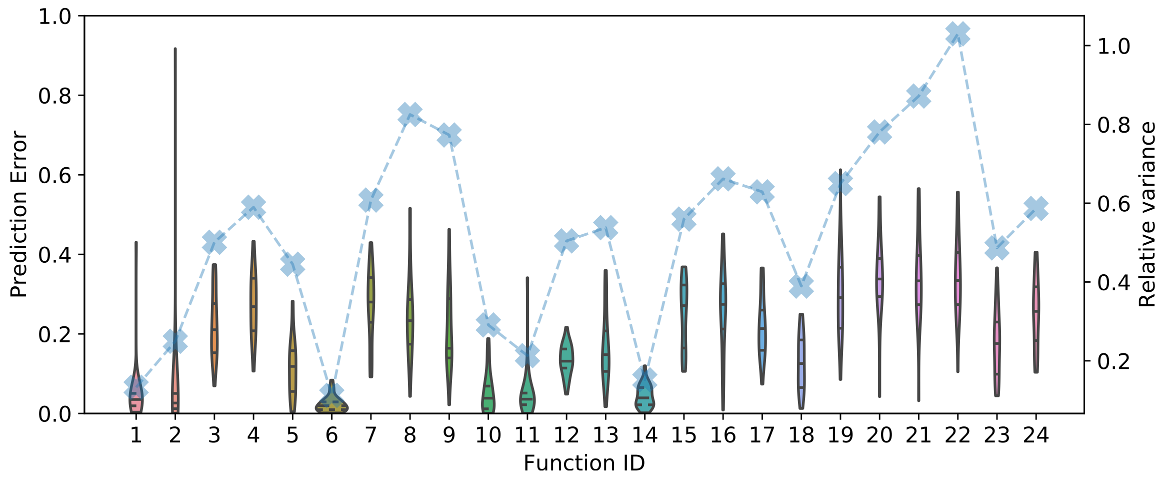


Figure 18: Comparison of the distribution of prediction errors (between the ERT found during the MIP-EGO run and the ERT found during the 250 validation runs) and the relative variance, i.e. the average standard deviation divided by the average AHT among the 30 configurations used (as described in the first paragraph of Section 4.7).

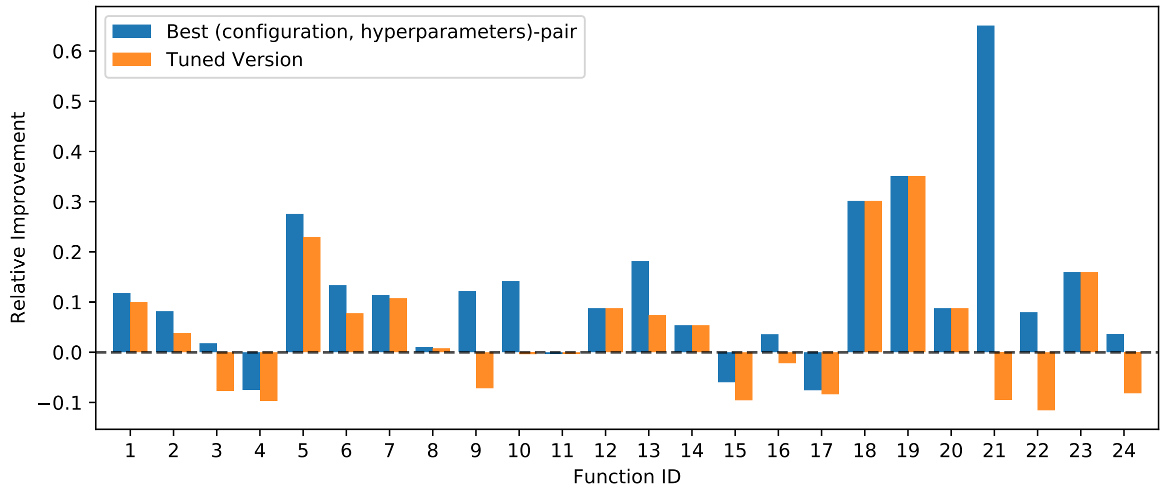


Figure 19: Relative improvement in ERT over the best configuration with default hyperparameters, both for the best (configuration, hyperparameters)-pair found (denoted by 'Best Pair') as for the tuned version of the selected configuration (denoted by 'Tuned version'). All ERTs are from 250 runs. Negative improvement are possible, since the tuned hyperparameters settings for each configuration are chosen based on 25 runs and could be biased, as described in Section 4.2.

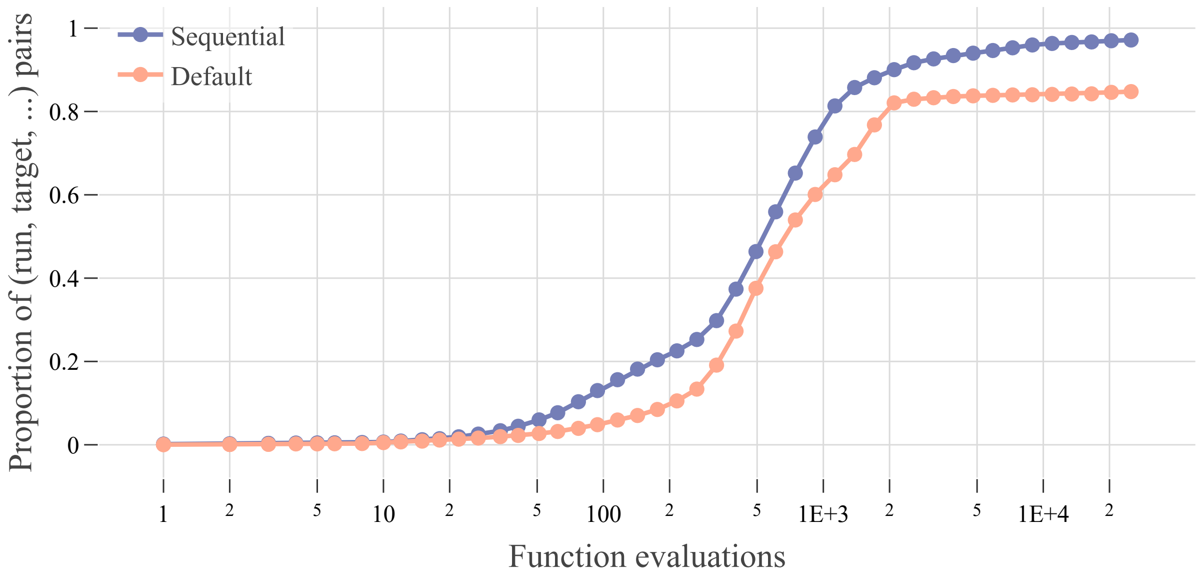


Figure 20: Empirical Cumulative Distribution Function (ECDF) of the default CMA-ES and the sequential approach as described in Section 4.7. Results are based on 250 runs. This visualization was generated by the IOHprofiler [DWY+18]. The ECDF shows the average fraction of (run, target)-pairs which were reached within x function evaluations by the respective algorithms.

4.8 Summary

In this chapter, the focus has been on the effects of hyperparameter tuning on the performance of different configurations. The need for hyperparameter tuning per configuration has been shown in Section 4.4. This emphasizes the need for a good algorithm selection technique, since Section 4.2 proved that the impact of variance is a large factor in the effectiveness of hyperparameter tuning. This causes configurations which perform well with default parameters to not necessarily correspond to the configurations which will perform well after the hyperparameter tuning.

Because of these differences in ranking, a model for ERT-prediction of a configuration was proposed. This RF model is trained to predict the ERT of a configuration with either default or tuned hyperparameters, after which the configuration with the best predicted ERT is selected to be run. We found this approach to be unreliable when using a limited number of training samples, and did not pursue this angle further. Instead, the focus was shifted towards a different method of algorithm selection. We selected 3 groups of configurations for each function, an ran hyperparameter tuning using MIP-EGO on all of them. The results from this approach, which are again visualized in Figure 20, are much more promising than only selecting the best configuration, but require many more function evaluations. Because of this, the next chapter discusses methods to integrate algorithm selection and configuration into a single approach, removing the reliance on available hitting time data.

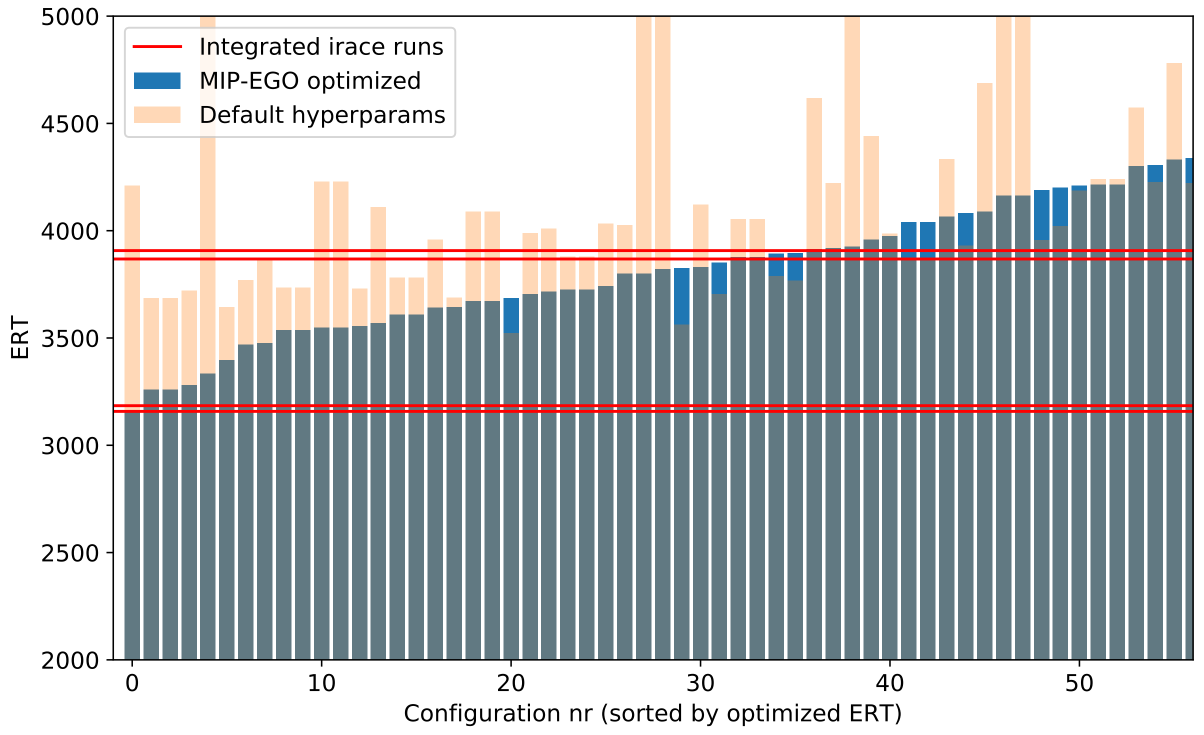


Figure 21: ERT (250 runs) of configurations with default and optimized hyperparameters, compared to the ERT of the (configuration,hyperparameters)-pairs found by irace, on instance 1 of F12.

5 Integrated approaches

In Section 4, the focus has been on optimizing hyperparameters and choosing configurations for which to optimize them, but we have kept a clear separation between the two parts. However, if we view the configuration itself as a hyperparameter, we can optimize it at the same time as. We call this the one-search-space approach. Since both MIP-EGO and irace support discrete or nominal parameters, we will compare these methods to each other and to the previous approach of splitting the configurations selection and hyperparameter optimization into two steps.

5.1 Exploration of procedures

To start with, we run irace on just a single instance of F12 and compare it to the results we got from MIP-EGO in Section 4.2. We give irace a budget of $25{,}000$ total runs. We repeat this experiment four times, and show the results in Figure 21. From this figure, we see that the results look promising. While two runs performed slightly worse than some of the configurations even with default hyperparameters, the other two runs achieve an ERT similar to that of the best one found using MIP-EGO on each configuration. The fact that we get different configurations (with IDs 867, $2{,}191$, 869 and $1{,}155$) for each of these runs might indicate that irace converges to a single configuration quite quickly, while it could still benefit from some more exploration.

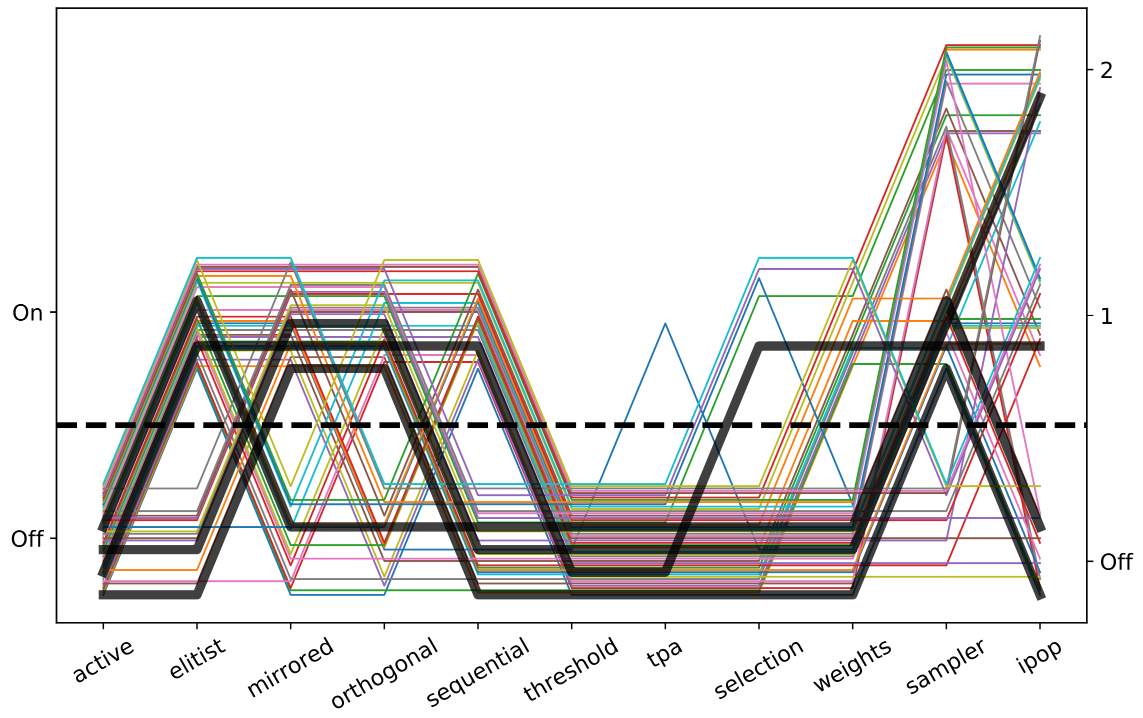


Figure 22: Combined module activation plot for F12 instance 1, with the configurations found by one-search-space irace overlaid in black.

If we take a closer look at which configurations were found by irace, we can compare these to the configurations which perform well with static hyperparameters. To visualize this comparison, we use the combined module activation plots. This plot represents each configuration as a single line, connecting the status of their modules. The best configuration is plotted at the bottom, with further configurations added on top until we have the set of 50 best performing ones. In Figure 22, we then overlay the configurations found by irace in black. We see that the found configurations seem to correspond to the overall structure of the best configurations, with TPA, active update and threshold always being off, which seems to be an important factor in the performance on F12, as 49 out of the best 50 configurations with default hyperparameters have these same module settings.

While Figure 21 only used a single instance, this is not a requirement of irace. We can repeat this experiment for all 5 instances. This is shown in Figure 23, where we have 2 runs of irace on 5 instance of F12, with a budget of $25,000$ runs each. We notice a similar pattern as in Figure 21, in that we get one run which is competitive with the best MIP-EGO result and one which still manages to outperform most configurations with default hyperparameter values.

5.2 Methods

In this section, we compare four different methods for the integrated algorithm selection and configuration approach:

- **Naïve Sequential approach**: We first select which configuration to use based on the available 25-run data for all static configurations, and then run MIP-EGO to optimize the hyperparameters of this configuration.

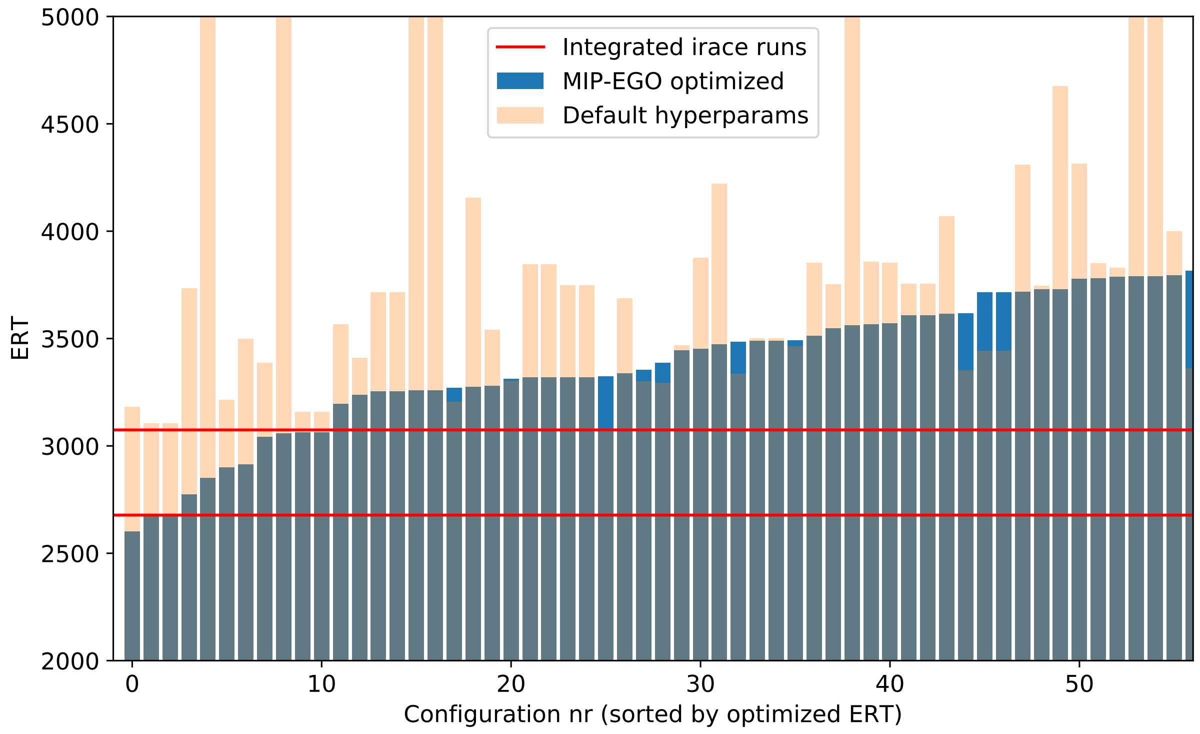


Figure 23: ERT (250 runs) of configurations with default and optimized hyperparameters, compared to the ERT of the *(configuration, hyperparameters)*-pairs found by irace, on 5 instances of F12.

- **Sequential approach**: As described in Section 4.7, a set of configurations is tuned using MIP-EGO, and the best resulting (configuration, hyperparameters)-pair is chosen for each function.
- **MIP-EGO**: We run MIP-EGO on the mixed-integer search-space to select the best *(configuration, hyperparameters)*-pair.
- **Irace**: As per the previous section, we use irace to find the best *(configuration, hyperparameters)*-pair.

5.3 Baseline

Before running the methods for integrated algorithm selection and configuration on all functions (using MIP-EGO and irace), a baseline for the performance of the sequential approaches needs to be established. This is done using two slightly different methods, both based on complete enumeration for the algorithm selection and MIP-EGO for the hyperparameter tuning. This is done by selecting the best static configuration based on the available 25-run data, and running MIP-EGO on these configurations. Either the best of these configurations, or a set of them (as in Section 4.7) is considered. Previously, the ERTs from these configurations with tuned hyperparameters were compared to the ERTs of the best configuration without hyperparameter tuning. This was shown in Figure 19. From this figure, it can be see that for most functions, the achieved improvement is relatively small, especially for the approach which only tunes hyperparameters for one configuration per function. As we discussed in Section 4.2, this is likely caused by the low amount of samples on which the hyperparameter-tuning is based.

We investigate this in more detail in Section 5.4. The total amount of runs used by these approach is $4{,}608 \cdot 25 + 200 \cdot 25 = 120{,}200$ for the variant which tunes hyperparameters only for the best configuration, and $4{,}608 \cdot 25 + 200 \cdot 25 \cdot 30 = 150{,}000$ for the other variant.

5.4 Comparison between MIP-EGO and irace

We now run MIP-EGO to perform the integrated optimization on all functions. We use a budget of $25{,}000$ total runs, equating to $1{,}000$ evaluations of (5×5) runs each, almost a factor of 5 lower than the naïve sequential approach. We then compare the resulting ERTs (on 250 runs) found during the optimization runs to the baseline set by the sequential approaches. The results from this comparison are visualized in Figure 24. From this, we can see that, for 17 out of 24 functions, the ERTs achieved by MIP-EGO are better than those of the best static configuration with tuned hyperparameters. On average, the amount of improvement is 15% over all functions. When compared to the more robust baseline, the improvement is present on only 10 functions.

The same experiment is performed using irace to see which method would be most suitable to the selection of *(configuration, hyperparameters)*-pairs. We use a budget of $25{,}000$ runs, to allow for a fair comparison to MIP-EGO. Since irace allocates these runs dynamically, we would expect the predictions of ERT to be more accurate than MIP-EGO. However, since irace uses the penalized average hitting time, the results for some more difficult functions might be skewed when comparing to ERT, since AHT is not a consistent estimator for the mean of the true hitting times when some runs do not manage to reach the specified target. As before, to be able to fairly compare to the other methods, we rerun the final selected configuration for 250 runs. The comparison to the baseline set by the naïve sequential method is shown in Figure 24. For irace, we see improvement over the baseline on 20 out of 24 functions, with the amount of improvement averaging to 14% over all functions. When compared to the more robust baseline, the improvement is present on only 12 functions.
As mentioned before, the comparison of ERTs from MIP-EGO, irace and the sequential methods is visualized in Figure 24. This figure shows that, in general, the ERT achieved by irace and MIP-EGO is comparable. Irace has a slight advantage, beating MIP-EGO on 14 out of 24 functions. However, both methods still manage to outperform the naïve sequential approach while using significantly fewer runs, and are only slightly worse than the more robust version of the sequential approach.

From Figure 24, we also notice that the prediction error for irace seems to be much lower than that of MIP-EGO or the sequential selection. We compare the prediction error, as defined in Definition 4.2, between the three methods we used, as well as the prediction error when only selecting the static configuration without hyperparameter tuning, in Figure 25. This figure makes the differences we noted previously much more clear. It shows that irace, even tough it uses penalized AHT instead of ERT, gives much more robust results than the other methods. The cases where the irace error is largest correspond to the functions where fewer runs managed to reach the specified target, indicating that it might be made more accurate if we modify the penalty used in irace to be consistent with ERT. Figure 25 also highlights the extremely large prediction error when just selecting the best static configuration from the complete enumeration, which was to be expected, as we use only 25 runs to select the best configuration out of a set of size $4{,}608$. This emphasizes the importance of the larger sample-sizes used in the verification, and the need for a reliable way to select the best configuration.

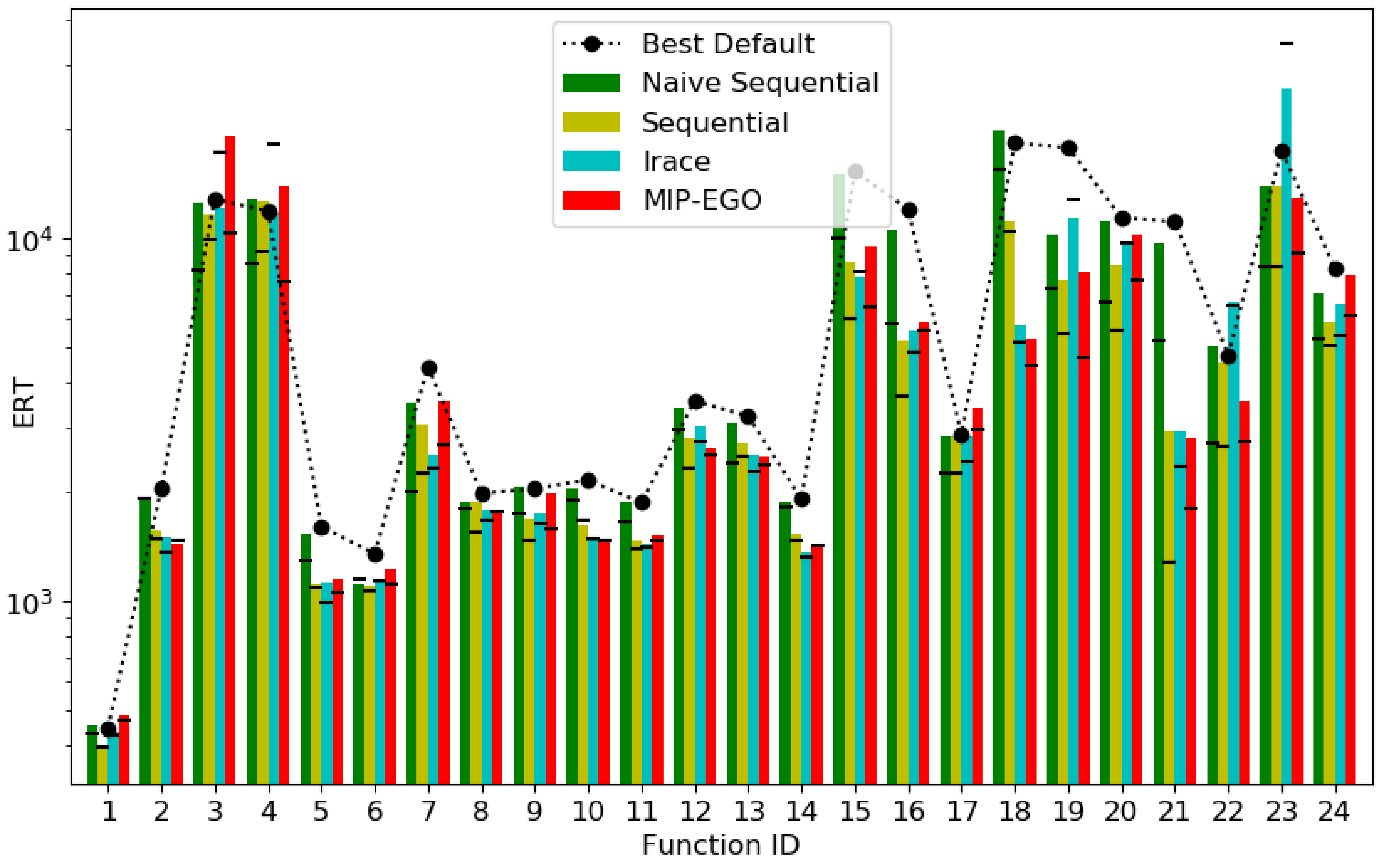


Figure 24: Resulting ERT (targets chosen as in [VvRBD19] and shown in Table 5) from running MIP-EGO and irace on the one-search-space, as well as the two sequential approaches described in Section 5.2. Configurations are selected based on 25 runs for MIP-EGO (both the sequential and integrated variants), or a variable number of runs for irace. The ‘predicted’ ERT based on these runs is shown as a small black bar, whereas all other shown ERTs are based on 250 verification runs. The dotted line represents the ERT of the best configuration with default hyperparameters. Precise ERT-values are shown in the Appendix, Table 5.

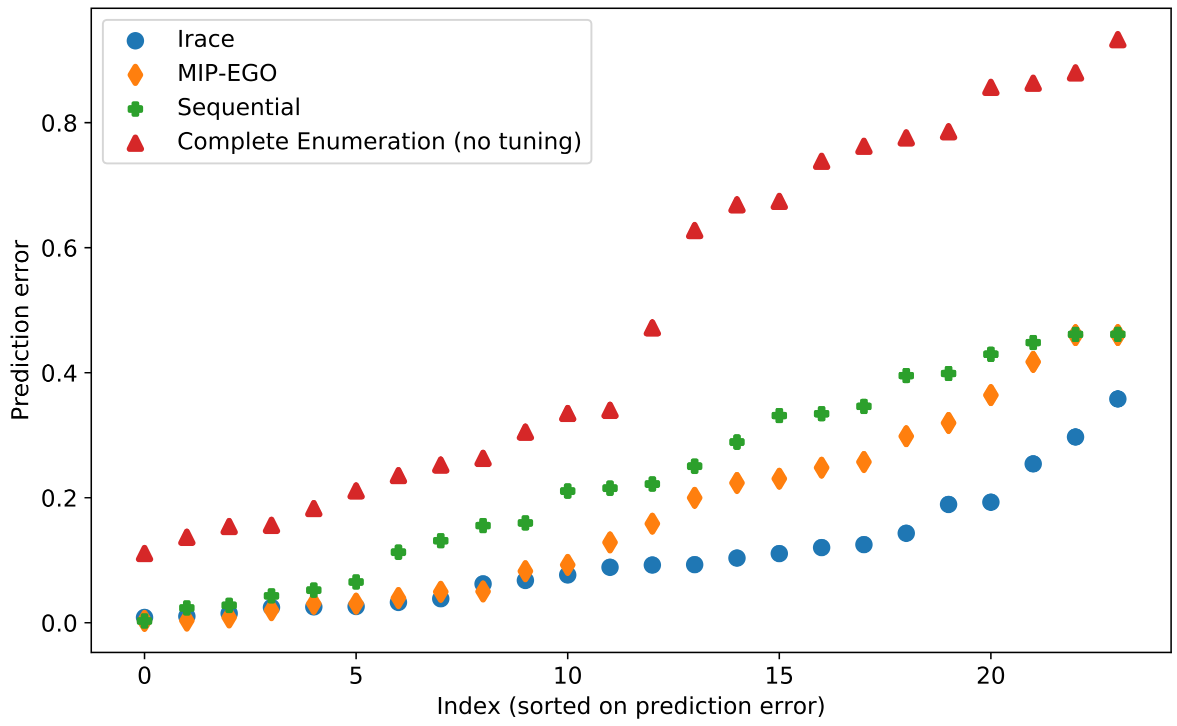


Figure 25: Prediction errors (see Definition 4.2) between ERT found during the run (penalized AHT in case of irace) and the ERT of 250 runs for MIP-EGO, irace and sequential one-search-space methods, as well as the prediction error when selecting the best configuration using complete enumeration (and not running hyperparameter tuning).

Overall, both irace and MIP-EGO give quite promising results. Both are able to find *(configuration, hyperparameters)*-pairs which outperform the static configuration found using sequential with tuned hyperparameters for most functions. This indicates that the one-search-space approach could be a useful technique for combining algorithm selection and algorithm configuration in the same procedure. Since both MIP-EGO and irace are very different techniques, it might make sense to combine the best parts from both of them into a procedure more suited to deal with the stochastic, non-normally distributed nature of the problem.

Since the initial experiments on F12 showed that running irace resulted in 4 different configuration when run 4 times, we want to investigate whether irace manages to maintain enough population diversity, or if it spends the majority of its budget of exploitation of a single or very few configurations. Ideally, if the population diversity gets too low, a restart would be used to remedy this. However, irace only restarts if the distance between the candidates is 0, i.e. they are exactly the same. With three continuous variables, this is unlikely to happen, as irace will spend more time evaluating the tiny changes in these parameter values.

To verify this assumption, we record how many different configurations are explored after the initial race, and compare this to the amount of different configurations explored by MIP-EGO. In Figure 26, it is shown that irace evaluates significantly fewer distinct configurations than MIP-EGO. When comparing this to the total amount of *(configuration, hyperparameters)*-pairs that were evaluated by irace, we see that, on average, only **2.6%** of these pairs contain distinct configurations. All others are solely different based on the hyperparameters, with on average **78.6%** of these pairs containing the same configuration. This indicates that the

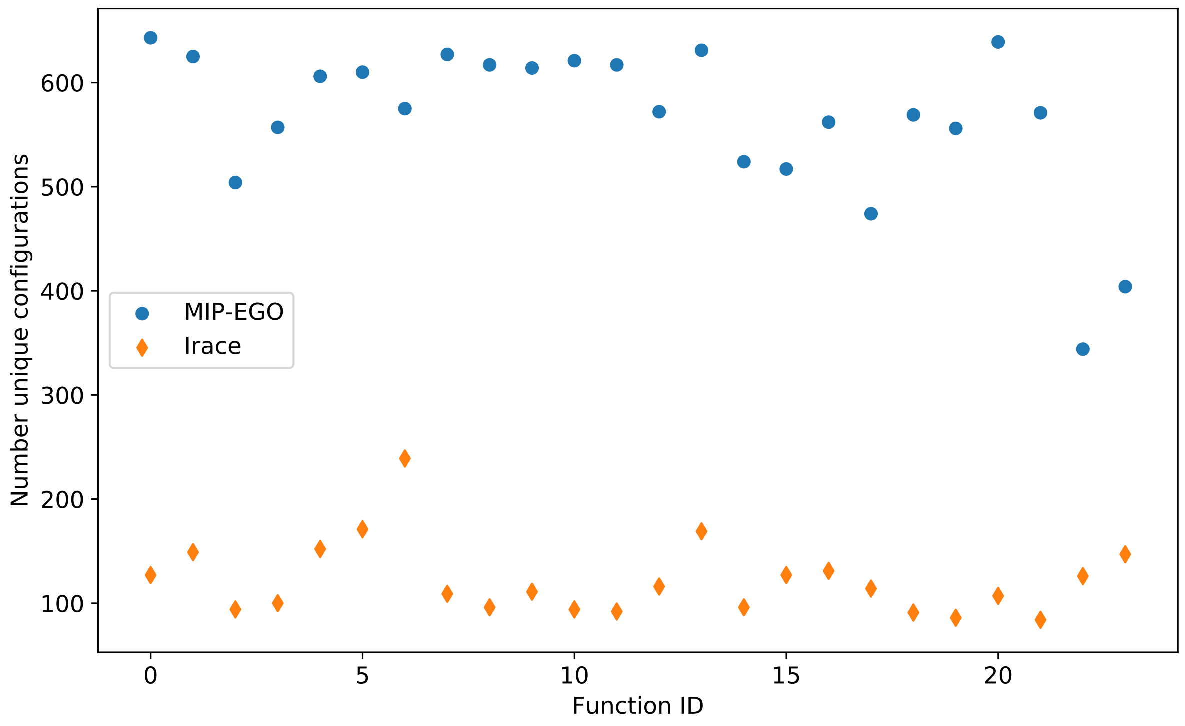


Figure 26: Number of distinct configurations explored after the initial sampling by both irace and MIP-EGO. MIP-EGO starts with a sample of 250 points, while irace starts with 333 candidates in the first race.

balance between exploration and exploitation is heavily favored towards exploitation of a single configuration, which explains why different runs of irace can produce widely different results, as shown in the initial experiment on F12.

5.5 Variance of results

In the previous section, we showed that irace is much less exploratory than MIP-EGO. One of the results from this focus on exploitation might be a large variance in performance of the found (configuration, hyperparameters)-pairs, relative to those found by MIP-EGO. To test this hypothesis, two functions, F1 and F20, were selected, on which 15 runs of irace and MIP-EGO are performed. Both methods get a budget of $25,000$ total runs, and the resulting (configuration, hyperparameters)-pairs are rerun 250 times.

From this experiment, the differences in terms of hitting time distributions between MIP-EGO and irace on F1 and F20 can be visualized. This is done in Figures 27 and 28. These figures show that irace significantly outperforms MIP-EGO on F1, while the differences in performance in F20 are much smaller. However, even for F20, these differences are still statistically significant (2-sample KS test, $\alpha = 0.01$). This indicates that the exploration done by MIP-EGO might not be as beneficial as predicted. Even tough irace considers significantly fewer configurations, it manages to outperform MIP-EGO by finding much more stable hyperparameters.

While the differences in distributions of hitting times might be explained by the balance between exploration and exploitation, that is not the only factor impacting the performance. As noted

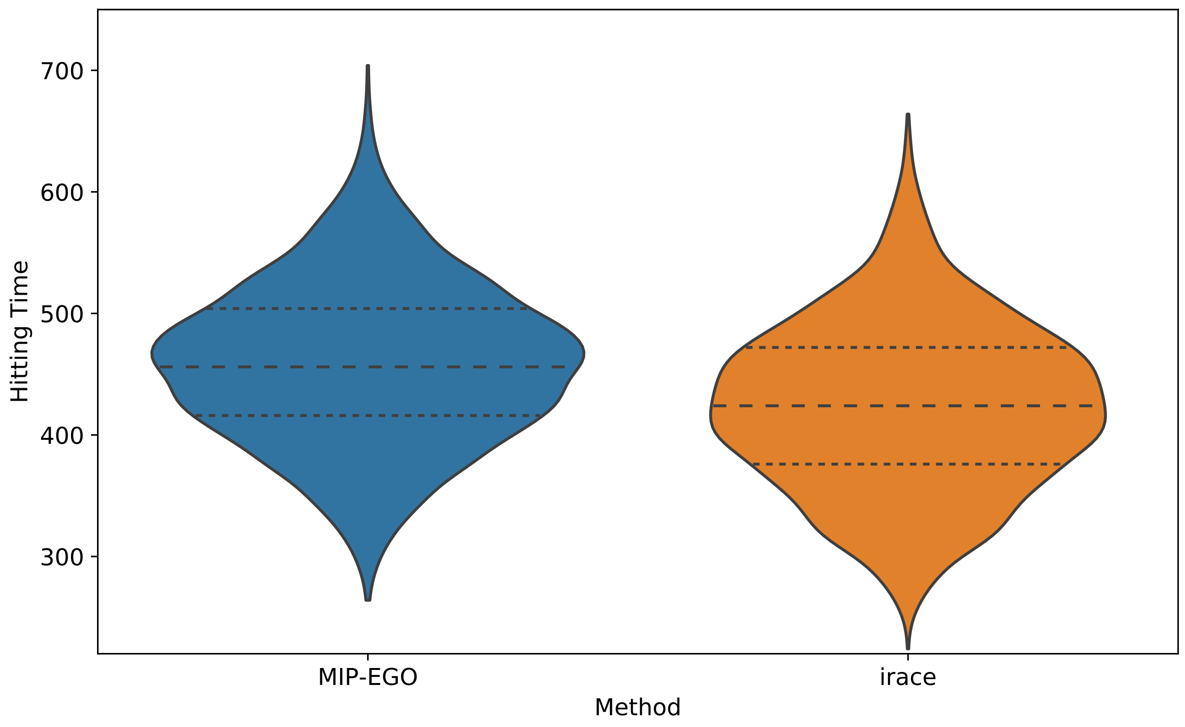


Figure 27: Distributions of hitting times of 15 (configuration, hyperparameters)-pairs (resulting from 15 independent runs of the integrated approaches), each of which run 250 times on benchmark function F1.

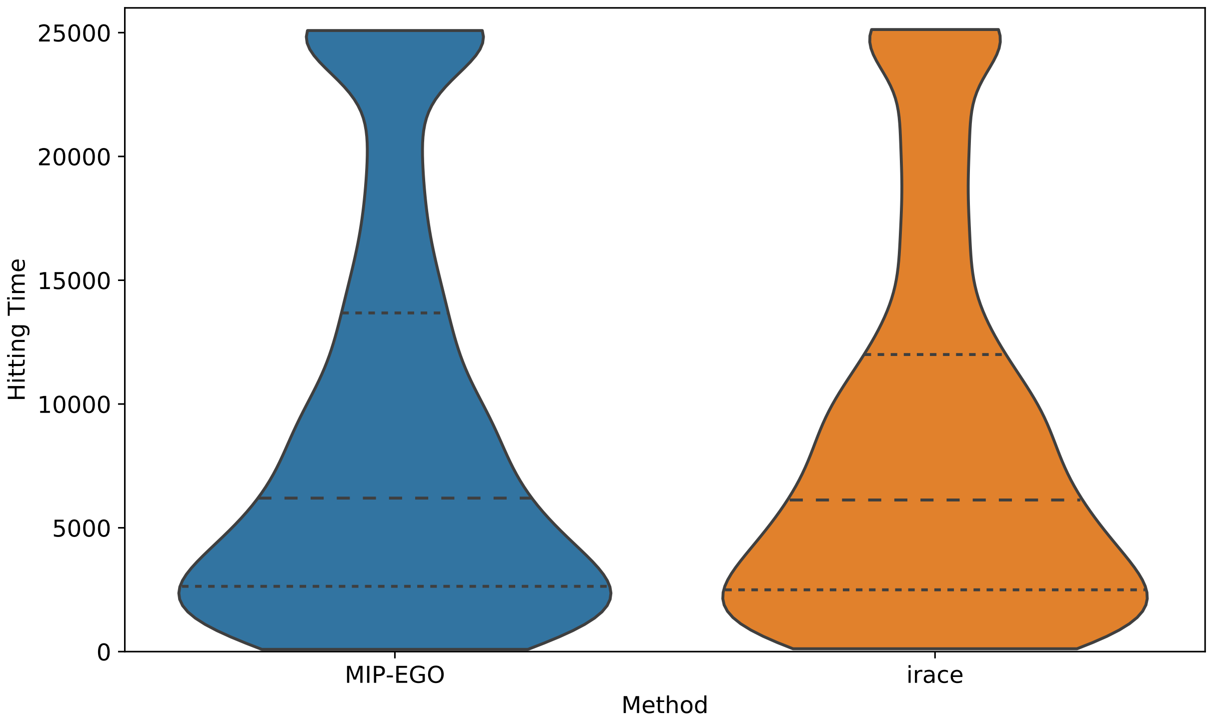


Figure 28: Distributions of hitting times of 15 (configuration, hyperparameters)-pairs (resulting from 15 independent runs of the integrated approaches), each of which run 250 times on benchmark function F20.

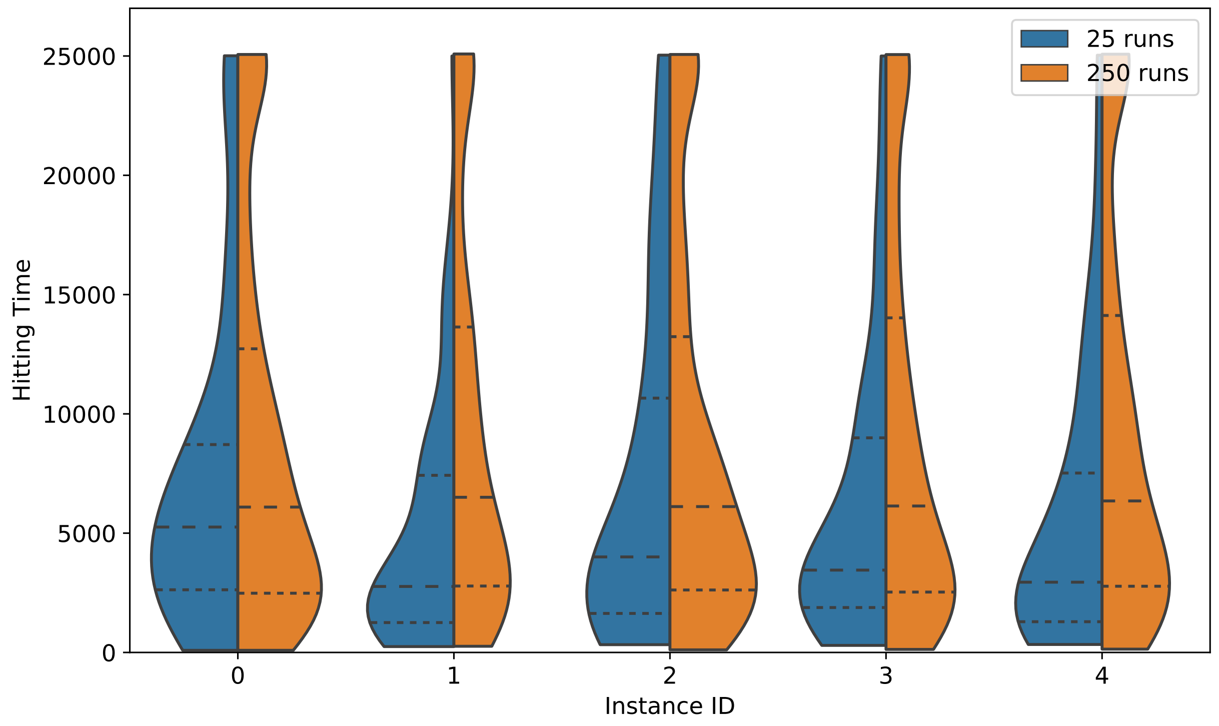


Figure 29: Distribution times of 15 different (configuration, hyperparameters)-pairs found in 15 independent runs of MIP-EGO on benchmark function F20. Split into two parts: 25 runs, which were evaluated during the MIP-EGO run and used to select the (configuration, hyperparameters)-pair, and 250 runs which were run afterwards to verify the results.

before, MIP-EGO bases its selection on only 5×5 runs, thus variance can have a huge impact on which configuration and hyperparameter setting is selected. This is confirmed when inspecting the ERTs MIP-EGO found from these 25 runs. For F20, it achieves an average ERT of $6,220$ during the run. However, when validating this (configuration, hyperparameters)-pair on 250 runs, the average ERT becomes $9,058$. This difference can also be visualized, as is done in Figure 29. When comparing the hitting time distributions of the runs which were done during MIP-EGO, the large differences towards the actual distributions are made visible. The most noticeable differences are the absence of hitting times of $25,000$, which is expected, since these hitting times indicate that the target was not hit, which has a large impact on the ERT.

5.6 Summary

From these experiments, it has been made clear that integrating algorithm selection and algorithm configuration into a single approach is very beneficial, leading to a much more efficient and effective approach to find good (configuration, hyperparameters)-pairs. There are some fundamental differences between the two approaches, MIP-EGO and irace, which were used in this work. However, both manage to achieve similar rates of improvement over sequential execution of algorithm selection and configuration. The full comparison between all methods discussed in this thesis is shown in the Appendix, in Table 5.

The relative gain of the Virtual Best Solver (VBS), which is the method which for each function selects the best integrated method, relative to several baseline ERTs (of configurations with

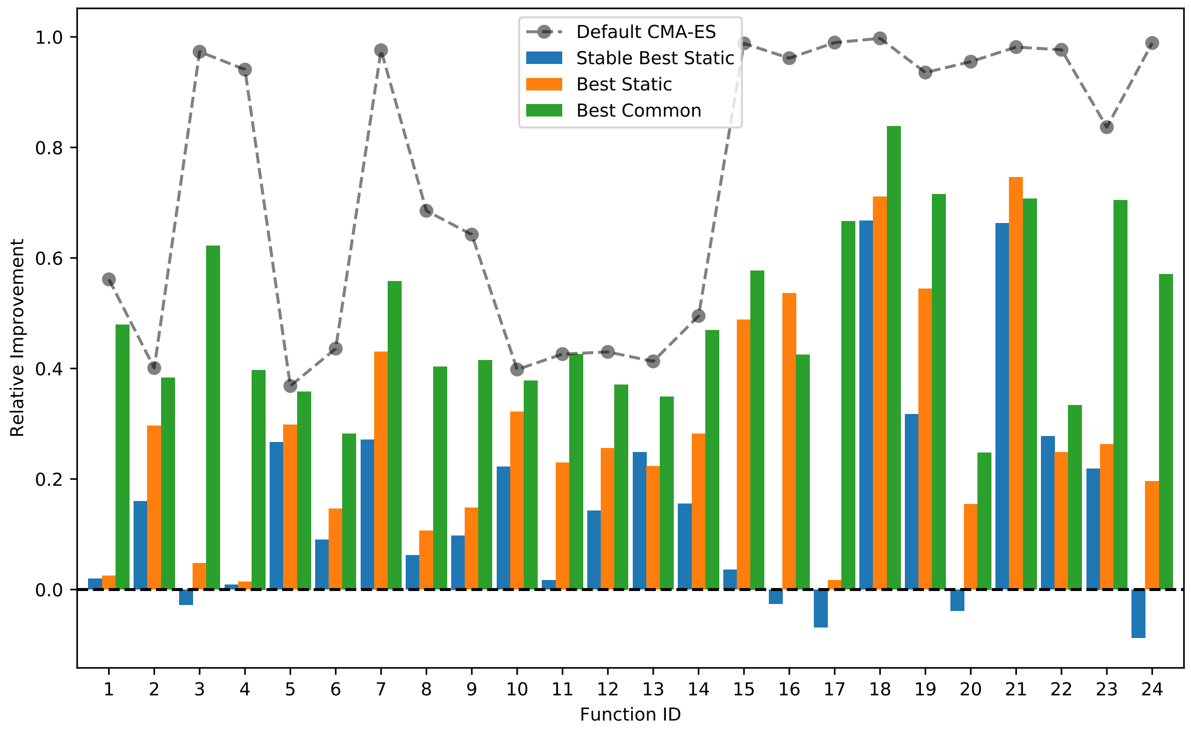


Figure 30: Relative improvement of the virtual best solver (best of MIP-EGO and Irace) against the best common configuration, the best static configuration (best on 25 runs), the stable best static configuration (best on 250 runs) and the default CMA-ES. All ERTs are gathered from 250 runs, with default hyperparameter values.

default hyperparameters) is shown in Figure 30. This indicates that the improvements possible by using the integrated methods described in this section can be as high as 50% when compared to the best configurations with default hyperparameters.

A summary of the main differences between the four main methods described in this thesis can be seen in Table 4. From this, we can see that the differences in terms of performance between the integrated and sequential methods is minimal, while they require a significantly lower budget. This budget value is in no way optimized, so an even lower budget than the one we used might achieve similar results. This might especially be true for irace, since we saw in Section 5.4 that it uses most of its budget to evaluate very small changes in hyperparameter values. An initial investigation into the effect of budget on the performance of irace is available in Appendix C. A final comparison between the sequential and integrated approaches is shown in Figure 31, which shows the ECDF-curves for both methods, as well as for the default CMA-ES. From this figure, it confirmed that the performance of the integrated and sequential methods is extremely similar, with both approaches significantly outperforming the default CMA-ES variant.

Method	**Naïve Sequential**	**Sequential**	**MIP-EGO**	**Irace**
Best on # of functions	0	9	9	6
Average Improvement over best modular CMA-ES	6.3%	24.7%	20.2%	20.7%
Average Improvement over default CMA-ES	67.4%	73.0%	72.9%	72.5%
Average Prediction Error	23.2%	18.8%	17.4%	10.6%
Budget (# function evaluations)	$120,200$	$150,000$	$25,000$	$25,000$
Percentage of unique configurations	95.8%	76.8%	77.8%	9.7%

Table 4: Comparison of the four methods for determining (configuration, hyperparameters)-pairs used in this thesis. Improvement over best modular CMA-ES refers to the relative improvement in ERT over the single best configuration with default hyperparameters. Percentage of unique configurations refers to how many of the evaluated candidates during the search contained unique configurations.

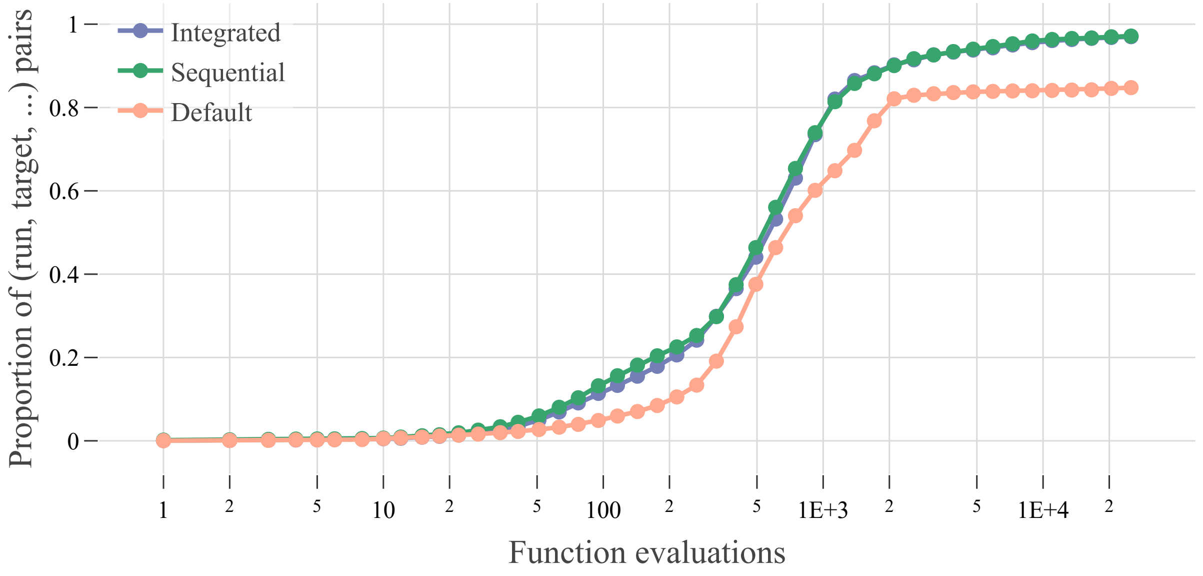


Figure 31: Empirical Cumulative Distribution Function (ECDF) of the default CMA-ES, the sequential approach as described in Section 4.7, and the virtual best integrated solver (best of irace and MIP-EGO). Results are based on 250 runs. This visualization was generated by the IOHprofiler [DWY+18]. The ECDF shows the average fraction of (run, target)-pairs which were reached within x function evaluations by the respective algorithms.

6 Conclusions

This thesis focused on the influence of hyperparameter tuning on the performance of CMA-ES, and aimed to investigate a way to integrate algorithm configuration and algorithm selection into a single approach. The many-algorithm context of modEA, which allows for the generation of $4,608$ different CMA-ES variants, proved to be a useful resource to perform experiments on both algorithm selection and algorithm configuration.

We have shown that hyperparameter tuning can have a large impact on the performance of different CMA-ES configurations. However, this gain in performance is not consistent, since some configuration benefit a lot more from having tuned hyperparameters than others. When comparing the ranking of configurations on ERT with their default hyperparameters to their ranking with tuned hyperparameters, we showed that large difference in ranking are possible. This shows the importance of hyperparameter tuning, as well as the fact that a simple sequential execution of algorithm selection and hyperparameter tuning might not result in finding the best (configuration, hyperparameters)-pair.

The main contribution of this thesis is the study of integrated approaches for algorithm selection and hyperparameter tuning. We used two hyperparameter tuning tools, MIP-EGO and irace, to optimize both the configuration and its hyperparameters at the same time. We showed that both methods manage to find (configuration, hyperparameters)-pairs which perform better than those found by the naïve sequential approach, while requiring significantly fewer function evaluations. Even when compared to a more robust sequential approach, the integrated methods manage to achieve similar performances, with a factor 6 fewer function evaluations needed.

When comparing MIP-EGO and irace, we notice that the differences in terms of performance are minimal. This holds both when just considering hyperparameters tuning as well as the integrated algorithm selection and configuration approach. However, the fundamental differences in the working mechanisms between these methods are significant. While MIP-EGO is designed to maintain a balance between exploration and exploitation, irace has a heavy focus on exploitation of a small set of configurations. The other main difference between the two methods is that MIP-EGO aims to minimize ERT over a set number of runs per instance, while irace allocates runs to instances dynamically, but minimizes penalized AHT. Even tough we identified these large differences in the working mechanisms of MIP-EGO and irace, the differences in terms of achieved ERT is minimal. This indicates that there is still room for improvement by combining the best parts from both methods.

7 Future Work

This work has used two hyperparameter tuning methods, MIP-EGO and irace, to create an integrated approach for algorithms selection and configurations. We have show that both methods manage to outperform the naive sequential approach, both in terms of final performance as in terms of function evaluations needed. However, the differences between these two methods are significant. This leads to the question of combining the best parts from both methods into a single approach. This could take advantage of the dynamic allocation of runs to instances and adaptive capping which irace uses, as well as the efficient generation of new candidate solutions using the working principles of efficient global optimization, as done in MIP-EGO.

Another extension to this work is the adaptation of the proposed approaches to the configuration switching context. Configurations switching, as introduced in [vRDB18], has been shown to give promising results in [VvRBD19]. A brief summary of the configuration switching is included in Appendix B. To introduce hyperparameter tuning into the switching approach would mean to create a quintuple $(C_1, P_1, \sigma, C_2, P_2)$, where (C_1, P_1) is the initial (configuration, hyperparameters)-pair, which switches to (C_2, P_2) after target σ (the splitpoint) is reached. To determine such a quintuple using the approach as described in [VvRBD19] would be unfeasible, as the size of the search space grows too large to allow for complete enumeration. However, the selection of (C_1, P_1) can be achieved using the integrated algorithm selection and configuration approaches described in this thesis. The main challenge for future work would then become the selection of the splitpoint and (C_2, P_2).

As a final, long term extension, we propose to investigate the possibility to create an algorithm which can adapt its configuration and hyperparameters online. This is might be achieved by using local landscape features or the internal state of the CMA-ES-parameters to determine when a switch would be beneficial. Some initial work in determining how landscape features change based on the current point in the search has been done in [JD19], and exploration of which features are stable has been done in [RDDD19]. Combining these works might eventually be able to lead to the creation of an adaptive, landscape-aware CMA-ES.

References

[ABH11] Anne Auger, Dimo Brockhoff, and Nikolaus Hansen. Mirrored Sampling in Evolution Strategies with Weighted Recombination. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, pages 861–868. ACM, 2011.

[AH05] A. Auger and N. Hansen. A restart CMA evolution strategy with increasing population size. In *2005 IEEE Congress on Evolutionary Computation*, volume 2, pages 1769–1776 Vol. 2, September 2005.

[AJT05] Anne Auger, Mohamed Jebalia, and Olivier Teytaud. Algorithms (x, sigma, eta): Quasi-random mutations for evolution strategies. In El-Ghazali Talbi, Pierre Liardet, Pierre Collet, Evelyne Lutton, and Marc Schoenauer, editors, *Artificial Evolution, 7th International Conference, Evolution Artificielle, EA 2005, Revised Selected Papers*, volume 3871 of *Lecture Notes in Computer Science*, pages 296–307. Springer, 2005.

[BAH$^+$10] Dimo Brockhoff, Anne Auger, Nikolaus Hansen, Dirk V. Arnold, and Tim Hohm. Mirrored Sampling and Sequential Selection for Evolution Strategies. In *Parallel Problem Solving from Nature, PPSN XI*, pages 11–21. Springer, September 2010.

[BDSS17] Nacim Belkhir, Johann Dréo, Pierre Savéant, and Marc Schoenauer. Per instance algorithm configuration of CMA-ES with limited budget. In *Proc. of Genetic and Evolutionary Computation Conference*, pages 681–688. ACM, 2017.

[Bel17] Nacim Belkhir. *Per Instance Algorithm Configuration for Continuous Black Box Optimization*. Thesis, Université Paris-Saclay, November 2017.

[BFK13] Thomas Bäck, Christophe Foussette, and Peter Krause. *Contemporary Evolution Strategies*. Natural Computing Series. Springer, 2013.

[CLIHS17] Leslie Pérez Cáceres, Manuel López-Ibáñez, Holger Hoos, and Thomas Stützle. An experimental study of adaptive capping in irace. In Roberto Battiti, Dmitri E. Kvasov, and Yaroslav D. Sergeyev, editors, *Learning and Intelligent Optimization*, pages 235–250, 2017.

[DWY$^+$18] Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck. IOHprofiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics. *arXiv e-prints:1810.05281*, October 2018.

[HAB$^+$16] Nikolaus Hansen, Anne Auger, Dimo Brockhoff, Dejan Tušar, and Tea Tušar. COCO: Performance Assessment. *arXiv:1605.03560 [cs]*, May 2016. arXiv: 1605.03560.

[Han08] Nikolaus Hansen. CMA-ES with Two-Point Step-Size Adaptation. *arXiv:0805.0231 [cs]*, May 2008. arXiv: 0805.0231.

[Han09] Nikolaus Hansen. Benchmarking a BI-population CMA-ES on the BBOB-2009 Function Testbed. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, pages 2389–2396. ACM, 2009.

[HHLB11] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[JA06] G. A. Jastrebski and D. V. Arnold. Improving Evolution Strategies through Active Covariance Matrix Adaptation. In *2006 IEEE International Conference on Evolutionary Computation*, pages 2814–2821, 2006.

[JD19] Anja Janković and Carola Doerr. Adaptive landscape analysis. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 2032–2035. ACM, 2019.

[LIDLC$^+$16] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43 – 58, 2016.

[LIDLSB11] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari. The irace package, iterated race for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.

[LIPC] Manuel López-Ibáñez and Leslie Pérez Cáceres. The irace package: Iterated race for automatic algorithm configuration. `http://iridia.ulb.ac.be/irace/`.

[LJD$^+$16] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

[MBT$^+$11] Olaf Mersmann, Bernd Bischl, Heike Trautmann, Mike Preuss, Claus Weihs, and Günter Rudolph. Exploratory landscape analysis. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 829–836. ACM, 2011.

[PMEVBR$^+$15] A. Piad-Morffis, S. Estévez-Velarde, A. Bolufé-Röhler, J. Montgomery, and S. Chen. Evolution strategies with thresheld convergence. In *2015 IEEE Congress on Evolutionary Computation (CEC)*, pages 2097–2104, May 2015.

[RDDD19] Quentin Renau, Johann Dreo, Carola Doerr, and Benjamin Doerr. Expressiveness and robustness of landscape features. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 2048–2051. ACM, 2019.

[vR18] Sander van Rijn. Modular cma-es framework from [vRWvLB16], v0.3.0. `https://github.com/sjvrijn/ModEA`. Available also as pypi package at `https://pypi.org/project/ModEA/0.3.0/`, 2018.

[vRDB18] Sander van Rijn, Carola Doerr, and Thomas Bäck. Towards an adaptive cma-es configurator. In *Proc. of 15th International Conference on Parallel Problem Solving from Nature (PPSN'18)*, volume 11101 of *Lecture Notes in Computer Science*, pages 54–65. Springer, 2018.

[vRWvLB16] Sander van Rijn, Hao Wang, Matthijs van Leeuwen, and Thomas Bäck. Evolving the structure of Evolution Strategies. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, December 2016.

[vRWvSB17] Sander van Rijn, Hao Wang, Bas van Stein, and Thomas Bäck. Algorithm Configuration Data Mining for CMA Evolution Strategies. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 737–744. ACM, 2017.

[VvRBD] Diederick Vermetten, Sander van Rijn, Thomas Bäck, and Carola Doerr. Github repository for online selection of CMA-ES variants. GitHub repository containing data is available at `https://github.com/Dvermetten/Online_CMA-ES_Selection`.

[VvRBD19] Diederick Vermetten, Sander van Rijn, Thomas Bäck, and Carola Doerr. Online selection of CMA-ES variants. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 951–959. ACM, 2019. Full report avaialbe at `arXiv:1904.07801`.

[VWBD] Diederick Vermetten, Hao Wang, Thomas Bäck, and Carola Doerr. Github repository containing the data from this thesis. `https://github.com/Dvermetten/CMA_ES_hyperparam`.

[WEB14] Hao Wang, Michael Emmerich, and Thomas Bäck. Mirrored Orthogonal Sampling with Pairwise Selection in Evolution Strategies. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 154–156. ACM, 2014.

[WEB18] H. Wang, M. Emmerich, and T. Bäck. Cooling strategies for the moment-generating function in bayesian global optimization. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, July 2018.

[WM97] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Trans. Evolutionary Computation*, 1(1):67–82, 1997.

[WvSEB17] Hao Wang, Bas van Stein, Michael Emmerich, and Thomas Back. A new acquisition function for bayesian optimization based on the moment-generating function. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 507–512. IEEE, 2017.

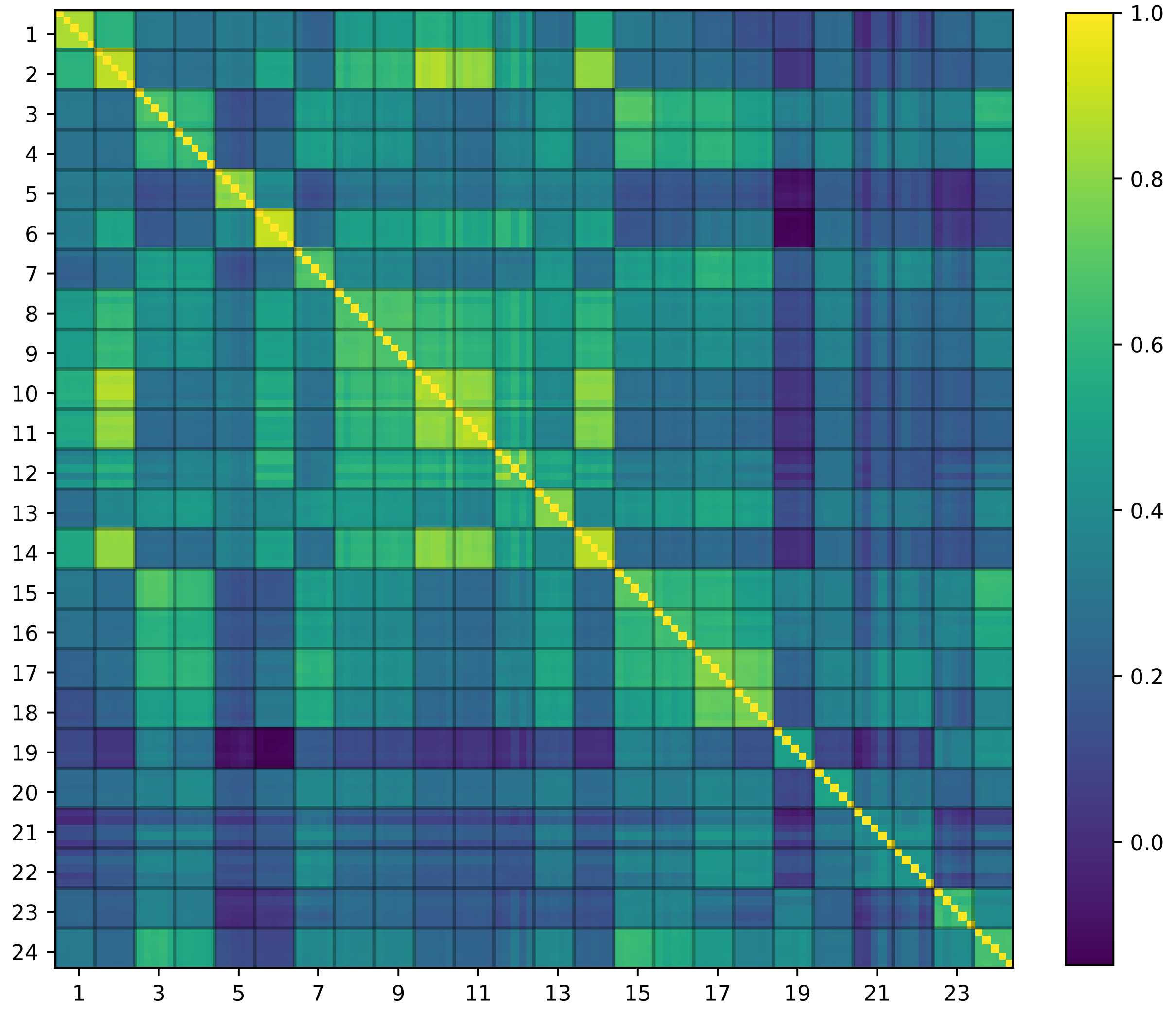


Figure 32: Kendall correlation of rankings of all configuration between instances of all benchmark functions. Rankings based on ERT from 5 runs per instance.

A Impact of instance variation

All experiments performed in this thesis have been aggregated over 5 instances of the selected benchmark function. However, it has previously been shown [BDSS17] that the optimal hyperparameter values might differ per instance. In this section, the data generated during the experiment described in Section 4.7 is analyzed to get an insight into the differences between the instances of the bbob-functions used. Figure 33 shows the distributions of hitting times for each instance of each function. This shows us that some functions have very significant differences between their instances. The most obvious example is F12, for which the differences in hitting times are very significant. This matches previous observations from Section 4.2, specifically Figure 7, in which we observed the clear multi-modality of the hitting times.

Another way to look at the differences between instances is to use the 5×5 hitting time data, which is available for all configurations. Figure 32 shows the Kendall correlations between these rankings across 5 instances of all benchmark function. This seems to confirm the findings from Figure 33, in that for most functions the differences between their instances is quite small. For F12, there is also a clear distinction between some of the instances, which matches our previous observations.

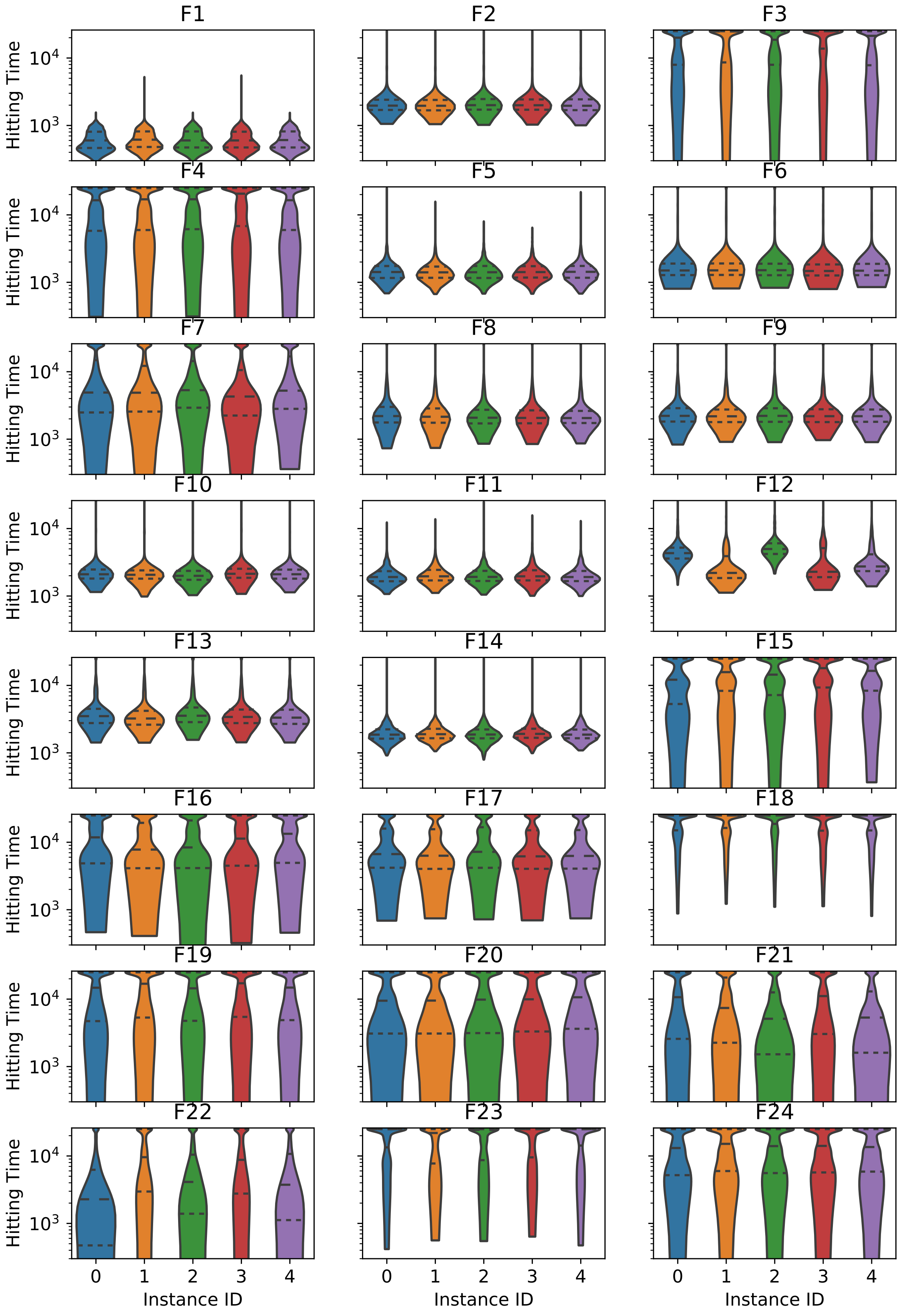


Figure 33: Distributions of hitting times per instance for all benchmark functions. All distributions are from 30 configurations (as described in Section 4.7), each with tuned hyperparameters (using MIP-EGO) and run 50 times on each instance.

B Configuration Switching

Previous work on modEA has introduced the concept of configuration switching [vRDB18]. This is motivated by the fact that different configuration have different convergence behaviour. These differences might be exploited by starting with a configuration c_1 which reaches a certain target σ with the fewest number of functions evaluations. We refer to this target σ as the splitpoint, as once this target is hit, a switch is made to a different configuration C_2. This principle is visualized in Figure 34.

In [VvRBD19], this configuration switching was implemented. Based on the results from this paper, we noticed that variance plays a big role in the performance of switching configurations. To mitigate this, a two-stage approach was developed to determine which switching configurations to run. This approach first gathers a collection of 'interesting' configurations: the top 50 best static ones and the configurations used in the top 50 theoretically best switching ones (determined by complete enumeration). Then these configurations were run 250 times to get some more robust data on which to base the final selection of switching configurations.

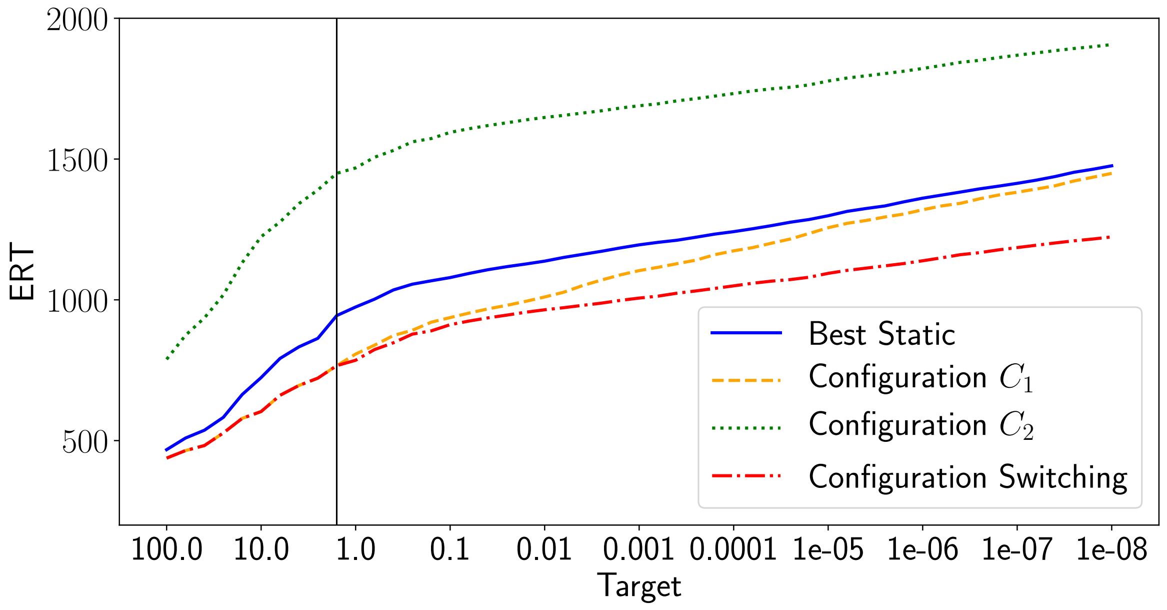


Figure 34: Example of how a configuration switch could theoretically outperform the best static configuration. It follows the convergence path of C_1 until the splitpoint σ, after which it follows the exact convergence behaviour of C_2, thus reaching the final target with less evaluations than the best static configuration.

C Impact of budget on performance of irace

In this thesis, the execution of the integrated algorithm selection and configuration has been performed with a fixed budget, namely $25,000$ evaluations. However, this is a rather arbitrary limit, and similar results might be achieved with a much lower budget. To test this hypothesis, another experiment was performed: For several different budget values, 3 runs of irace were performed on F12. As always, the resulting (configuration, hyperparameters)-pairs from this experiment are then run 250 times to allow for a fair comparison. The hitting time distributions resulting from these runs are visualized in Figure 35.

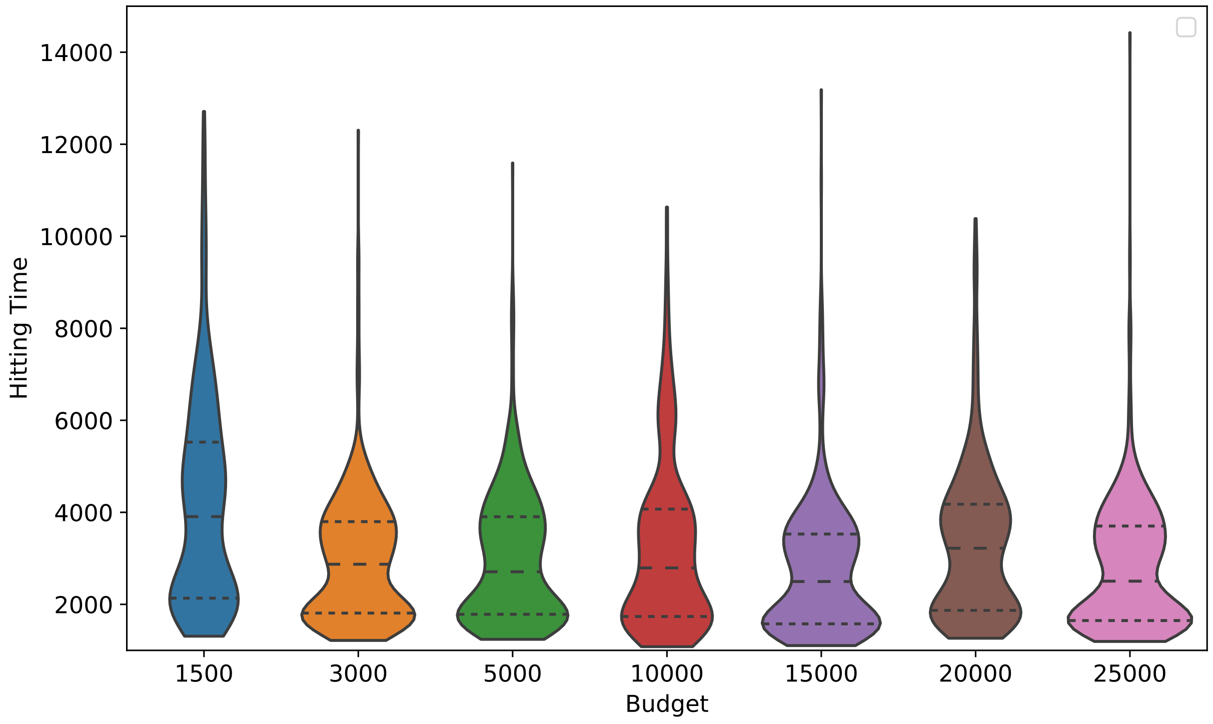


Figure 35: Hitting time distributions from (configuration, hyperparameters)-pairs as determined by irace using different budgets. Hitting times are from 3 runs of irace, for 250 runs each, for a total of 750 runs per budget-value.

From Figure 35, it can be seen that the differences in performance between different budget values are present, but not very large. For the lower budget values, the hitting times have a much higher variance, which might point to a larger prediction error, since a distribution with high variance might get picked over a better one with low variance when very few samples are selected. To verify this, we calculated the average prediction error for each budget value. These are visualized in Figure 36. While this shows clearly that 3 samples is too little to make any definitive statements, it seems like the prediction error is only slightly worse for the lowest budget values compared to the highest.

While the scope of this experiment is too small to draw any real conclusions, it seems to indicate that a smaller budget might work similarly well to the $25,000$ which was used in this thesis. A more robust search into the optimal budget value might be done in future work, but is outside of the scope of this project, as this only aims to prove the viability of the integrated algorithm selection and configuration method, which is still shown with a budget of $25,000$.

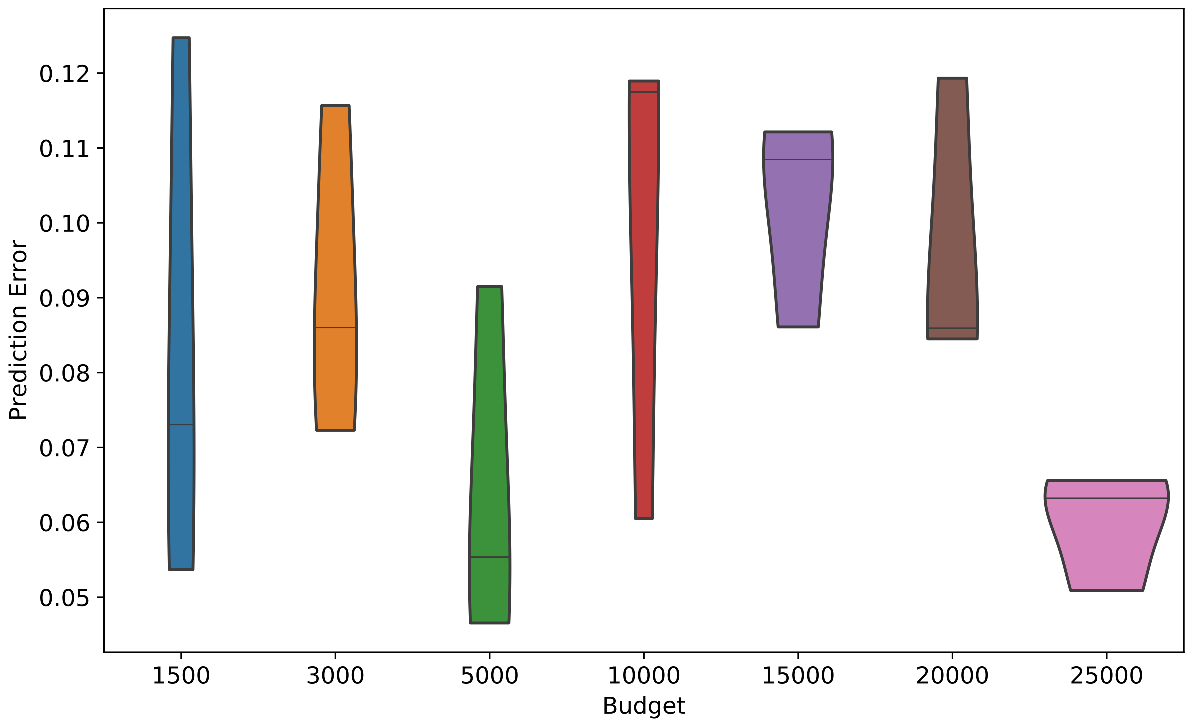


Figure 36: Distributions of prediction errors from (configuration, hyperparameters)-pairs as determined by irace using different budgets. Each is based on 3 runs of irace, comparing the ERT for 250 runs to the predicted ERT.

D Parsing Configuration IDs

Throughout this thesis, we refer to configurations within the modEA framework by their configuration ID. This number uniquely represents the configuration in terms of its module configuration. To convert the module activations vector into their configuration ID, we can use the following algorithm:

$$\text{ConfID}(\vec{m}) = \begin{bmatrix} 2304 & 1152 & 576 & 288 & 144 & 72 & 36 & 18 & 9 & 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} m_1 \\ \vdots \\ m_{11} \end{bmatrix}$$

Where m_x indicates the value of module number x, as in Table 1. To convert from configuration ID to module activations, the inverse function is used.

FID	Target	Common Conf	Common ERT	Static Conf (25)	Static ERT (25)	Static Tuned ERT	Static Conf (250)	Static ERT (250)	Seq. Conf	Seq. ERT	MIP-EGO Conf	MIP-EGO ERT	Irace Conf	Irace ERT
1	10e-8.0	1152	521	921	447	456	3227	445	3229	**392**	903	487	4598	436
2	10e-8.0	0	1,781	3714	2,042	1,921	1281	1,711	4585	1,573	1264	**1,438**	867	1,501
3	10e0.4	2	16,655	869	12,828	12,604	872	11,896	149	**11,688**	617	19,158	3173	12,221
4	10e0.8	2306	12,869	884	11,933	12,896	599	11,866	896	12,747	1028	14,043	884	**11,767**
5	10e-8.0	594	1,254	1311	1,607	1,540	1010	1,539	3190	**1,115**	1034	1,148	1029	1,128
6	10e-8.0	3457	1,363	2166	1,354	1,120	2039	1,271	2165	**1,102**	2176	1,224	1902	1,156
7	10e-8.0	2	3,936	3208	4,439	3,525	1436	3,468	40	3,071	1570	3,577	1274	**2,527**
8	10e-8.0	3458	2,710	1016	1,985	1,884	866	1,891	864	1,872	869	**1,774**	3176	1,790
9	10e-8.0	2306	2,704	1015	2,045	2,068	870	1,931	869	**1,695**	869	1,983	3176	1,742
10	10e-8.0	0	1,898	3572	2,161	2,043	1309	1,882	1008	1,615	867	**1,465**	885	1,493
11	10e-8.0	1	1,822	2706	1,872	1,872	3281	1,467	3281	1,472	3281	1,522	3208	**1,443**
12	10e-8.0	594	3,930	2019	3,568	3,422	864	3,096	864	2,825	3172	**2,655**	1030	3,037
13	10e-8.0	0	2,847	1166	3,241	3,100	1166	3,350	3188	2,740	2596	**2,517**	2308	2,544
14	10e-8.0	0	1,714	1267	1,912	1,873	1418	1,625	1418	1,538	3317	1,432	3	**1,372**
15	10e0.4	2	13,975	865	15,351	15,043	872	8,145	1013	8,629	3317	9,515	1013	**7,854**
16	10e-2.0	2	9,784	3187	12,035	10,614	890	5,437	887	**5,245**	896	5,864	890	5,579
17	10e-4.4	2305	7,257	869	2,889	2,859	872	2,658	869	2,859	869	3,418	866	**2,840**
18	10e-4.0	1	16,234	865	18,383	19,915	1664	15,968	1664	11,149	4118	**5,312**	2099	5,740
19	10e-0.6	2	11,261	586	17,810	10,295	590	11,886	590	**7,718**	3173	8,113	3181	11,357
20	10e0.2	2	9,723	4379	11,439	11,174	1988	9,313	1988	**8,505**	440	10,307	3860	9,671
21	10e-0.6	3458	8,489	4343	11,149	9,713	1283	8,406	2900	2,939	2614	**2,831**	2470	2,931
22	10e0.0	3458	4,473	1592	4,767	5,097	2039	4,957	3458	4,564	2610	**3,582**	1337	6,699
23	10e-0.8	594	15,472	872	17,549	13,916	866	16,566	866	13,912	869	**12,938**	4145	25,812
24	10e1.0	2	12,475	2882	8,272	7,071	881	6,117	3029	**5,893**	1016	7,950	878	6,653

Table 5: Comparison of all methods discussed in this thesis. For each function, the used target value is selected in accordance with Table 3 from [VvRBD19]. All columns with 'conf' represent configuration numbers. The common configurations are taken from Table 2. Static (25) refers to the best configuration based on the 25 hitting times available for all configurations, while Static (250) refers to the best configuration after rerunning the set of configurations from Section 4.7 and selecting the one with the best ERT on 250 runs. Static tuned ERT is the same as the naive sequential approach, while Seq. refers to the sequential approach as described in Section 4.7. All ERT data is based on hitting times for 50 runs on 5 instances. The lowest ERTs per function are bolded.