Universiteit Leiden

Master Computer Science

Building a Portfolio of Optimization Algorithms for Good Anytime Performance via Automated Algorithm Configuration

Name:	Navin Pophare
Student ID:	S2691957
Date:	24/06/2022
Specialisation:	Computer Science: Data Science
1st supervisor:	Hao Wang
:	LIACS, Leiden University
2nd supervisor:	Carola Doerr
:	CNRS and Sorbonne Université, Paris

Master's Thesis in Computer Science

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

Abstract

In recent years, there has been extensive research in the domain of optimization algorithms and in building algorithm portfolios. In our work, we aim at building a parallel algorithm portfolio that achieves good anytime performance. Anytime performance is the performance of portfolio algorithms irrespective of the budget/objective function for which they have been run.

To obtain our results, we apply automated algorithm configuration to the best-known portfolio algorithm that achieves further improved anytime performance. We base all our experiments on YABBOB (Yet Another Black-Box Optimization Benchmark) suite, available in the Nevergrad platform. For our experiments, we make use of area under average ECDF (Empirical Cumulative Distribution Function) curve as the performance metric for portfolio algorithms. We obtained approximately 4% improvement in terms of anytime performance for the portfolio algorithm over the single best solver. Furthermore, we obtained an additional 1.5% improvement for the best hyperparameter configured version of the portfolio algorithm over the default version.

Contents

Chapter 1

Introduction

Since the inception of machine learning and artificial intelligence, optimization algorithms have been a crucial topic of study for computer scientists and mathematicians. While numerous optimization algorithms have been studied and developed over the past few decades, constructing a portfolio of optimization algorithms [Bau14] and configuring their parameters in a way that would yield maximal performance on objective functions or decision problems, has always been a challenge. Algorithm configuration is the tuning of parameters of the optimization algorithms such that the performance of the algorithm is enhanced. The parameter values shape the search trajectory of the algorithm for solutions and hence affect the quality of solutions found by the algorithm, therefore configuration is often considered an essential step in building portfolios.

In our work, we aim to build a parallel portfolio of optimization algorithms via automated algorithm configuration to achieve a good anytime performance. In several real-world situations, we often do not know in advance how much time will we have to solve a given optimization problem. Therefore, we require an algorithm that would be expected to achieve a good performance by finding high-quality solutions at any given moment irrespective of the runtime (budget/number of function evaluations) available. This performance can also be referred to as *anytime performance*. We aim to combine algorithms in pairs and run them in parallel in order to exploit complementarity of the two algorithms as well as the advantages of parallel computing resources.

1.1 Research Question

Our work primarily aims to search for and select the best performing Nevergrad [RT18] optimizers for a set of numerical objective functions (also implemented in Nevergrad) for constructing a portfolio. We make use of area under average ECDF (Empirical Cumulative Distribution Function) [Dek10] curve as the performance metric for the algorithms in our

portfolio. Formally and to be more specific, we investigate:

- The performance improvement of combinations of algorithms run in parallel over single algorithms
- Whether these combinations can be further configured in order to achieve enhanced performance.
- The discrepancy between tuning hyperparameters of optimizers for overall performance across functions and tuning for performance on each function individually.

We test several optimization algorithms on numerical functions and evaluate the optimizers on the basis of cumulative performance across functions. The optimizers and functions we perform our experiments on are implemented in the optimization platform Nevergrad [RT18]. Nevergrad is an open-source Python3 library that offers an extensive collection of algorithms that are gradient-free and presents them in a standard ask-and-tell Python framework. The library offers a wide range of optimizers along with objective functions and benchmark routines for smooth comparison of optimizers. We also make use of IOHprofiler [Doe+18] which is a benchmarking and profiling tool for iterative optimization heuristics. IOHprofiler or IOH is available as an open-source Python library and we utilize its functionalities in our work mainly for the purpose of analyzing results obtained from the experiments conducted with Nevergrad. Our experimentation includes simulation of parallelism to run optimizers in combinations, in order to construct a portfolio. Furthermore, we aim to explore configuration spaces of each of the best performing solvers to search for parameter configurations corresponding to further improved performance with the help of the R-package *irace* [Lóp+16].

Our portfolio algorithm is able to obtain an approximate 4% improvement over the single best solver. We also determine the appropriate hyperparameter tuning method for the purpose of algorithm configuration, and our portfolio algorithm achieves a further approximate improvement of 1.5% in performance after algorithm configuration.

The structure of this report is as follows: in Chapter 2 we discuss a brief overview of related work around this thesis; in Chapter 3 we elaborate the concepts and background relevant to our work; in Chapter 4, we elucidate the methodologies along-with tools used to conduct experiments and analyze respective results; in Chapter 5 we outline the results with appropriate discussion and finally in Chapter 6 we provide conclusions explaining our findings along with proposing possibilities of future work.

Chapter 2

Related Work

Our work is centered around parallel portfolio construction of optimization algorithms. This section accounts for current research that influenced our work.

2.1 Algorithm Selection

The foundation of portfolio construction was laid by John R. Rice in 1976 [Ric76]. His work primarily focused on selection of the most appropriate optimization algorithm based on the problem(s) at hand. For this purpose, an abstract model was defined that would take into account the characteristics of the problem(s) to predict the best-suited algorithm. The characteristics of the problem(s) at hand were termed as *features* and hence the model's objective was to determine a selection mapping scheme from the problem's feature space or from the problem space to the algorithm space such that the performance of the algorithm on each problem in the problem space, in terms of a performance metric, is maximized. The major challenge faced by this approach was dealing with large and diverse problem spaces as well as algorithm spaces.

2.2 Algorithm Portfolios

Gomes and Selman proposed the construction of algorithm portfolios in 1999 in their work: Algorithm Portfolios [GS01]. The idea was to construct a portfolio by combining multiple algorithms with different strengths and weaknesses, and running them either in parallel or interleaving on a single processor. Another form of portfolio construction was also proposed in the same work, which involved implementing an algorithm "restart" strategy. This approach involved running a series of short, sequential and scheduled runs of an algorithm on a function, such that each run is provided with a different seed value. The goal was to study algorithm performance in different runs and use the insights for designing better heuristic algorithmic strategies.

2.3 Portfolio Selection based on Domain Knowledge

The next major breakthrough occurred in 2003 with Portfolio Approach to Algorithm Selection (Leyton-Brown et. al., 2003) [Ley+03]. This work proposed exploiting domain knowledge of problems/objective functions to extract certain features of the problem instances that would most appropriately be able to indicate the runtime of an algorithm on those instances. This knowledge was used to build a regression model to learn a real-valued function that could map features to the runtime of an algorithm. Building a portfolio by this approach involved training the model for each algorithm, predicting the respective runtimes and subsequently selecting the algorithm expected to run fastest. The major drawbacks of this approach were the time and resources required for computing features and training the regression model on each algorithm.

2.4 Advanced Methods for Portfolio Construction

In the coming years, several other newer methods were proposed for portfolio construction, of which many approaches explored parallel implementations of portfolios. ParamILS: An Automatic Algorithm Configuration Framework [Hut+09] proposed setting of parameters of a given target algorithm such that the performance over a set of problem instances is maximized. Subsequently, automated configuration of algorithms and combining them into portfolios was proposed in Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection [XHL10], ISAC - Instance-Specific Algorithm Configuration [Kad+10] [Ker+19].

Although our work is inspired by the above-mentioned methods in this subsection, unlike these methods, we do not attempt to compute features of instances of problems to automatically determine optimal parameter settings of a target algorithm. Instead, we aim to construct a portfolio for anytime performance, such that given a set of objective functions, our portfolio algorithm/parameter setting(s) would be expected to perform well on a maximum number of functions in order to optimize the overall performance. Our work adopts a reverse strategy as compared to the above-mentioned approaches in the sense that we build a parallel portfolio first and then find parameter settings of the portfolio algorithm to further enhance algorithm performance, since it is not known currently how to perform selection and configuration simultaneously.

Chapter 3

Background

Over the years, tools for analyzing and comparing performance of algorithms on numerical optimization problems have become more complex and advanced. IOHprofiler [Doe+18] is one such tool used for comparing iterative optimization heuristics. It enables a detailed statistical evaluation of the performance of algorithms through distribution of fixed-target running time and fixed-budget function values. This chapter discusses the tools we used for our experiments along with functions that were optimized, optimizer algorithms used to optimize those functions, algorithms and metrics used for performance analysis in detail.

3.1 IOHprofiler

We have made extensive use of the functionalities provided by IOHprofiler [Doe+18] for our experiments. It is an open-source Python library which contains several tools for analysing running time data of an optimization algorithm and statistical evaluation of algorithm performance. Particularly, IOHprofiler library provides implementation of objective functions, wrapping functionality for converting user-defined functions or functions implemented in other platforms to IOH compatible functions, storing relevant performance data of an optimization algorithm etc. It also enables interactive evaluation, which allows the user to choose the ranges and the precision of the displayed data according to his/her needs.

3.2 YABBOB Benchmark Suite

We used numerical optimization functions from the YABBOB (Yet Another Black-box Optimization Benchmark) [Ben+21] optimization benchmark suite provided by Nevergrad [RT18] to perform experiments. The YABBOB suite resembles similar benchmark suites for derivative-free numerical black-box optimization such as the BBOB suite of the COCO platform [Han+21]. A black-box function is a function whose characteristics, features and constraints are not known before-hand. More specifically, we used the 10-dimensional instances of the YABBOB suite functions to be minimized, in a maximum of 5000 function

Function ID	Objective Function	Property
25	HM	Newly proposed function for Nevergrad
26	Rastrigin	Highly multimodal, Minima regularly distributed
27	Griewank	Has widespread but regularly distributed local minima
28	Rosenbrock	Unimodal, global minimum lies in a parabolic valley
29	Ackley	Visualized as a flat outer region and a large hole at the centre
30	Lunacek	Has multiple minima
31	Deceptive multi-modal	Highly rugged, global minimum may vary
32	Buche-Rastrigin	Multimodal, multiple local optima placed asymmetrically
33	Multipeak	Function value can never go over 10000
34	Sphere	Convex, unimodal, continuous and visualized as curvature of a sphere
35	Double Linear slope	Function value is absolute sum of components of the input variable
36	Cigar	Valley shaped (similar to a cigar)
37	Alt-Cigar	Similar to Cigar function, variables in inverse order
38	Ellipsoid	Unimodal, ill-conditioned with smooth irregularities
39	Alt-ellipsoid	Unimodal, condition number is 100
40	Discus	Globally quadratic, has local irregularities
41	Bent-Cigar	Ill-conditioned
42	Deceptive ill-conditioned	Different powers function, ill conditioned
43	Deceptive path	Sharp-ridge function, gradient towards the ridge does not flatten out

Table 3.1: YABBOB Benchmark suite functions, their respective function IDs in IOH and properties

evaluations/budget per-run for the experiments and analysis of the results. The objective functions along with their respective function IDs in IOH and properties are summarized in Table 3.1.

The functions mentioned in Table 3.1 have been implemented in Nevergrad but were not used directly from Nevergrad for our experiments. We used Nevergrad's ArtificialFunction library to create instances of the functions and then wrap them to functions compatible with IOH using IOH's $wrap\_real\_problem()$ [Doe+18] method. This method simply takes in an instance of the Nevergrad implementation of an objective function and converts it to IOH compatible form.

3.3 IOH Logger

In order to store the results of our experiments to analyze them later, we made use of the logger provided by IOHexperimenter library [Doe+18]. Although several types of loggers are available within the library, we have used the default logger for our experiments. It keeps a record of the total number of function evaluations performed on an objective function by an optimizer and the function value obtained by the optimizer at each evaluation. The default logger is compatible with IOHanalyzer [Wan+20]. This enables us to analyze the stored results using IOHanalyzer, which is an efficient and powerful tool for visualizing algorithm performance over different functions and analysing various statistics related to the performance.

3.4 Optimizers

Nevergrad [RT18] offers a variety of optimizers, each with their own strengths and weaknesses under different circumstances. To use a Nevergrad optimizer, it must be instantiated with the dimensionality of the objective function and the budget for which the function is to be evaluated. The optimizer algorithms that we used for building our portfolio are discussed in detail in this section.

3.4.1 CMA: Covariance Matrix Adaptation Evolutionary Strategy

As the name suggests, CMA is an evolution strategy [Han16]. It is most suitable for optimizing ill-conditioned functions. Condition number measures change in output of an objective function for a small change in input [Wik22a]. If a function has a high condition number, then it is an ill-conditioned function. In each iteration of an evolution strategy, the population of candidate solutions is updated by the means of mutation and recombination operations, and a new population is created by selecting only the fittest candidates from the updated candidates. The covariance matrix specifies the covariance (dependency) between pairs of candidate solutions. CMA primarily aims at guiding its search for the optimum in the search space by learning from and exploiting the information about the covariance matrix in each iteration. It does not rely on the derivatives of the objective function, hence it is suitable for optimizing black-box functions. It is an exemplary choice for non-noisy environments and large budgets ($budget \sim 1000 * dimension$) .

3.4.2 Shiwa

Shiwa [Liu+20] is a Nevergrad optimizer algorithm that selects another algorithm for optimization based on competence map. Competency mapping identifies the strengths and weaknesses of an optimizer under given circumstances, therefore the aim is to select the most suitable algorithm based on the nature of the objective function to be optimized

and the resources available to optimize it. For the settings used for our experiments, i.e.; $num\_workers = 1$ and $dimension > 1$, objective function is continuous and not noisy; then Shiwa would select CMA algorithm for optimizing the function, if the evaluation budget is greater than 6000. For the same settings but budget less than 6000, it would select $ChainCMAPowell$ [RT18] as the optimizer.

Chain CMA-Powell

Chaining in Nevergrad consists of running algorithm-1 during T1 function evaluations, then algorithm-2 during T2, then algorithm-3 during T3, etc. Each algorithm is fed with the solutions found by the previous algorithm, hence it begins its search from the point where the previous algorithm's search ended [RT18]. The algorithm $ChainCMAPowell$ in Nevergrad runs CMA in the first half function evaluations and Powell [Pow64] in the later half.

3.4.3 Particle Swarm Optimization

Particle swarm optimization (PSO) [KE95] is an optimization method that mimics the movement of birds in a flock or a swarm of fish swimming together. Every candidate solution in the population is referred to as a particle, and each particle is attributed a position in the search space as well as a velocity with which the particle moves in the search space as the search progresses. The entire swarm of particles is also attributed with a position and velocity. The idea is to guide the search for optimum, by exploiting the information about the best-known positions of the particles as well as the entire swarm, in order to move the other particles towards these 'better' positions in the search space.

3.4.4 Differential Evolution (DE)

Differential evolution (DE) [SP97] is a population-based method that functions purely like an evolution strategy. It involves a population of candidate solutions, which are updated over iterations by means of mutation and recombination operations and are used to generate a new population consisting of better solutions. The property of DE that separates it from other evolution strategies is its mutation scheme, known as differential mutation [Qin10]. In this mutation scheme, a new parameter vector is created by adding the weighted difference vector between two individuals in the population to a third individual. DE can also be moulded into specialized variants by the means of modifications based on the problem requirements.

3.4.5 Nelder-Mead

Nelder–Mead [NM65] in n dimensions maintains a set of $n + 1$ test points arranged as a simplex [Wik22c]. The $n + 1$ test points are represented by the vertices of the simplex

in the search space. Mathematical transformations are then applied to these vertices in order to decrease the value of the objective function. The process repeats in each iteration until a termination criterion is met. The termination criterion can be either the simplex becoming small enough or the vertices coming close enough to each other.

3.4.6 Naive-Iso-EMNA or EMNA

EMNA stands for Estimation of Multivariate Normal Algorithm and it is a stochastic, derivative-free method for numerical optimization of non-linear or non-convex continuous optimization problems [TBC16]. It samples new candidate solutions from a multivariate normal distribution function, built using a subset of solutions from the previous generation. For our experiments, we use the Nevergrad implementation of EMNA, also referred to as NaiveIsoEMNA wherein $Naive$ signifies a Boolean variable that is set to $True$ while dealing with a non-noisy problem and to $False$ otherwise, and Iso signifies the Isotropic version of EMNA. The *isotropic* variable being set to $True$ implies that the Gaussian distribution matrix is an identity matrix [RT18].

3.4.7 Diagonal CMA

Diagonal CMA is an improvement over CMA which makes use of adaptive diagonal decoding [AH19]. The covariance matrix C is represented in the form of DCD, where D is a diagonal matrix that represents the coordinate-wise variances of the sampling distribution. The diagonal matrix can learn a rescaling of the problem in the coordinates within linear number of function evaluations. Diagonal-CMA can exploit the separability of the problem without compromising performance on the non-separable problems.

3.4.8 EDA: Estimation of Distribution Algorithm

Estimation of Distribution Algorithm [Pel05] is an evolutionary algorithm that, unlike traditional evolutionary algorithms, makes use of an explicit distribution model instead of crossover and mutation operators. In EDA, an initial population P is generated, and the fitness of each individual in the population is evaluated. A probabilistic model of P is built using the candidates with better fitness values. In each iteration, a new set of candidates is sampled from the model obtained after the previous iteration and the process continues until a termination criterion is fulfilled [MG21]. The Nevergrad implementation of EDA comes with a warning that states the implementation might be incorrect.

3.4.9 NGO

NGO stands for Nevergrad Optimizer (also known as NGOpt) and the algorithm is similar to Shiwa in the sense that it is a competence map algorithm which selects another appropriate algorithm for optimizing the objective function based on experiment settings

[Meu+22]. For optimizing a function which is fully continuous and not noisy, NGO chooses an algorithm from among One Plus One, Chained CMA-Powell, Differential Evolution and CMA. For the condition $dimension > 1$ and $500 < budget < 6000$, NGO selects CMA as the optimizer for optimizing the objective function. Nevergrad contains implementations of multiple variants of NGO/NGOpt, but for our experiments we use the NGOpt4 variant which is also the default variant for optimizing single objective functions in Nevergrad.

3.5 Nevergrad Ask and Tell interface

The ask and tell interface [RT18] is where the actual optimization of the objective function takes place. The interface provides for three key main methods.

- $ask()$: This method suggests a candidate to the optimizer on which the objective function is supposed to be evaluated.
- $tell()$: The objective function is evaluated for the candidate suggested by the $ask()$ method, and the optimizer is updated with the value of the function for this candidate using the $tell()$ method.
- $provide\_recommmendation()$: Once the maximum number of function evaluations have been performed, the $provide\_recommmendation()$ method returns the candidate that is considered best by the optimizer. This candidate is the optimum function value found within the specified number of function evaluations.

3.6 Empirical Cumulative Distribution Function (ECDF)

For our experiments, we use area under average empirical cumulative distribution function (ECDF) [Dek10] curve as a metric to measure the performance of an optimization algorithm. The value of the ECDF at any specified value of the measured variable is the fraction of observations of the measured variable that are less than or equal to the specified value [Wik22b]. In the context of our experiments, the ECDF of an optimization algorithm A for an objective function F to be minimized, given the target value v and value of the budget B (number of function evaluations), can be defined as the fraction of runs in which A is able to obtain a solution of value at least v within B function evaluations for F. In mathematical terms, the calculation of ECDF is done using the equation below:

$$ECDF(A, F, v, B) = \frac{\text{No. of runs in which } A \text{ finds a solution} \leq v \text{ within } B \text{ evaluations}}{\text{Total number of runs of } A \text{ on } F}$$

For our analysis, we have calculated the average ECDF given a set of target values V instead of the regular ECDF given a single target value. The calculation of the average

ECDF for an optimizer A run on an objective function F to be minimized, given a set of target values V and budget B is represented by the equation below:

$$ECDF_{avg}(A, F, V, B) = \frac{\sum_{\forall v \in V} ECDF(A, F, v, B)}{\text{Total no. of elements in } V}$$

3.6.1 Area under average ECDF curve

The curve traced by the points representing the average ECDF values at each budget value or specific budget values, is defined as the average ECDF curve. The area under this curve (AUC) determines the performance of the optimization algorithm (further discussed in Section 4.1.1). The calculation of AUC of a short budget interval $B_{i-1} - B_i$ using trapezoidal rule [Wik22d] for an optimizer A run on an objective function F to be minimized, given a set of target values V and budget B is represented by the equation below:

$$AUC_{(B_i - B_{i-1})} = (B_i - B_{i-1}) \times \frac{ECDF_{avg}(A, F, V, B_i) + ECDF_{avg}(A, F, V, B_{i-1})}{2}$$

The total AUC over the entire budget B is then calculated as:

$$AUC_B = \sum_{0}^{B} AUC_{(B_i - B_{i-1})}$$

Chapter 4

Methodology

In our work, we emphasize on the concept of parallel simulation to perform experiments and build an efficient parallel portfolio for good anytime performance. This chapter elaborates on the experiments performed in our work and the methodologies adopted for these experiments in detail.

4.1 Simulation of Parallelism

Parallel approaches are widely being used in recent years to achieve effective speedup and to improve the quality of solutions obtained by optimization algorithms. With the availability of parallel computing resources, running multiple algorithms in parallel to boost overall performance has become a possibility. In our work, we propose an Empirical Cumulative Distribution Function (ECDF)-based parallel simulation technique. In this approach, to extrapolate the performance of two optimization algorithms run in parallel on a specific objective function, we construct the average ECDF for a set of target values $V = \{10^2, 10, 1, 10^{-1}, ...10^{-6}\}$, of both the algorithms and then consider only the higher ECDF values of the two, at each function evaluation. This is done in order to obtain the combined average ECDF curve of the two optimizers. Such a curve for maximum (combined) average ECDF values is depicted in Figure 4.1. This approach is equivalent to running the two optimization algorithms in parallel on a specific function and only recording the better of the two solutions found by both optimizers at each function evaluation/budget value. We define the maximum ECDF values obtained from two algorithms, as the ECDF values obtained from the combination of the two algorithms in consideration. This also enables us to take full advantage of algorithm performance complementarity when the aim is to optimize a set of objective functions with distinct features. We refer to the combined average ECDF of two optimization algorithms as $ECDF_{avg}(A_1, A_2, F, V, B)$ hereafter, where A_1 and A_2 are optimizations algorithms, F is the objective function, V is the set of target values and B is the budget. For analysis and comparison of optimizer algorithms and algorithm combinations with each other, we consider the area under this average

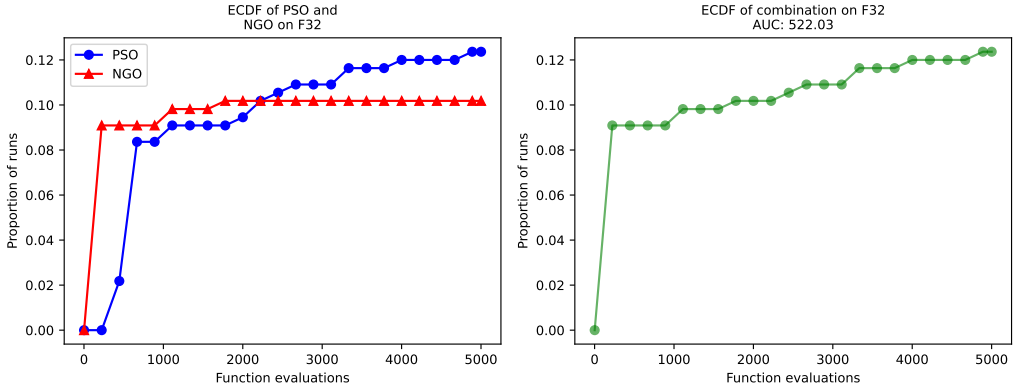


Figure 4.1: Calculation of combined $ECDF_{avg}$ values for algorithms PSO and NGO on function 32

ECDF curve as a performance metric.

4.1.1 Area under average ECDF curve as performance metric of an algorithm

We determine the quality of solutions obtained by an optimizer by comparing them against a set of specific values. This comparison is done by the means of calculation of average ECDF values at each function evaluation executed by the optimizer. The quality of solutions obtained at each function evaluation, is therefore, directly proportional to the average ECDF value at that particular function evaluation, i.e., better quality solutions correspond to higher average ECDF values. If we were to plot the average ECDF values against function evaluations/budget values, we would obtain a graph which we refer to as the average ECDF curve for an optimizer-function pair. Higher average ECDF values result in higher area under the curve. This in turn implies that the quality of solutions obtained is directly proportional to the area under the average ECDF curve for an optimizer-function pair. For our experiments, we use the *auc*() function defined in the Scikit-learn module *sklearn.metrics* [Ped+11]. It takes in as input, lists of values of the x and corresponding y coordinates (budget and average ECDF values respectively, in our case) that define the curve in a two-dimensional plane to calculate the area under that curve using the trapezoidal rule [Wik22d]. We therefore use this area under the average ECDF curve (AUC) as a performance metric of the optimizer on any function. The comparison of different optimizers on functions and overall, for experiments performed as a part of our work, is done on the basis of AUC values. The calculation of area under average ECDF curve using the *sklearn.metrics.auc*() function is described in Algorithm 1.[1]

[1]The link to github repository for code: `https://github.com/Naveen8297/portfolio_anytime_performance`

Algorithm 1 Calculating the area under average ECDF curve of algorithms A_1 and A_2 on objective function F (minimized)

$A_1 \leftarrow Optimizer_1$
$A_2 \leftarrow Optimizer_2$
$ECDF\_values \leftarrow emptyList\{\}$
$func\_eval\_list \leftarrow emptyList\{\}$
for `B in budget` **do**
    $ECDF_{avg}(A_1, A_2, F, V, B) = max(ECDF_{avg}(A_1, F, V, B), ECDF_{avg}(A_2, F, V, B))$
    $ECDF\_values.append(ECDF_{avg}(A_1, A_2, F, V, B)$
    $func\_eval\_list.append(B)$
end for
$AUC(A_1, A_2, F, V) \leftarrow sklearn.metrics.auc(func\_eval\_list, ECDF\_values)$

4.2 Hyperparameter Optimization using Irace

We performed experiments wherein we automatically tuned the parameters of optimization algorithms using *irace* [Lóp+16] package and observed their performance, with the obtained parameter values, on different functions. The parameter tuning has been done for pairs/combinations of algorithms simulated to be run in parallel as described in the previous subsection.

4.2.1 The irace package

The irace [Lóp+16] package (in R) implements an iterated racing procedure for the automatic configuration of the parameters of an optimization algorithm. The process saves the burden of manual tuning and can be used, when the performance of an optimization algorithm depends on the values of its parameters, for finding the parameter configuration which corresponds to the optimal performance. The optimization algorithm for which tuning is performed by irace is referred to as the target algorithm. The requirements for irace to be able to perform hyperparameter optimization of a target algorithm are described in the following subsections.

Parameter space definition

The parameter space definition is simply the names, range and all possible choices of different parameters for which the tuning has to be performed. This can be passed to irace as input by means of a text (.txt) file. It can also be referred to as the search space for parameters.

Executable

The executable is the program for which parameter tuning is supposed to be done by irace. In our case, the executable is a Python file in which the target algorithm(s) to be tuned is defined. It accepts configuration parameters and returns the corresponding result of the target algorithm.

Target runner

The target runner is a script which executes the executable via command line, with a specific parameter configuration passed to it as input by irace and passes on the result of execution back to irace. It acts as an interface for communication between irace package and the executable [Lóp+16].

Configuration scenario

The configuration scenario is the set of problem instances on which the parameters of the target algorithm must be tuned (trained and tested). It can be passed to irace by means of a text file.

Once irace has obtained these four requirements, it performs the following steps:

- Searches for the parameters in the defined search space
- Executes the target runner with the found parameter values, on the problem instances defined in the configuration scenario
- Evaluates the algorithm performance on those problem instances

This process continues in an iterative manner until the parameter configuration(s) corresponding to the best performance (within a specified budget) have been found.
The parameter tuning experiments described in this section are of two types: tuning for quality of solutions (target value) found by the algorithms and tuning directly for the combined average ECDF of pairs of algorithms. Both of these types of experiments are elaborated in the further subsections.

4.2.2 Parameter tuning for quality of solutions

Initially, we attempted to tune parameters for every algorithm individually and for the final function value obtained by it after a specific number of evaluations. This simply means that we attempted to make irace search for parameter configurations of the target algorithm which would correspond to the optimal solutions found by it within a specified budget. This approach appears simpler as each target algorithm is dealt with separately. However, it is not accurate; since for analysis and comparisons in our previous experiments in this paper, we made use of the area under average ECDF curve (AUC) as a performance metric. Therefore, tuning for the area under the combined average ECDF curve of two target algorithms instead of that for quality of solutions found by single algorithms is a more appropriate and sensible approach.

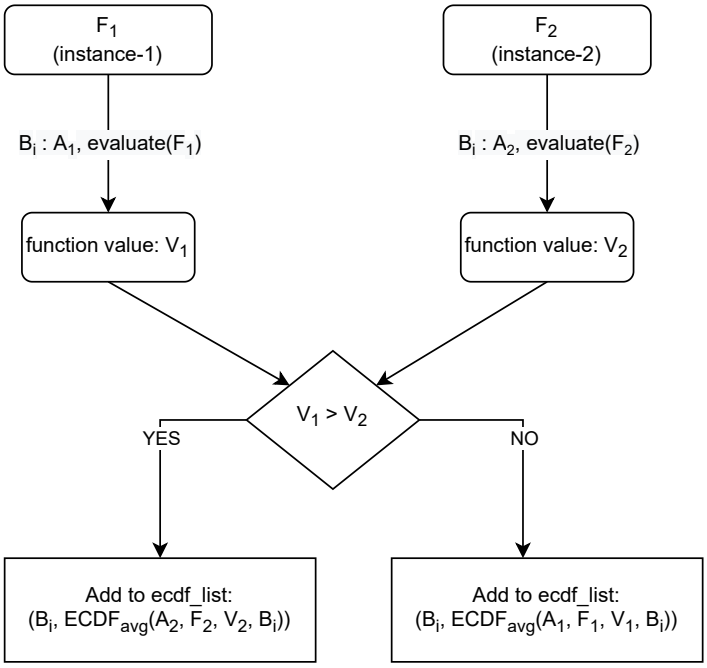


Figure 4.2: Average ECDF calculation at budget value B_i for function F in the hyperparameter tuning process for target algorithm pair (A_1, A_2)

4.2.3 Parameter tuning for area under average ECDF curve (AUC) for pairs of target algorithms

To improve the tuning process, we tuned the hyperparameters of pairs of target algorithms for the area under their combined average ECDF curve ($ECDF_{avg}$) instead of that for the respective final function values achieved by them individually. This approach has been adopted mainly for two reasons: first, since we have made use of a similar approach for quantifying the performances of the default versions of algorithms and to simulate parallelism for pairs of algorithms, and second, to keep the comparison between the performance of hyperparameter-variants and default versions of algorithms fair. In this approach, we instantiate two copies of the same objective function on which the each respective algorithm from the pair of target algorithms has to be run; in a single target runner file. We then evaluate each instance of the objective function with an algorithm from the pair of algorithms. Then, we calculate the average ECDF $ECDF_{avg}$ at each budget value until we have used up all function evaluations in the budget. The flowchart depicted in Figure 4.2 illustrates the working of this approach at budget value B_i, for function F (2 instances of F: F_1 and F_2) and algorithm pair (A_1, A_2).
In other words; for each budget value, we perform an evaluation on both instances of the function with respective algorithms, and only use the lower of the two target values (in case of minimization) to calculate the average ECDF for the combined algorithm performance. Once we have the average ECDF values corresponding to all function evaluations in the budget, we simply calculate the AUC and pass it to irace as result.

Tuning for overall performance across functions

For tuning against area under average ECDF curve, initially we attempted to tune the hyperparameters of optimizers to achieve improved overall performance across all 19 objective functions. In this approach, the objective functions are passed on to irace as instances in the configuration scenario. This implies that the resultant configurations returned by irace after the tuning process would be those which irace considers would perform best across all the objective functions passed to it as instances.

Tuning for performance on each objective function individually

We further attempted to tune the optimizers for performance on each objective function separately. This approach involves passing the objective functions to irace as categorical parameters, either via command line or directly in the R file, instead of instances. Since the objective function is passed on as a categorical parameter, for each objective function, irace explores the configuration space for "good" configurations that are expected to perform well on that specific function.

Chapter 5

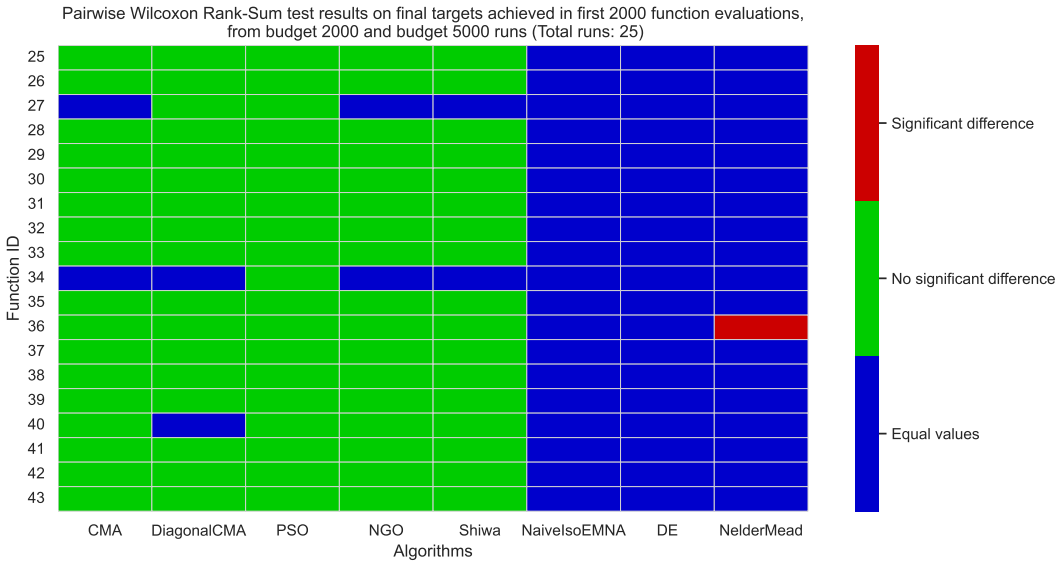
Results

5.1 Impact of budget on algorithm performance

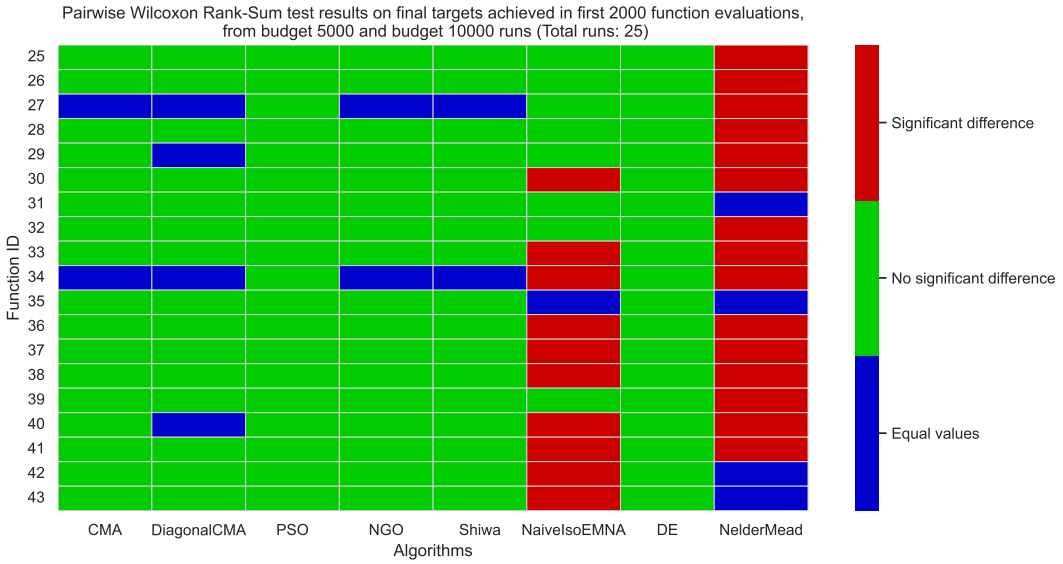
As a part of our preliminary experiments, we check the impact of budget on the performance of optimization algorithms. This is essential since we aim at optimizing the anytime performance of algorithms, which implies that at any given time, our portfolio algorithm must be able to achieve a good performance irrespective of the budget it will be run for. We execute 25 runs each for budgets: 2000, 5000 and 10000 for algorithms: CMA, DiagonalCMA, PSO, NGO, Shiwa, NaiveIsoEMNA, DE and NelderMead; and then compare for each algorithm, the different budget-runs with each other. This comparison was carried out to determine whether the search trajectory of the optimization algorithms differs or not with varying budget. The set of 25 target values obtained after the first fe function evaluations, for algorithm A and budget B can be denoted as: $TV_{(fe,B)}(A)$. Therefore, to make the comparisons for all algorithms, we performed the Pairwise Wilcoxon Rank-Sum test [Wil45] on:

- $TV_{(2000,2000)}$ and $TV_{(2000,5000)}$
- $TV_{(2000,5000)}$ and $TV_{(2000,10000)}$
- $TV_{(2000,2000)}$ and $TV_{(2000,10000)}$
- $TV_{(5000,5000)}$ and $TV_{(5000,10000)}$

We also applied Bonferroni p-value corrections [BA95] and the corrected α for our experiments stands at 1%. The results for the Pairwise Wilcoxon Rank-Sum tests are depicted in Figures 5.1 and 5.2. The label "Significant difference" indicates that there is a significant difference between corresponding values in sets $TV_{(fe_i,B_i)}$ and $TV_{(fe_j,B_j)}$. Similarly, the label "No significant difference" indicates that there is no significant difference between corresponding values in sets $TV_{(fe_i,B_i)}$ and $TV_{(fe_j,B_j)}$, and the label "Equal values" indicates the corresponding values in $TV_{(fe_i,B_i)}$ and $TV_{(fe_j,B_j)}$ are equal.

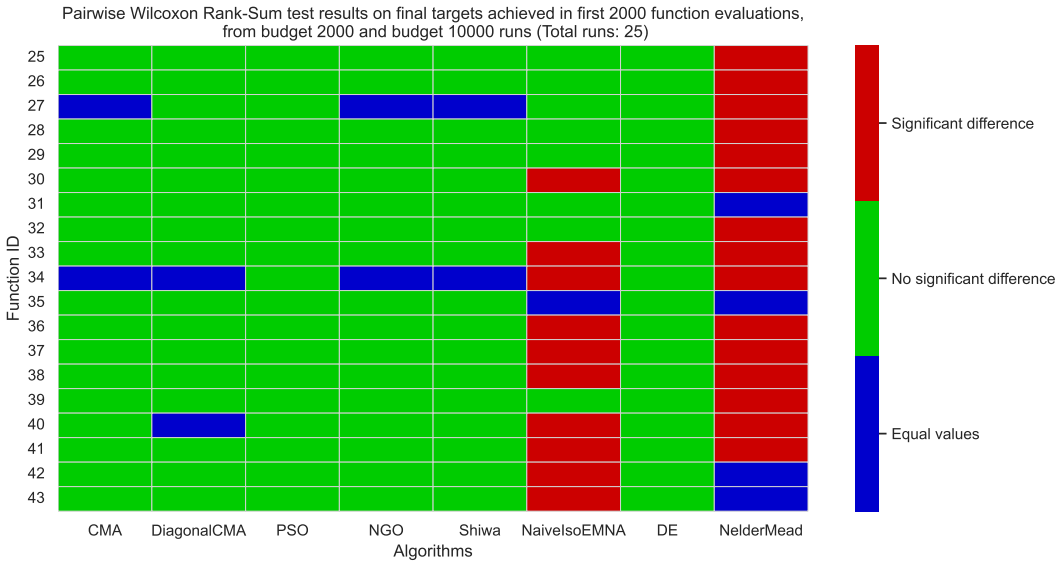


(a) $TV_{(2000,2000)}$ and $TV_{(2000,5000)}$

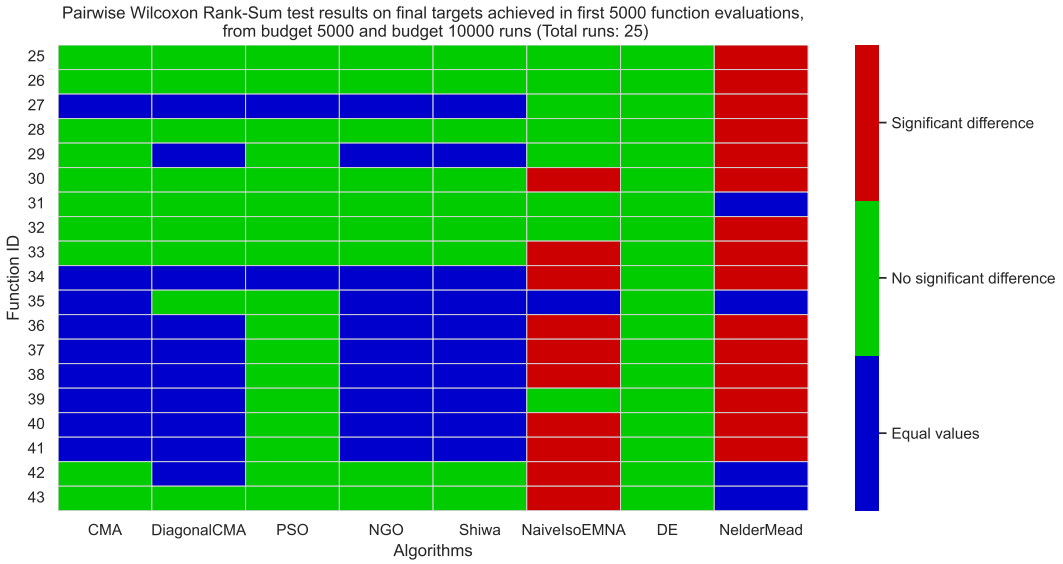


(b) $TV_{(2000,5000)}$ and $TV_{(2000,10000)}$

Figure 5.1: Pairwise Wilcoxon Rank-Sum test results for $TV_{(2000,2000)}$, $TV_{(2000,5000)}$ and $TV_{(2000,5000)}$, $TV_{(2000,10000)}$



(a) $TV_{(2000,2000)}$ and $TV_{(2000,10000)}$



(b) $TV_{(5000,5000)}$ and $TV_{(5000,10000)}$

Figure 5.2: Pairwise Wilcoxon Rank-Sum test results for $TV_{(2000,2000)}$, $TV_{(2000,10000)}$ and $TV_{(5000,5000)}$, $TV_{(5000,10000)}$

From Figure 5.1(a), we can infer that after 2000 functions evaluations on functions 27 and 34, the set of 25 target values achieved by CMA in 25 respective runs for budget 2000 is exactly equal to the set of 25 target values achieved by CMA after 2000 function evaluations, but for budget 5000.

Figure 5.1(b) and 5.2(a) depict that, for all of the algorithm-function pairs except those of NaiveIsoEMNA and NelderMead, the set of target values for runs of a particular budget is either exactly similar to or has no significant difference from the set of target values for runs of another budget. When the sets are exactly equal, it means that the algorithm has found the optimal solution for that function and the target values have converged.

From Figure 5.2(b) it is clear that the target values converge for certain function-algorithm combinations (except those of NaiveIsoEMNA and NelderMead) after 5000 function evaluations. For NaiveIsoEMNA and NelderMead it cannot be concluded that the smaller budget runs be considered as truncation of the respective higher budget runs. The clear cause for this is not known. However, for the remaining algorithms, we can conclude that the smaller budget runs be considered as truncation of the respective higher budget runs.

The results depicted above show that the search trajectory/search path for solutions of optimization algorithms, except for NaiveIsoEMNA and NelderMead, is not affected by the budget which they are run for. The finding is important since it is now known for which algorithms, the anytime performance will not be affected by budget. The results also depict that the solutions found by algorithms stagnate at higher budgets, i.e., they do not improve with further evaluations. The fact that the solutions stagnate leads us to choose an appropriate budget value $budget = 5000$ for our further experiments where we check the performance of parallel portfolio algorithms.

5.2 Performance analysis of algorithm combinations

Our subsequent experiments entail an analysis of algorithm combinations with simulated parallelism as described in Chapter 4. Performance metric used is the area under average ECDF curve (AUC). Experiments for all optimizers were performed for 25 runs each, with each run lasting up to a total of 5000 function evaluations ($budget = 5000$) on every objective function. Therefore, the AUC values have been averaged over 25 runs. Figure 5.3 demonstrates the performance of algorithm combinations along-with that of single algorithms in terms of normalized AUC values. All AUC values have been normalized between 0 and 1 by dividing them by the maximum possible AUC value, i.e., 5000. It is to be noted that the performance of single algorithms depicted is actually the performance of combinations of the same algorithm(s), with both instances run independently and

then combined after simulating parallelism. Hence, for the single algorithms, the AUC values correspond to 2 independent sets of 25 runs each for 2 respective instances of the same algorithm. For the results discussed in Section 5.2 and 5.3, all algorithms and their respective combinations listed are the default Nevergrad versions and the above-mentioned experiment settings have been used.

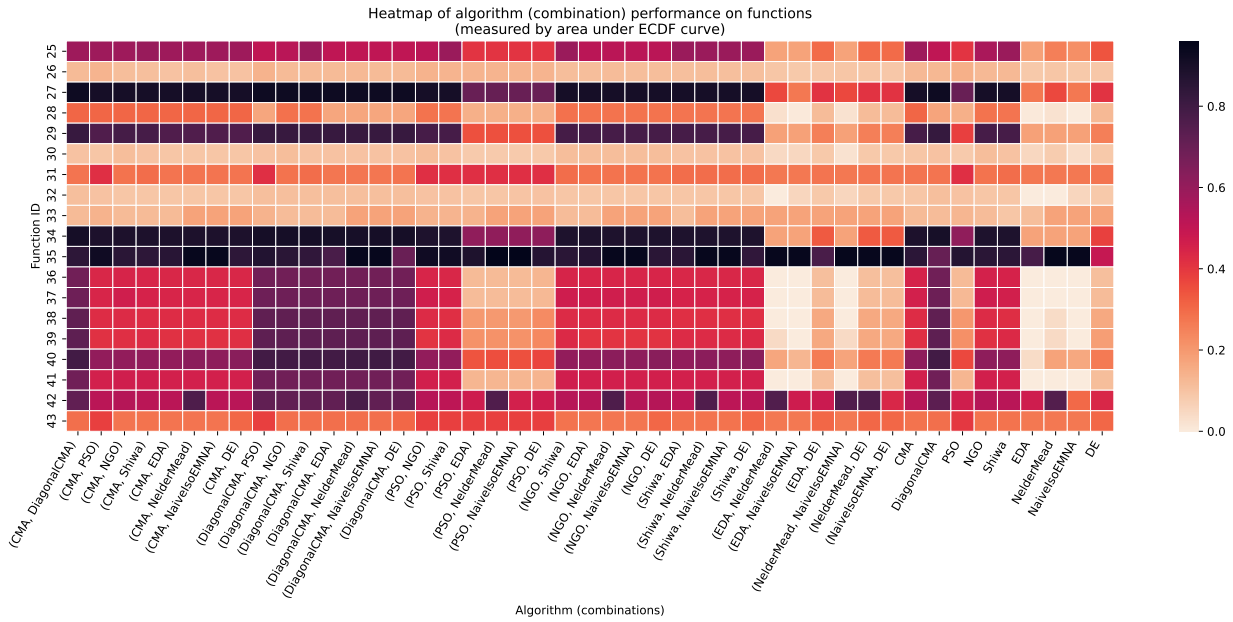


Figure 5.3: Performance of algorithm combinations in terms of normalized AUC values, averaged over 25 runs

From Figure 5.3, we interpret that optimizers EDA, NaiveIsoEMNA and NelderMead are the worst performing algorithms for every function. DE performs slightly better than the aforesaid algorithms but is still one of the worst performing algorithms overall. Consequently, the combinations (EDA, NelderMead), (EDA, NaiveIsoEMNA), (EDA, DE), (NelderMead, NaiveIsoEMNA), (NelderMead, DE) and (NaiveIsoEMNA, DE) are the worst performing combinations. PSO performs better than DE and hence its combinations with the worst performing single algorithms (EDA, NaiveIsoEMNA, NelderMead and DE) perform better than the worst performing combinations mentioned above. On the other hand, DiagonalCMA and all of its combinations emerge as the best performing combinations overall, outperforming all other combinations by a considerable margin for particularly for functions 36-42.

5.3 Inspection for improvement over single algorithms

5.3.1 Performance comparison of algorithm combinations with constituent algorithms, for each function

Initially we inspected for improvement by comparing the performance of an algorithm combination/pair with that of the better performing algorithm out of the two, for each function. In other words, to check for improvement of the algorithm pair (A_1, A_2) on an objective function, it will be compared to either A_1 or A_2 depending on which of the two performs better independently on the same objective function. Figure 5.4 depicts this comparison by means of a heatmap. To depict improvement, the absolute AUC values of the combinations have been divided by the AUC values of the respective better performing constituent algorithms. As mentioned in Section 5.2, every single algorithm depicts comparison between the combination of two independent instances of that algorithm, with the better performing instance of the two for each function.

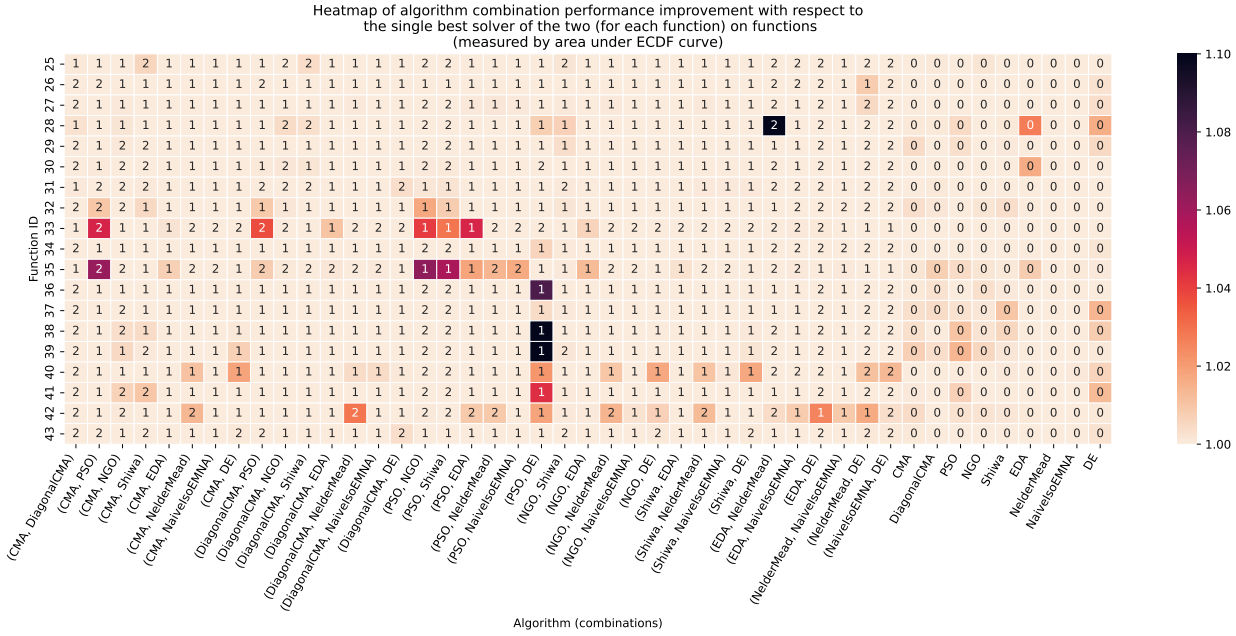


Figure 5.4: Performance improvement of algorithm combinations over respective better performing constituent algorithms for each function
*The number in the cells indicates the better performing constituent algorithm/single best solver

The improvement depicted in Figure 5.4 can also be termed as the comparison between the portfolio algorithm (algorithm combination) and the better performing constituent algorithm, for each objective function. The cells labeled with 1 indicate that for the corresponding function and algorithm pair, the first listed algorithm is the better performing constituent solver. Similarly the cells labeled 2 indicate that the second listed algorithm is the better performing constituent solver. The cells corresponding to single algorithms are all labeled 0 because in each such case, the better performing solver is the same algorithm.

The improvement does not seem significant from this figure, since the comparison depicted is a hard-line one.

5.3.2 Performance comparison of algorithm combinations with constituent algorithms, across all functions

To obtain a clear and concise picture of performance improvement of algorithm combinations over single algorithms, we resort to their comparison overall across all functions instead of for each function. In other words, we depict improvement of the portfolio algorithms with respect to the respective better performing constituent solvers for performance aggregated across all functions. The method adopted to depict this improvement is same as that used in the previous subsection, i.e., dividing the aggregated (summation of) AUC value across all functions for the portfolio algorithm by that of the better performing constituent solver. Figure 5.5 depicts this improvement. The cell labels indicate the better performing constituent solver. All algorithms and their respective combinations listed are the default Nevergrad versions.

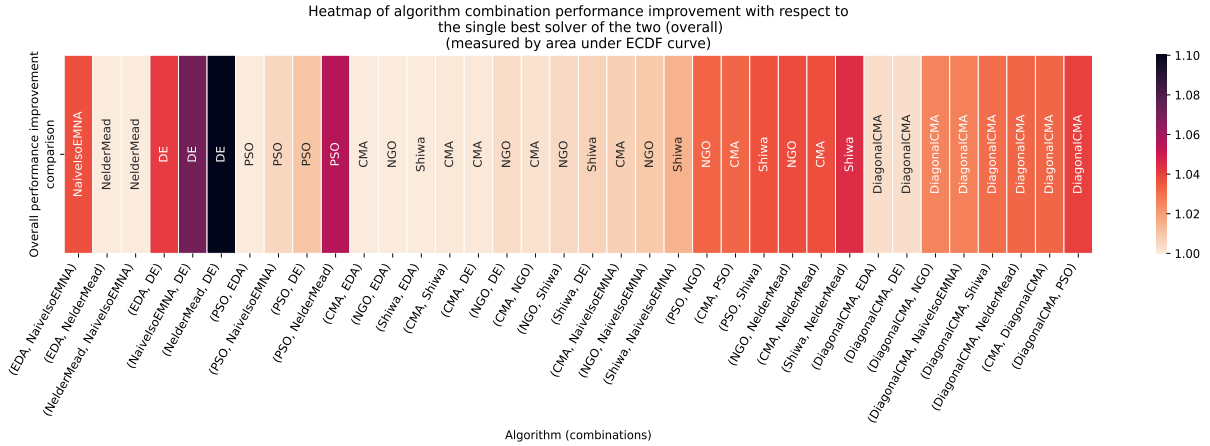


Figure 5.5: Performance improvement of algorithm combinations over respective better performing constituent algorithms across functions

*The algorithm combinations have been sorted in ascending order according to the area under average ECDF curve

**The algorithm name mentioned in the cells indicates the better performing constituent algorithm/single best solver

From Figure 5.5, we can infer that the performance improvements of algorithm combinations over single algorithms are in the range of 0 to 10%. Certain algorithms, when coupled with either PSO or Diagonal CMA exhibit significant performance improvement.

To obtain an even more clearer picture of the performance of algorithm combinations across functions, we created a bar plot of the aggregated performance of each algorithm and algorithm combination. To create this bar plot, we simply add the normalized AUC

values from Section 5.2 across all functions, arrange the algorithms in ascending order according to these aggregated normalized AUC values, and plot them. The plot depicted in Figure 5.6 can also be termed as that depicting the ranking of algorithms and algorithm combinations. Higher lengths of bars in the bar plot correspond to better performance of the respective algorithms/algorithm combinations. All algorithms and their respective combinations listed are the default Nevergrad versions.

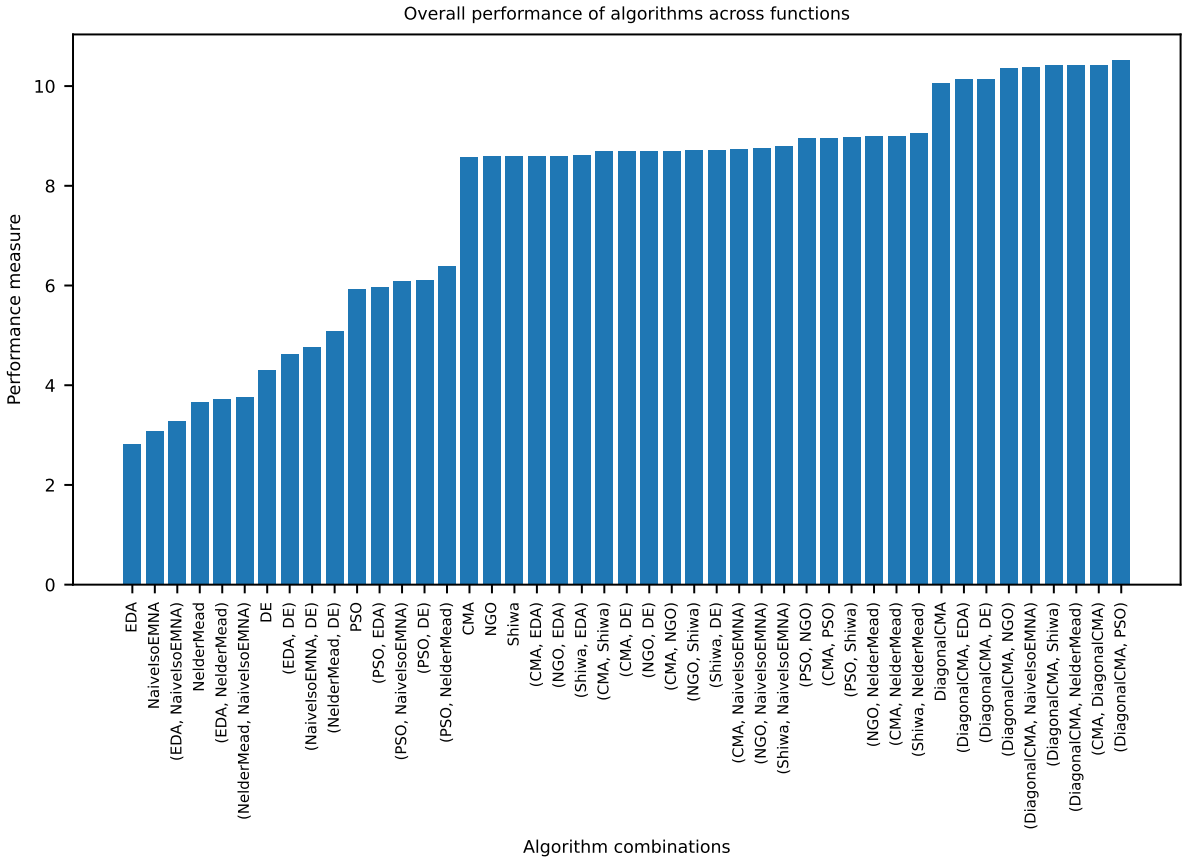


Figure 5.6: Algorithm and combination performance bar plot

From Figure 5.6, we can infer that the combination (DiagonalCMA, PSO) outperforms every other algorithm and algorithm combination and can be termed as the best performing portfolio algorithm. We can also infer that the best performing single algorithm is Diagonal CMA and the best performing algorithm combinations are those which have Diagonal CMA as a constituent solver.

5.4 Performance of algorithm combinations after hyperparameter tuning

For performing experiments with irace, we chose two optimization algorithms offered by Nevergrad, namely; Diagonal CMA and Particle Swarm Optimization (PSO), since their

Target algorithm	Parameter	Default	Search domain type	Search space
DiagonalCMA	_popsize	8	Closed interval, integer	[5, 30]
	_random_init	False	Categorical, boolean	(True, False)
	_scale	1.0	Closed interval, real	[1.0, 5.0]
PSO	llambda	40	Closed interval, integer	[30, 50]
	_transform	'arctan'	Categorical, string	('arctan', 'identity', 'gaussian')*

Table 5.1: Target algorithms, parameters chosen for tuning, their default values and search space with respective domain types
*Transforms corresponding to $\_transform$ parameter are implemented in Nevergrad and are used based on the given string values

combination outperformed every other algorithm combination. The parameters we chose for each of these algorithms, their default values in the Nevergrad implementation, the type of the search space domain, and the parameter space definition we passed to irace for tuning are summarized in Table 5.1.

5.4.1 Tuning for overall performance across functions

As discussed in Section 4.2.3, we first present the results of tuning hyperparameters of algorithm combinations for AUC values, aggregated across all functions. The aim was to search for configurations that would yield maximal overall performance. For experiments described in this section, we chose the (DiagonalCMA, PSO) combination for tuning as it turned out to be the best performer in the experiments described in the previous section. The data used for this analysis was obtained from experiments performed for 25 runs, for $budget = 5000$, for each configured variant of the combination. Figure 5.7 depicts the AUC values, normalized between 0 and 1 by dividing by 5000, for each function-configured combination variant pair. The configured variants are listed in the format (DCMA_{_popsize}_{_scale}_{_random_init}, PSO_{llambda}_{_transform}). The combination (DiagonalCMA, PSO) represents the default Nevergrad version(s) of both DiagonalCMA and PSO. In total, we have obtained 8 elite configurations from 8 respective independent runs in irace.

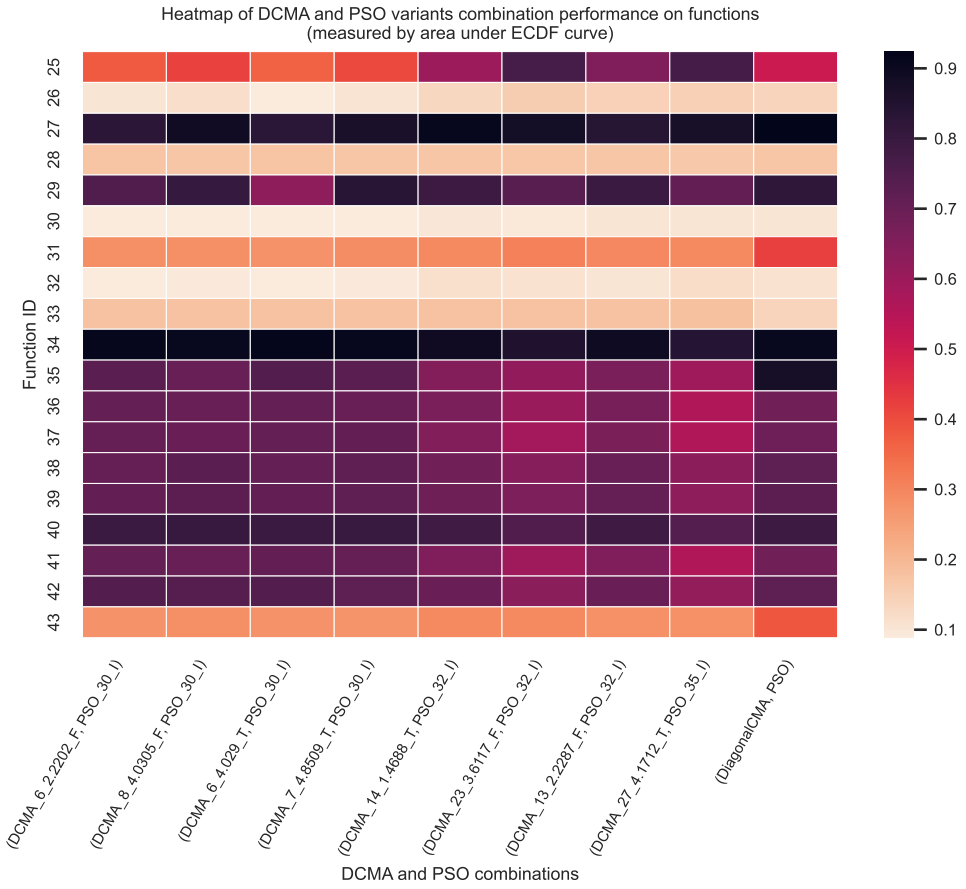


Figure 5.7: Performance of DiagonalCMA and PSO variant combinations in terms of normalized AUC values, averaged over 25 runs
*The default variant combination is listed as (DiagonalCMA, PSO)

From Figure 5.7, it is clear that the default variant combination i.e., (DiagonalCMA, PSO) performs better than the configured variant combinations on functions 31, 35 and 43. However, which combination performs the best overall cannot be clearly determined from the figure. Therefore, to further enhance the results presented in Figure 5.7, we carry out a different type of normalization wherein the AUC value for each function-optimizer combination pair is divided by the AUC value for the same function but (DiagonalCMA, PSO) combination, i.e., the default variant combination. This method is appropriate since it enables easier comparison of performance of configured variant combinations with the performance of default variant combination, for each function. Figure 5.8 depicts this comparison.

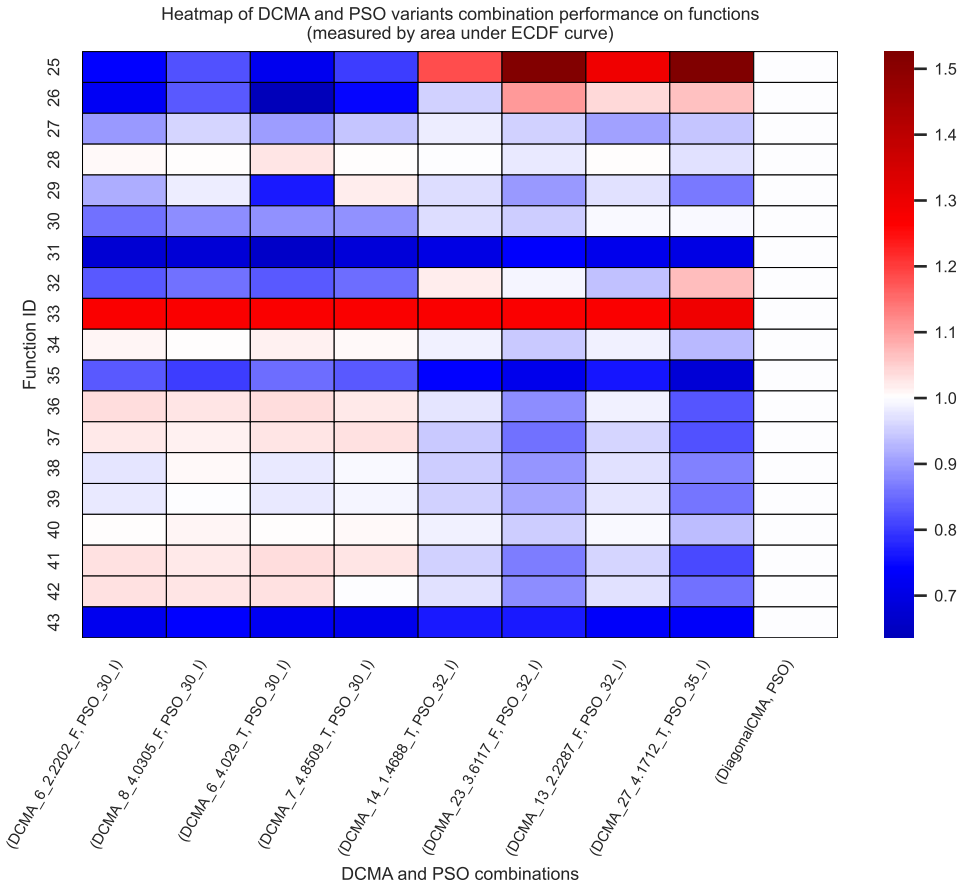


Figure 5.8: Performance of DiagonalCMA and PSO variant combinations in terms of normalized AUC w.r.t. default variant's (DiagonalCMA, PSO) AUC values (for each function), averaged over 25 runs

From Figure 5.8, we can infer that no configured variant combination performs better than the default variant combination on all functions. All configured variant combinations perform better than the default variant combination only on function 33 (Multipeak function) whereas all of them perform worse than the default on functions 31, 35 and 43 (Deceptive-multimodal, Double-linear slope and Deceptive-path functions, respectively).

In Figure 5.9, we present a bar plot similar to Figure 5.6 for all variant combinations, based on aggregated normalized AUC values across all functions (normalization between 0 and 1 for each function, as depicted in Figure 5.7).

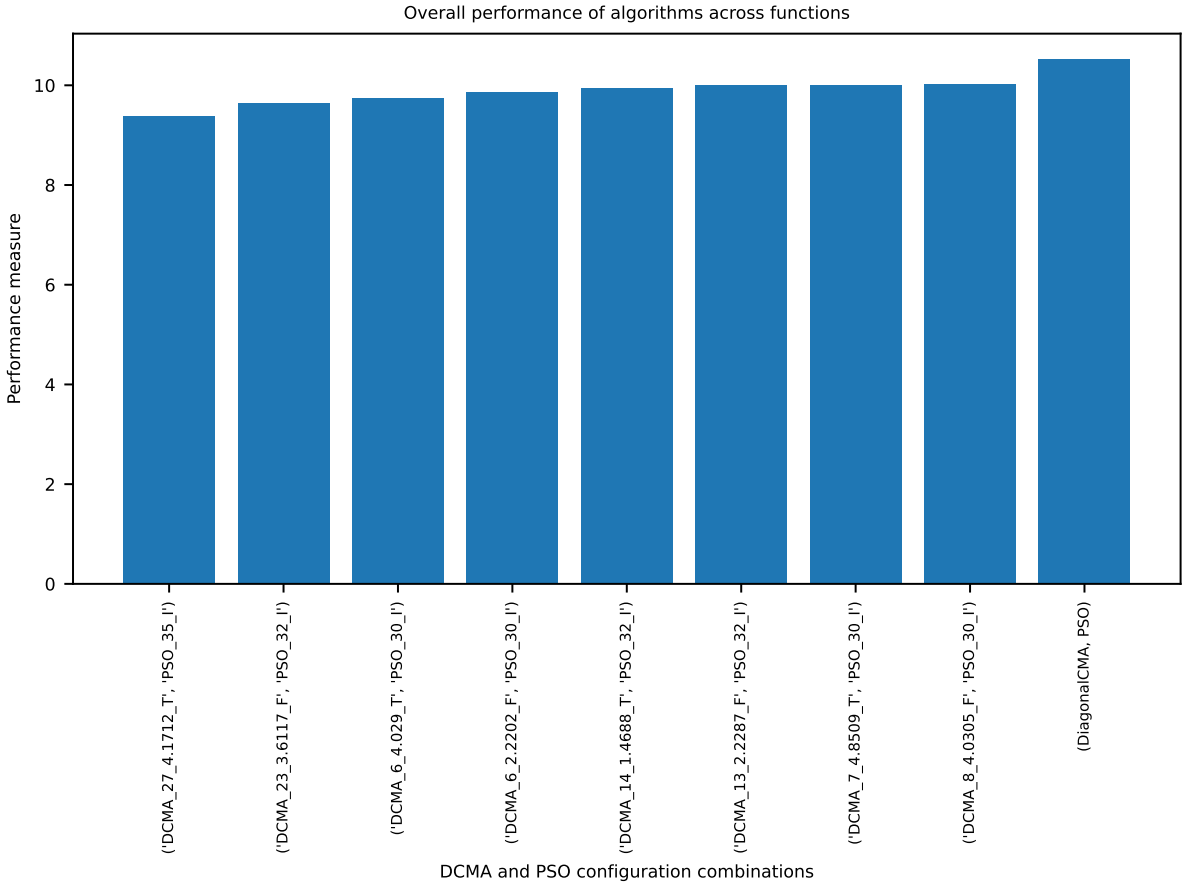


Figure 5.9: Bar plot of aggregated normalized AUC values across functions for configured and default variant combinations

It is clear from the bar plot that all configured variant combinations are outperformed by the default variant combination by a small margin. From Figures 5.8 and 5.9, we can infer that the best performing configured variant after the default variant, i.e., (DCMA_8_4.0305_F, PSO_30_I), outperforms the default variant slightly on functions 36, 37, 38, 40, 41 and 42. However, it performs worse than the default variant on functions 25-27 and 29-32 along with 35 and 43. Whereas, the configured variants ((DCMA_23_3.6117_F, PSO_32_I), (DCMA_27_4.1712_T, PSO_35_I), (DCMA_13_2.287_F, PSO_32_I)) that perform better than the default variant on functions 25 and 26, perform worse than the default variant on functions 36, 37, 38, 40, 41 and 42. We can also see that the configurations can be classified into two types based on value of the parameter *_popsize*: *_popsize* $<= 8$ and *_popsize* > 10. Performance on functions 36, 37, 38, 40, 41 and 42 is crucial in determining the configurations with good anytime performance, and configurations with *_popsize* $<= 8$ can be seen as satisfying this criterion. On the other hand, configurations with *_popsize* > 10 exhibit improved performance on functions 25 and 26.

It is worthy to note that the configuration space in this type of tuning approach was significantly larger as compared to the approach discussed in the further subsection, which could have been one possible reason for *irace* not being able to find a better performing

configuration setting than the default one. A deeper configuration search by *irace*, with an increased number of iterations, could have been performed and possibly ended with a desirable configuration setting; but the tuning process would then be tedious without any guarantee of the desirable result being produced.

5.4.2 Tuning for performance on each objective function individually

We further attempted to tune hyperparameters for performance for each function separately. We again chose the optimizer combination (DiagonalCMA, PSO) for tuning, since it is the best performing combination in its default variant. For this approach, we analyse the performance of 19 elite configured variants (corresponding to 19 respective functions), along-with the performance of the default variant, on all objective functions. In other words, we present the performance of 19 elites+1 default=20 variant combinations on 19 objective functions in the form of a heatmap. Performance for experiments described in this subsection has been measured based on AUC values, averaged over 5 runs ($budget = 5000$ for each run) for each function-variant combination pair. The configured elite variants are listed in the format (DCMA_{_popsize}_{_scale}_{_random_init}, PSO_{llambda}_{_transform})_{function-ID} wherein {function-ID} corresponds to the function for which *irace* considered the configuration setting to be elite. (In other words, the name of the variant which *irace* considered would perform best on any function f_id, ends with $\_f\_id$). Figure 5.10 depicts the AUC values, averaged over 5 runs and normalized between 0 and 1 by dividing by 5000.

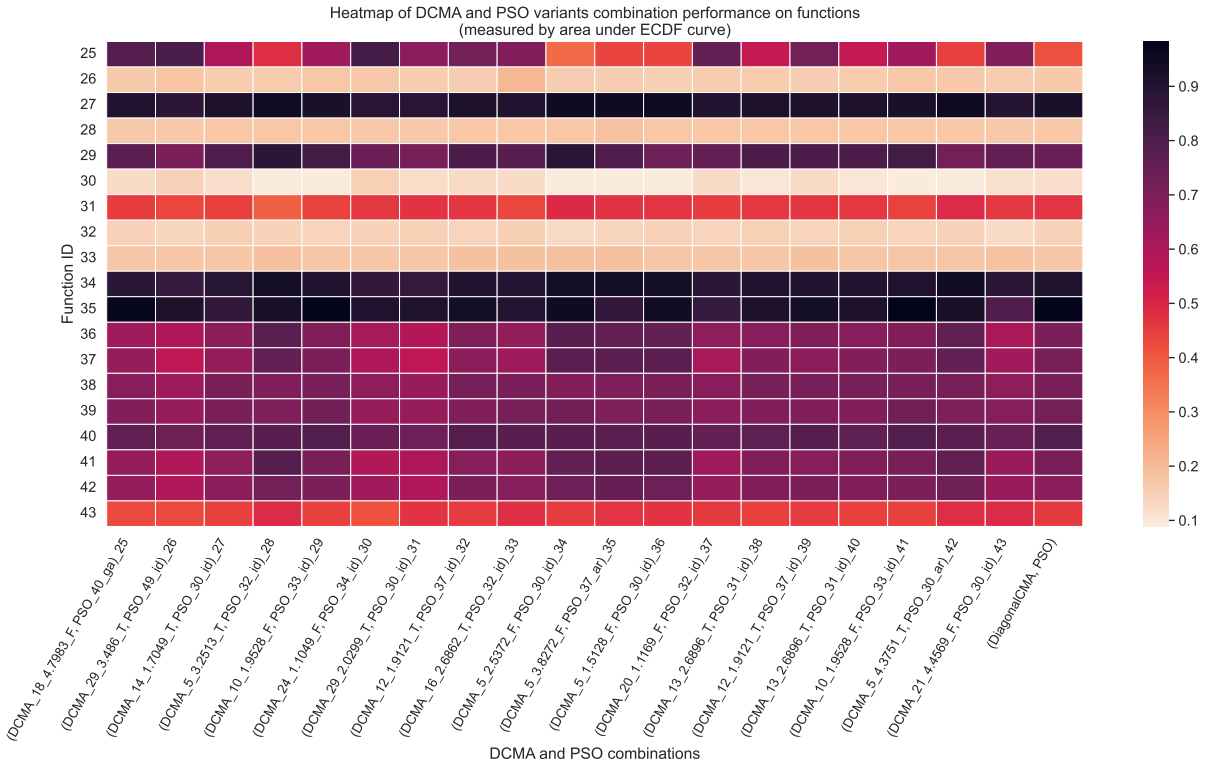


Figure 5.10: Performance of DiagonalCMA and PSO elite variant combinations in terms of normalized AUC values, averaged over 5 runs
*The default variant combination is listed as (DiagonalCMA, PSO)

It is clearly visible from Figure 5.10 that most configured variants indeed outperform the default variant on function 25. To further assess the improvements over all functions, we present a comparison with a normalization scheme for AUC values (averaged over 5 runs) similar to that used for depicting Figure 5.8. Figure 5.11 depicts the performance in this normalization scheme.

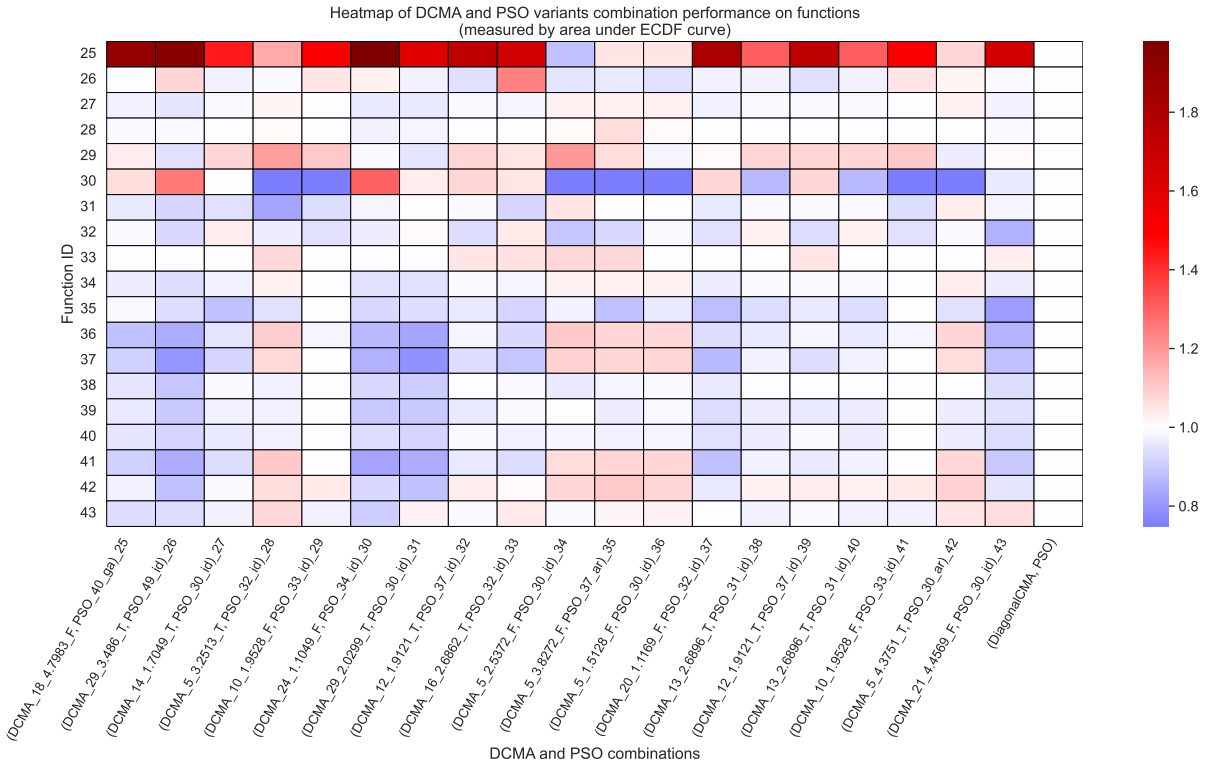


Figure 5.11: Performance of DiagonalCMA and PSO elite variant combinations in terms of normalized AUC w.r.t. default variant's (DiagonalCMA, PSO) AUC values (for each function), averaged over 5 runs

From Figure 5.11, we can interpret that all configured variants except (DCMA_5_2.5372_F, PSO_30_id)_34 outperform the default variant combination on function 25. On function 29, the default variant combination is outperformed by as much as 11 configured variants. On functions 36, 37, 41 and 42; the default variant is outperformed by (DCMA_5_3.2513_T, PSO_32_id)_28, (DCMA_5_2.5372_F, PSO_30_id)_34, (DCMA_5_3.8272_F, PSO_37_ar)_35, (DCMA_5_1.5128_F, PSO_30_id)_36 and (DCMA_5_4.3751_T, PSO_30_ar)_42 configured variant combinations.
In Figure 5.12, we present a bar plot similar to Figure 5.9 for all elite variant combinations, based on aggregated normalized AUC values across all functions (normalization between 0 and 1 for each function, as depicted in Figure 5.10).

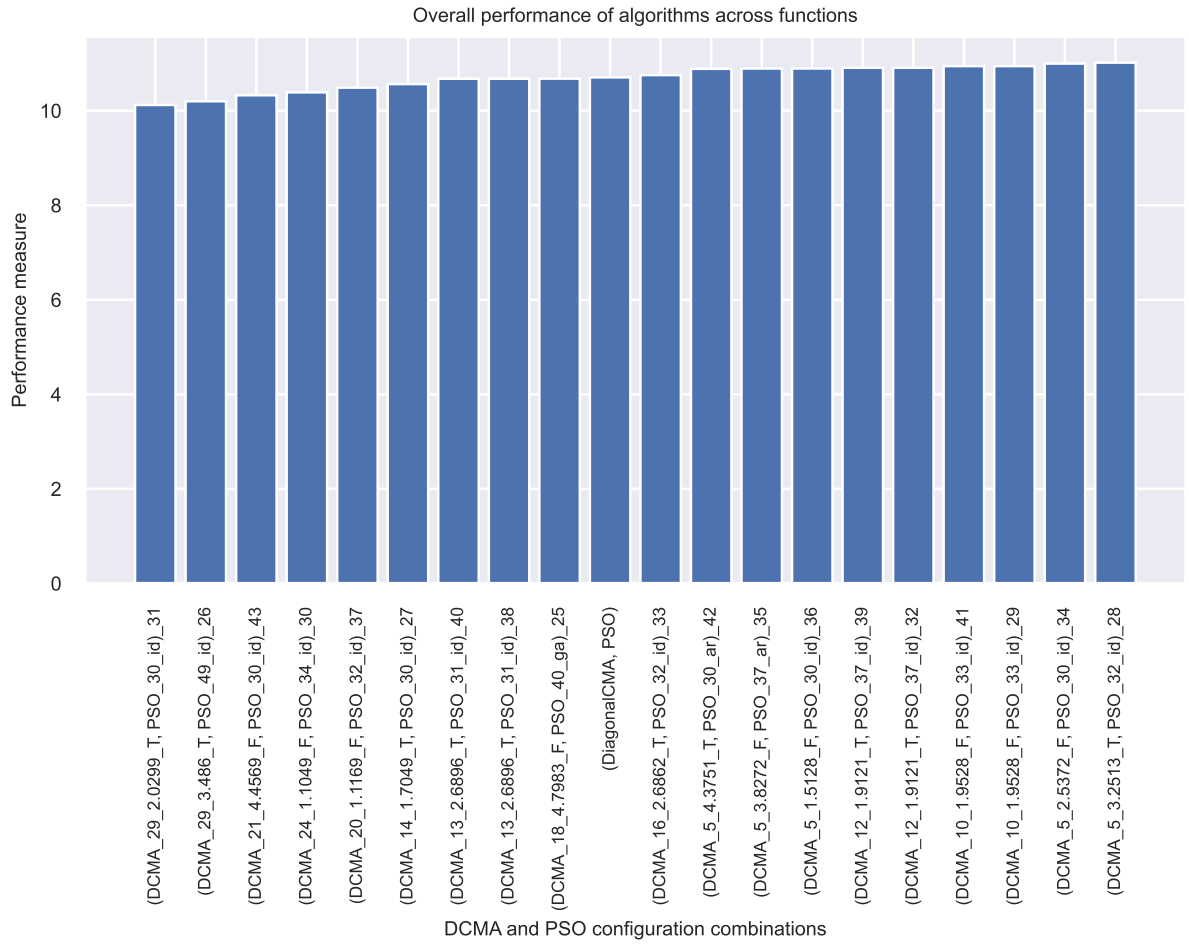


Figure 5.12: Bar plot of aggregated normalized AUC values across functions for configured elite and default variant combinations

Figure 5.12 indicates that 10 elite configured variant combinations managed to outperform the default variant combination overall, by a small margin. It is interesting to note that although the tuning was done for each function individually, some variants out of these 10 elite configured variants were able to outperform the default variant on multiple functions, and were themselves outperformed by the default variant for the remaining few functions. It is also worthy to note that in case of some configured variants, the performance is compromised for multiple functions but enhanced for a few functions.

Other observations of interest from Figures 5.11 and 5.12 are that the configured variants that outperformed the default variant can be divided into two categories similar to that in Section 5.4.1, according to the value of parameter *_popsize*: *_popsize* = 5 and *_popsize* >= 10. The configurations with *_popsize* = 5 ((DCMA_5_3.2513_T, PSO_32_id)_28, (DCMA_5_2.5372_F, PSO_30_id)_34, (DCMA_5_3.8272_F, PSO_37_ar)_35, (DCMA_5_1.5128_F, PSO_30_id)_36 and (DCMA_5_4.3751_T, PSO_30_ar)_42)) outperform the default variant on functions 25, 36, 37, 41 and 42. All these variants except two ((DCMA_5_1.5128_F, PSO_30_id)_36

and (DCMA_5_4.3751_T, PSO_30_ar)_42)) perform better on function 29 as well. On the other hand, configurations with $\_popsize >= 10$:
((DCMA_10_1.9528_F, PSO_33_id)_29, (DCMA_12_1.9121_T, PSO_37_id)_32, (DCMA_18_2.6882_T, PSO_30_id)_33, (DCMA_12_1.9121_T, PSO_37_id)_39 and (DCMA_10_1.9528_F, PSO_33_id)_41) outperform the default variant on functions 25 and 29 but perform worse than the default on functions 36, 37, 41 and in some cases, 42. Hence, it can be inferred that performance on functions 36, 37, 41 and 42 is crucial in determining improvement due to algorithm configuration.

Chapter 6

Conclusions

The primary aim of our work was to construct a portfolio of optimization algorithms via automated algorithm configuration that would yield a good anytime performance. Anytime performance is relevant in practice since in the real-world, we often come across situations where we do not know in advance how much time we will have to solve a given optimization problem, and therefore need algorithms that can find high-quality solutions at any moment. We chose to construct a portfolio with pairs of algorithms run in parallel to take full advantage of performance complementarity.

In the course of our work, we performed a series of experiments to determine the performance improvement, in terms of area under average ECDF curve, of parallel portfolio algorithms over single algorithms. Furthermore, we investigated the parameter space of these portfolio algorithms and explored methods to perform hyperparameter tuning in order to achieve further improved performance. More specifically, we explored two possible methods: tuning hyperparameters of the portfolio algorithm to obtain optimized overall performance (on all objective functions) and tuning to obtain optimized performance per function. We refer to the second method as specialized tuning for objective functions. Our findings reveal that the configurations obtained after specialized tuning achieve a good anytime performance and we also determine certain factors which are crucial to achieve a good anytime performance. These findings can be used to design portfolio algorithms for real-world problems such that anytime performance is maximized.

We observed that the portfolio algorithm that yields the best anytime performance is a combination of DiagonalCMA and PSO, named as (DiagonalCMA, PSO), with both algorithms run independently in parallel, of which only the better solution found at each evaluation was considered for the performance metric calculation. The portfolio algorithm (DiagonalCMA, PSO) demonstrates an approximate 4% increase in performance across functions/anytime performance, in terms of area under average ECDF curve, as compared to the single best solver out of the two i.e., DiagonalCMA. Furthermore, we conclude that

specialized hyperparameter tuning of the portfolio algorithm (DiagonalCMA, PSO) for objective functions results in elite configurations (one elite configuration per function) that also yield a good performance on other objective functions for which the experiments were carried out. The elite configuration: (DCMA_5_3.2513_T, PSO_32_id)_28 obtained from specialized tuning for function-28 i.e., Rosenbrock function, exhibits an approximate improvement of 1.5% in anytime performance as compared to the default variant (DiagonalCMA, PSO).

6.1 Future work

Since our work is centred around a specific set of objective functions, it will be worthwile to carry it out further by applying the approach to other benchmark suites and real-world applications. We could also broaden the approach used in our work for building a portfolio by combining more than two algorithms to further exploit performance complementarity.

The next plausible extension would be to perform selection of optimization algorithms to be run in parallel and algorithm configuration simultaneously, instead of performing the two tasks sequentially. Moreover, another possible future work could be carrying out an in-depth investigation of the specialized hyperparameter tuning method, particularly to determine the pattern in which parameter values are selected for those configurations that yield a better anytime performance than the default variant. As an extension of the above-mentioned task, the functions on which elite configurations yield a better performance than the default variant may also be analysed in depth to determine factors that shape the resulting performance improvement.

Additionally, since the entirety of our work is focused on anytime performance, we could extend our methods towards building of some theoretical foundation for portfolios for anytime performance in the future.

Bibliography

[AH19] Youhei Akimoto and Nikolaus Hansen. "Diagonal Acceleration for Covariance Matrix Adaptation Evolution Strategies". In: *CoRR* abs/1905.05885 (2019). arXiv: 1905.05885.

[BA95] J Martin Bland and Douglas G. Altman. "Multiple significance tests: the Bonferroni method." In: *BMJ* 310 6973 (1995), p. 170.

[Bau14] Petr Baudivs. "COCOpf: An Algorithm Portfolio Framework". In: (2014).

[Ben+21] Pauline Bennet, Carole Doerr, Antoine Moreau, Jeremy Rapin, Fabien Teytaud, and Olivier Teytaud. "Nevergrad: black-box optimization platform". In: *ACM SIGEVOlution* 14 (Apr. 2021), p. 8. DOI: 10.1145/3460310.3460312.

[Dek10] Michel Dekking. In: *A modern introduction to probability and statistics: Understanding why and how*. Springer, 2010, pp. 219–219.

[Doe+18] Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck. "IOHprofiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics". In: *arXiv e-prints:1810.05281* (Oct. 2018). arXiv: 1810.05281.

[GS01] Carla P. Gomes and Bart Selman. "Algorithm portfolios". In: *Artificial Intelligence* 126.1 (2001), pp. 43–62. DOI: https://doi.org/10.1016/S0004-3702(00)00081-3.

[Han+21] Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tusar, and Dimo Brockhoff. "COCO: a platform for comparing continuous optimizers in a black-box setting". In: *Optim. Methods Softw.* 36.1 (2021), pp. 114–144. DOI: 10.1080/10556788.2020.1808977.

[Han16] Nikolaus Hansen. "The CMA Evolution Strategy: A Tutorial". In: *CoRR* abs/1604.00772 (2016). arXiv: 1604.00772.

[Hut+09] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stuetzle. "ParamILS: An Automatic Algorithm Configuration Framework". In: *Journal of Artificial Intelligence Research* 36 (Oct. 2009), pp. 267–306. DOI: 10.1613/jair.2861.

[Kad+10] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. "ISAC –Instance-Specific Algorithm Configuration". In: *Proceedings of the 2010 Conference on ECAI 2010: 19th European Conference on Artificial Intelligence*. NLD: IOS Press, 2010, pp. 751–756. ISBN: 9781607506058.

[KE95] J. Kennedy and R. Eberhart. "Particle swarm optimization". In: 4 (1995), 1942–1948 vol.4. DOI: 10.1109/ICNN.1995.488968.

[Ker+19] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. "Automated Algorithm Selection: Survey and Perspectives". In: *Evolutionary Computation* 27.1 (2019), pp. 3–45. DOI: 10.1162/evco_a_00242.

[Ley+03] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim Mcfadden, and Yoav Shoham. "A Portfolio Approach to Algorithm Selection". In: *IJCAI International Joint Conference on Artificial Intelligence* (May 2003).

[Liu+20] Jialin Liu, Antoine Moreau, Mike Preuss, Baptiste Rozière, Jérémy Rapin, Fabien Teytaud, and Olivier Teytaud. "Versatile Black-Box Optimization". In: *CoRR* abs/2004.14014 (2020). arXiv: 2004.14014.

[Lóp+16] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Thomas Stützle, and Mauro Birattari. "The irace package: Iterated Racing for Automatic Algorithm Configuration". In: *Operations Research Perspectives* 3 (2016), pp. 43–58. DOI: 10.1016/j.orp.2016.09.002.

[Meu+22] Laurent Meunier, Herilalaina Rakotoarison, Pak-Kan Wong, Baptiste Rozière, Jérémy Rapin, Olivier Teytaud, Antoine Moreau, and Carola Doerr. "Black-Box Optimization Revisited: Improving Algorithm Selection Wizards Through Massive Benchmarking". In: *IEEE Trans. Evol. Comput.* 26.3 (2022), pp. 490–500. DOI: 10.1109/TEVC.2021.3108185.

[MG21] Türkay Metin and R. Gani. *31st European Symposium on Computer Aided Process Engineering: Escape-31*. Elsevier, 2021.

[NM65] John A. Nelder and Roger Mead. "A Simplex Method for Function Minimization". In: *Comput. J.* 7 (1965), pp. 308–313.

[Ped+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Pel05] Martin Pelikan. "Probabilistic Model-Building Genetic Algorithms". In: (2005), pp. 13–30. DOI: 10.1007/978-3-540-32373-0_2.

[Pow64] M. J. D. Powell. "An efficient method for finding the minimum of a function of several variables without calculating derivatives". In: *The Computer Journal* 7.2 (Jan. 1964), pp. 155–162. DOI: 10.1093/comjnl/7.2.155. eprint: https://academic.oup.com/comjnl/article-pdf/7/2/155/959784/070155.pdf.

[Qin10] Anyong Qing. "Basics of Differential Evolution". In: *Differential Evolution in Electromagnetics*. Ed. by Anyong Qing and Ching Kwang Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 19–42. DOI: 10.1007/978-3-642-12869-1_2.

[Ric76] John R. Rice. "The Algorithm Selection Problem." In: *Advances in Computers* 15 (1976), pp. 65–118.

[RT18] J. Rapin and O. Teytaud. "Nevergrad - A gradient-free optimization platform". In: *GitHub repository* (2018).

[SP97] Rainer Storn and Kenneth V. Price. "Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces". In: *Journal of Global Optimization* 11 (1997), pp. 341–359.

[TBC16] Dania Tamayo-Vera, Antonio Bolufé-Röhler, and Stephen Chen. "Estimation multivariate normal algorithm with thresheld convergence". In: (2016), pp. 3425–3432. DOI: 10.1109/CEC.2016.7744223.

[Wan+20] Hao Wang, Diederick Vermettern, Furong Ye, Carola Doerr, and Thomas Bäck. "IOHanalyzer: Performance Analysis for Iterative Optimization Heuristic". In: *arXiv e-prints:2007.03953* (2020). arXiv: 2007.03953.

[Wik22a] Wikipedia contributors. “Condition number — Wikipedia, The Free Encyclopedia”. In: (2022). [Online; accessed 31-May-2022].

[Wik22b] Wikipedia contributors. “Empirical distribution function — Wikipedia, The Free Encyclopedia”. In: (2022). [Online; accessed 1-June-2022].

[Wik22c] Wikipedia contributors. “Simplex — Wikipedia, The Free Encyclopedia”. In: (2022). [Online; accessed 2-June-2022].

[Wik22d] Wikipedia contributors. “Trapezoidal rule — Wikipedia, The Free Encyclopedia”. In: (2022). [Online; accessed 1-June-2022].

[Wil45] Frank. Wilcoxon. “Individual Comparisons by Ranking Methods”. In: *Biometrics* 1 (1945), pp. 196–202.

[XHL10] Lin Xu, Holger Hoos, and Kevin Leyton-Brown. “Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 24.1 (July 2010), pp. 210–216.