Towards Large Scale Automated Algorithm Design by Integrating Modular Benchmarking Frameworks

Amine Aziz-Alaoui, Carola Doerr, Johann Dreo

Towards Large Scale Automated Algorithm Design by Integrating Modular Benchmarking Frameworks

Amine Aziz-Alaoui, ISAE-SUPAERO, Université de Toulouse, France*
Carola Doerr, Sorbonne Université, CNRS, LIP6, Paris, France
Johann Dreo, Thales Research & Technology, Palaiseau, France†

May 6, 2021

Abstract

We present a first proof-of-concept use-case that demonstrates the efficiency of interfacing the algorithm framework ParadisEO with the automated algorithm configuration tool irace and the experimental platform IOHprofiler. By combing these three tools, we obtain a powerful benchmarking environment that allows us to systematically analyze large classes of algorithms on complex benchmark problems. Key advantages of our pipeline are fast evaluation times, the possibility to generate rich data sets to support the analysis of the algorithms, and a standardized interface that can be used to benchmark very broad classes of sampling-based optimization heuristics.

In addition to enabling systematic algorithm configuration studies, our approach paves a way for assessing the contribution of new ideas in interplay with already existing operators—a promising avenue for our research domain, which at present may have a too strong focus on comparing entire algorithm instances.

1 Introduction

When confronted with an optimization problem in practice, one of the major challenges that we face is the selection (and the configuration) of an algorithm that corresponds well to the given problem structure, optimization objective(s), and the available resources (compute, possibility to parallelize computations, accessibility of the problem, etc.). A vast amount of different optimization techniques exist, which renders this *algorithm selection problem* non-trivial.

In practice, algorithm selection is often biased by personal preferences and experiences, as well as by practical aspects such as the availability of ready-to-use implementations. Supporting practitioners in making more systematic choices is one of the key objectives of our research domain. A key tool for deriving such recommendations is *algorithm benchmarking*, *i.e.,* the analysis of empirical performance data and search trajectories of one or several algorithms on one or several optimization problems [BDB+20, HAR+20]. Several important benchmarking tools and software frameworks have been developed by our community to ensure sound and meaningful data extraction. These platforms address different stages of the algorithm selection process. They cover, for example, instance selection and generation [SB15, WW18, ZR20], feature extraction [KT16], algorithm configuration [HHLB11, BBFKK10, LDC+16, LJD+16], experimentation [HAR+20, RT18, DWY+18, WKB+14], data analysis [CSC+19, EPK20, FGLP11], and performance extrapolation [KHNT19]. However, most of these tool are developed in isolation,

*This work was partially done during the M2 master internship of Amine Aziz-Alaoui at École Polytechnique, Institut Polytechnique de Paris, CNRS, LIX, Palaiseau, France. He is now a PhD student at Institut de Recherche Technologique Saint Exupéry, Toulouse, France.

†Corresponding author, Johann@Dreo.fr.

paying little attention to building compatible interfaces to other benchmarking modules. This significantly hinders their wider adoption.

With this work we demonstrate the benefits of a fully modular benchmarking pipeline design, which keeps the different steps of the benchmarking study in mind. We see our work as a proof of concept for better compatibility between benchmarking software. On the practical side, our pipeline paves a way for assessing the benefits of new algorithmic ideas in the context of and in interplay with other operators and ideas that our community has to offer.

1.1 Our Contribution

Concretely, we propose in this work a benchmarking pipeline that integrates the modular algorithm framework Paradiseo [KMRS02, CMT04] with the algorithm configuration tool irace [LDC+16], the experimental platform IOHexperimenter [DWY+18], and the data analysis and visualization module IOHanalyzer [WVY+20]. We test our pipeline on tuning a family of genetic algorithms, inspired by [YWDB20], on the so-called W-model problem instances suggested in [WCLW20].

Quality of the results: We show that irace is capable of finding algorithm instances which outperform all baseline algorithms selected by hand, and this for each of the 19 problem instances that we consider. The relative advantage of the best out of 15 irace suggestions over the best baseline algorithm, measured in terms of volume under the discretized Empirical Attainment Function (see Sec. 3.1), varies between 1% and 30%, with a median gain of 13%.

Scalability: Targeting per-instance algorithm design on synthetic benchmark, our algorithmic framework is capable of generating large set of solvers, up to several millions of unique configurations. We show that it is possible to tackle such spaces thanks to fast computations. For instance, we give irace a budget of 100 000 target runs for each of 19 problems, and it completes the full task in approximately 3 hours on a laptop. In our experience, our C++ pipeline is at least 10 times faster than heavily optimized counterparts in Python, not mentioning that most of the available modular frameworks are not always heavily optimized.

Take-away for instance selection: As a side result, we observe that similar algorithm instances can be suggested by irace for some problems, suggesting that the diversity in performance profiles sought in [WCLW20] may be weaker than intended. Our work suggests that an approach like ours may result in a more reliable instance selection, since it will be less biased by a small set of baseline algorithms, but rather be built on a large and diverse set of possible algorithm instances.

Extendability: Our pipeline is ready to perform large benchmark studies, covering large classes of continuous and discrete optimization algorithms. For example, local searches, particle swarm optimization, estimation of distribution algorithms and using numerical or bitstring encodings. Similarly, the pipeline gives direct access to all problems collected in IOHprofiler, which comprises in particular the BBOB functions from the COCO framework [HAR+20], the Nevergrad problem suites [RT18], the W-model instances [WW18], the PBO suite [DYH+20], etc. Additionally, any benchmark or solver which would be plugged into IOHprofiler would be easily used to further extend this study.

1.2 Comparison to Previous Works

Our work is a top-down approach for automatic algorithm design [MLDS14], which uses a parametrized algorithmic framework to instantiate many *algorithm instances*. Following [LIKS17], we observe that this differs from bottom-up "grammar-based" approaches [dSR18a, dSR18b, PS19] or Grammatical Evolution [RCN98, LPC12], which allow for easily designed algorithm space, but complicates algorithm instantiation and optimization. In our case, the width of the design space is already large and we target fast algorithm instantiation. We thus favor the top-down approach. In this first study, we only consider categorical

parameters, for the sake of implementation simplicity.

A similar approach to ours was suggested in [LS12, BLIS20, BLIS16] for multi-objective optimization. Those studies also use irace, but the authors implemented their own modular algorithm frameworks that are restricted to multi-objective optimization. Our work significantly scales up this kind of study, by leveraging larger algorithm design spaces, larger sets of benchmarks, with more problems and allowing a more detailed analysis of the results.

Few other studies consider a bi-objective measurement of performance for automated algorithm design. Most notably, [LS14] introduced the use of the hypervolume for given quality and time budgets. In our case, we use the volume under the curve of the empirical cumulative histogram of quality and time attainments [dFFH01]. This should behave as the sum of hypervolumes defined for a set of thresholds covering the whole domain, instead of a single one.

1.3 Structure of the Paper

Sec. 2 briefly introduces the individual modules of our algorithm design pipeline and how they interplay with each other. The use-case on which we apply this pipeline, as well as the experimental setup are summarized in Sec. 3. The results of our empirical analysis are described in Sec. 4. We conclude our paper in Sec. 5 with a discussion on promising avenues for future work.

1.4 Availability of Code and Data

The code and data used for this study have been archived at [AADD21]. Up-to-date versions of the code are available at `https://github.com/jdreo/paradiseo` and `https://github.com/IOHprofiler/IOHexperimenter`.

2 The Modular Benchmark Pipeline

Figure 1 summarizes our automated algorithm design pipeline for the concrete use-case that will be studied in Sec. 3. The pipeline links an algorithm configurator with an algorithm generator and a benchmark platform. The algorithm configurator asks the algorithm generator to instantiate an algorithm, which then solves a problem of the benchmark platform while being observed by a logger. After this run, the logger's data are summarized as a scalar performance measure, which is sent back to the algorithm configurator. We briefly present in this section the different components of our pipeline, and explain the reasons behind our choices.

Algorithm Framework: Paradiseo Many evolutionary algorithms share similar design patterns, and are often composed of similar operators. This has given rise to several platforms which aim at supporting their users in designing evolutionary heuristics by compiling a set of readily-available operators within a standardized software environment. Given the substantial work that has been put into these frameworks, we decided to build our pipeline around one of the most powerful toolboxes. To this end, we have ranked 39 frameworks among the ones easily available on the web, based on an adhoc metric combining rapidity, activity, features and license, *e.g.,* [NDV15, SL19, GP06, Jen, ECF, FDG+12, CEP08], to name only a few. Since speed is a major concern for our pipeline, we favor frameworks written in C++. To select an up-to-date framework and to ensure availability of support in case of technical issues, we also checked the contribution activity in recent years. These two criteria reduced our choices to Paradiseo [KMRS02, CMT04], OpenBeagle [GP06], and ECF [ECF]. Among these three, Paradiseo covers the largest portfolio of algorithm families, which are composed in the framework by assembling atomic functions (called *operators*). Paradiseo is also the most actively maintained framework among the three, so that we decided to use it for our work.

The upper part of Figure 1 shows the core classes of Paradiseo involved in our setting.

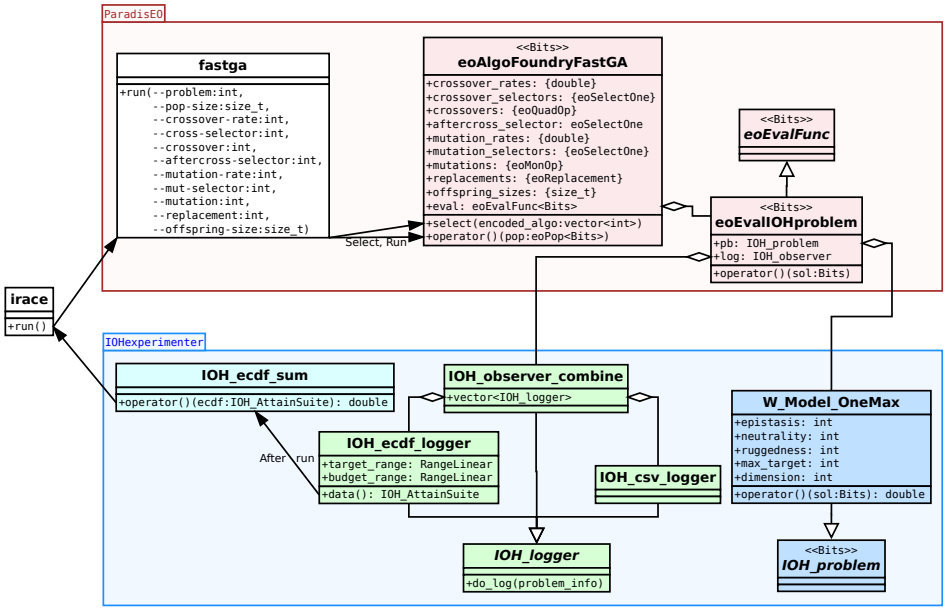

Figure 1: Summary diagram of the FastGA evaluation pipeline involving the Paradiseo (upper part, red colors) and IOHexperimenter (lower part, blue colors) frameworks along with the irace entry point. The execution starts from the irace `run` command on the left, goes through the Paradiseo modules, which call the IOHexperimenter problem (in blue) and loggers (in green). After the run of the algorithm, a statistic is computed on the ECDF data (in cyan), which is then returned to irace as performance metric (*i.e.,* this is the "fitness value" that the evaluation associates to the configuration under evaluation). Involved classes are represented using the UML convention. For the sake of clarity, the IOHprofiler prefix is written as `IOH` and the type of the `eoAlgoFoundryFastGA` slots are indicated as {`double`} instead of `eoOperatorFoundry<double>`.

Algorithm Configuration: irace Several algorithm configuration tools have been developed in the last decade. Among the most common ones used in our community are irace [LDC+16], SMAC [HHLB11], SPOT [BBFKK10], GGA [AMS+15], and hyperband [LJD+16]. We have chosen irace[1] for this study, for practical considerations (previous experience, availability of documentation, support from development team).

Experimental Environment: IOHexperimenter The IOHprofiler project [DWY+18] is a modular platform for algorithm benchmarking of iterative optimization heuristics (IOH). Within this project, IOHexperimenter provides synthetic benchmarks which are very fast to execute and a standardized way of observing algorithms behavior through so-called *loggers*. We have chosen this platform, because it is fast and its modular design made it particularly easy for us to integrate the algorithm design framework (being written in C++, as Paradiseo). IOHprofiler is also actively maintained, and provides access to broad ranges of different optimization processes.

Compared to Nevergrad [RT18], we particularly like the detailed logging options, which provide information about the anytime behavior of the algorithms—information that is currently not available in Nevergrad. Compared to the COCO [HAR+20] environment, IOHprofiler makes it considerably easier to test algorithms' performance on our own benchmark problems or suites. Finally, the project also supports interactive performance analysis and visualization module, IOHanalyzer, which we used for the interpretation of our data.

The lower part of Figure 1 shows the classes related to the loggers and the problems that are used in our experimental study in Sec. 3.

[1] Version 3.4.1 of `https://cran.r-project.org/web/packages/irace/`, ran with *R* 3.6.3.

Data Records: fast ECDF Logger In our use-case, we decide to tune algorithms for good anytime performance, and to use volume under the approximated empirical cumulative density function (ECDF) curve as objective. To this end, we implement within IOHexperimenter an efficient way of computing these values. This "ECDF logger" will be described in Sec. 3.1.

Data Analysis and Visualization with IOHanalyzer Data analysis and visualization is performed via IOHanalyzer [WVY+20], another module of the IOHprofiler project [DWY+18].

3 Use-Case and Experimental Setup

Our use-case is the optimization of the anytime performance of a genetic algorithm on selected instances of the W-model problem. Our performance measure (Sec. 3.1), the algorithmic framework (Sec. 3.2), and the problems (Sec. 3.3) are introduced in the first three subsections. We then summarize the experimental setup of the whole pipeline in Sec. 3.4. Our objective is to find the best algorithm for each instance, which would be the first step of a per-instance, landscape-aware algorithm selection, for instance. We thus do not consider training versus test sets.

3.1 Anytime Performance Measure: AUC

In order to allow for large scale experiments, we implement a fast logger within IOHexperimenter, which essentially stores a histogram of the two-dimensional distribution of the number of runs having reached a quality/time target. The time dimension is given as the number of calls to the objective function, linearly discretized between zero and the allowed budget. The quality dimension is given as the absolute value of the best solution found during the run, linearly discretized between zero and the known $V_{\max}$ bound (see Table 1). This is essentially a discrete version of the Empirical Attainment Function [dFFH01], which is related itself to the multivariate Empirical Cumulative Distribution Function [GF02].

Figure 2 shows two examples of such histograms, arbitrarily chosen. The matrix defines the considered quality/time targets (v, t). The color of each cell corresponds to the probability that the algorithm has identified, within the first t function evaluations, a solution of quality at least v. The darker a cell, the larger the fraction of runs that could successfully meet the quality/time target. Note here that we assume minimization as objective.

Using the histogram of the performance ECDF instead of its continuous counterpart allows to keep the data in-memory, in compact data structures, without having to rely on slow disk accesses.

The performance of the considered algorithm is computed as a statistic on this histogram. In our study, we use the volume under the curve (3D counterpart of the area under the curve, AUC) of the discretized ECDF, approximated as the sum of the EAF histogram. This allows for a compromise between quality and time, which is easily available because we consider synthetic benchmarks with known bounds.

3.2 The $(\mu \dotplus \lambda)$ "Fast" GA Family

We chose for our use-case a family of $(\mu \dotplus \lambda)$ GAs, which is to a large extend inspired by the study [YWDB20]. Algorithm 1 summarizes the framework, called "FastGA" in the implementation.

Essentially, given a parent population of μ points, each of the λ offspring is created by first deciding which variation operator is applied (line 9): with probability p_c the offspring is generated by first recombining two search points from the parent population (lines 11–13) and then randomly deciding (with probability p_m) whether or not to apply a mutation operator to

Algorithm 1: A Configurable Family of $(\mu \overset{+}{,} \lambda)$ Genetic Algorithms.

1 **Input:** Budget B, configuration (μ, λ, p_c, p_m), choice of the operators and conditional parameters. Note that P and P' are multi-sets, *i.e.,* the same point may appear multiple times;
2 **Initialization:**
3 $P \leftarrow \text{InitialSampling}(\mu)$;
4 evaluate the μ points in P;
5 Evals $\leftarrow \mu$;
6 **Optimization: while** Evals $< B$ **do**
7 $P' \leftarrow \emptyset$;
8 **for** $i = 1, \ldots, \lambda$ **do**
9 Sample $r_c \in [0, 1]$ u.a.r.;
10 **if** $r_c \leq p_c$ **then**
11 $\left(y^{(i,1)}, y^{(i,2)}\right) \leftarrow \text{SelectC}(P)$;
12 $\left(y'^{(i,1)}, y'^{(i,2)}\right) \leftarrow \text{Crossover}\left(y^{(i,1)}, y^{(i,2)}\right)$;
13 Sample $z^{(i,1)} \in \left\{y'^{(i,1)}, y'^{(i,2)}\right\}$ u.a.r. ;
14 Sample $r_m \in [0, 1]$ u.a.r. ;
15 **if** $r_m \leq p_m$ **then**
16 $z^{(i,2)} \leftarrow \text{Mutation}\left(z^{(i,1)}\right)$;
17 **else**
18 $z^{(i,2)} \leftarrow z^{(i,1)}$;
19 **else**
20 $z^{(i,1)} \leftarrow \text{SelectM}(P)$;
21 $z^{(i,2)} \leftarrow \text{Mutation}\left(z^{(i,1)}\right)$;
22 Evaluate $z^{(i,2)}$;
23 Evals $\leftarrow$ Evals+1;
24 $P' \leftarrow P' \cup \left\{z^{(i,2)}\right\}$;
25 $P \leftarrow \text{Replace}(P, P', \mu)$;

the so-created offspring (lines 15–18). When crossover was not selected in line 10, the offspring is created by mutation (lines 20–21). When all λ offspring have been created, the iteration is completed by a replacement step (line 25).

Implementation of this Family in Paradiseo: We implement this family of GAs through Paradiseo's "foundries", which allow to register a set of operators (*e.g.,* several kind of mutations) within a "slot" (*e.g.,* the step at which mutation is called within the algorithm). Before each call, it is possible to instantiate a specific operator among the registered ones, for each slot, thus assembling one of the algorithm instance among all the possible combinations of operators. Note that operators can be simple numbers, like a probability. Operators are referenced within slots by their indices.

Most of the operators we use were already available in Paradiseo, to the exception of mutations operators with indices 1–5 (see below), which we implemented for this study. We also implemented the algorithm 1 as the `eoFastGA` class[2], in which to plug the operators.

We consider the following operators and parametrizations, which result in a (large) total number of 1 630 475 different configurations of Algorithm 1. Numbers in brackets indicate the indices of the corresponding operators within its slot.

[2]All our code is contributed to the Paradiseo project.

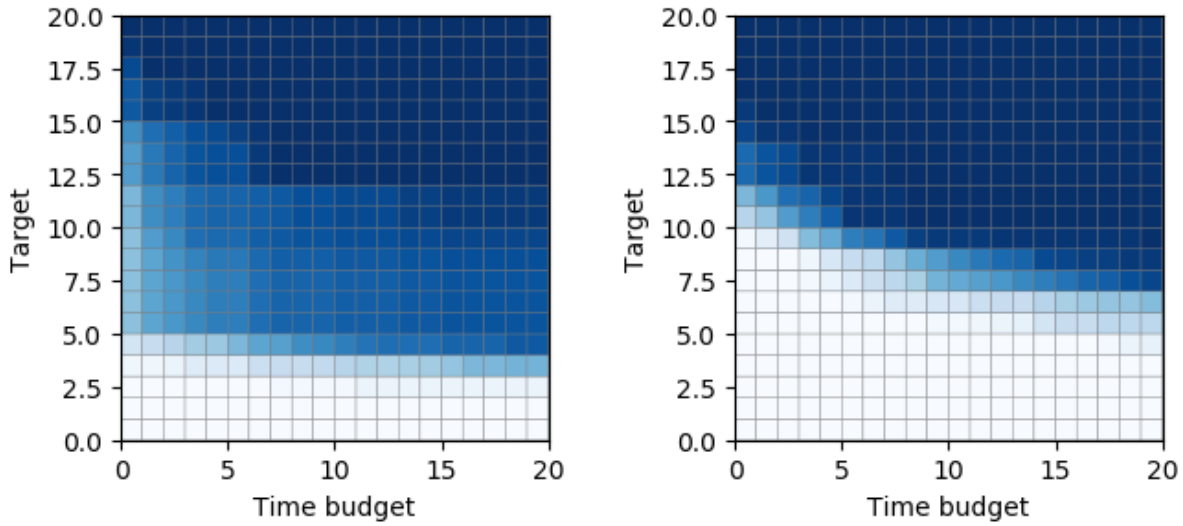

Figure 2: Example of two ECDF histograms. Abscissa shows the time dimension and the ordinate shows the target quality. The colormap shows the probability for the considered algorithm to reach a quality/time target falling in each bucket of the histogram. The figures show ECDF histograms for the 1ptGA algorithm (see Sec. 3.2) after 50 independent runs on problem 2 (left) and of problem 5 (right), respectively, using 20 buckets on each axis.

InitialSampling(μ): Initialization of the Algorithm (1 option) We only consider independent uniform sampling, *i.e.*, the μ points are i.i.d. uniform samples. The corresponding ParadisEO operator is `eoInitFixedLength`.

Crossover rate p_c (6 options). We consider $p_c \in \{0, 0.2, 0.4, 0.5, 0.6, 0.8\}$. Being only able to use the *integer* and *categorical* interface for irace.

SelectC(P): Selection of two points for the crossover operation (7 options). Note that in the implementation, the selection operator (line 11) is called twice to select the two candidate points.

`[0]` `eoRandomSelect()`: Uniformly select a point from P (without removing the first selected individual from the set P used by the second selection). (1 option).

`[1]` `eoStochTournamentSelect(`k`)`: Select a point from P with tournament selection, *i.e.*, we select uniformly at random k different points in P and the best one of these is selected. k denotes the tournament size as percentage of population (*i.e.*, $k \in [0, 1]$). (1 option, $k = 0.5$).

`[2]` `eoSequentialSelect()`: Select the best point from P (with respect to the objective function value). This operator is sometimes referred to as *elitist selection* or *truncation selection*. When called twice, it selects the two distinct best points from P. (1 option).

`[3]` `eoProportionalSelect()`: Select a point from P with so-called fitness-proportional selection, *i.e.*, point $x \in P$ is chosen with probability $f(x)/\sum_{y \in P} f(y)$. (1 option).

`[4--6]` `eoDetTournamentSelect(`k`)`: Like `eoDetTournamentSelect`, but k is deterministic. (3 different options, each one for $k \in [2, 6, 10]$).

Crossover(x, y): Bivariate Variation Operators (11 options).

`[0--4]` `eoUBitXover(`b_c`)`: Uniform crossover with bias (or "preference" in ParadisEO) b_c, setting (independently for each position $i \in [1..n]$) $z_i = x_i$ with probability b_c and setting $z_i = y_i$ otherwise. z denotes the offspring element coming from the crossover of x and y. (5 different options, $b_c \in [0.1, 0.3, 0.5, 0.7, 0.9]$).

`[5--9]` `eoNPtsBitXover(`k`)` : k-point crossover, which selects $i_1, \ldots, i_k$ uniformly at random and without replacement from $[1..n]$ and sets $z_i = x_i$ for $i \in [1..i_1] \cup [i_2 + 1..i_3] \cup \ldots$ and sets $z_i = y_i$ for $i \in [i_1 + 1..i_2] \cup [i_3 + 1..i_4] \cup \ldots$ (5 different options, $k \in [1, 3, 5, 7, 9]$).

`[10]` `eo1PtBitXover()`: Classic 1-point crossover. (1 option). We mistakenly added this option even if it is the same as the previous crossover with $k = 1$.

Mutation probability p_m (6 options): We consider $p_m \in \{0, 0.2, 0.4, 0.5, 0.6, 0.8\}$.

Mutation(x): Univariate Variation Operator (11 options) All mutation operators are unary unbiased in the sense proposed in [LW12]. For a compact representation, we follow the characterization suggested in [DDY20] and define the mutation operators via the distributions that they define over the possible mutation strengths $k \in [0..n]$. After sampling k from the operator-specific distribution, the k-bit flip operator, $\text{flip}_k(\cdot)$, is applied; it flips the entries in k uniformly chosen, pairwise different bits (*i.e.*, the k bits are chosen u.a.r. without replacement).

`[0]` `eoUniformBitMutation()`: The "uniform" mutation operator, which samples k uniformly at random in the set $[0..n]$. (1 option).

`[1]` `eoStandardBitMutation(`$p = 1/n$`)`: This is the standard bit mutation with mutation rate p. It chooses k from the binomial distribution $\mathcal{B}(n, p)$. (1 option).

`[2]` `eoConditionalBitMutation(`$p = 1/n$`)`: A conditional standard bit mutation operator with mutation rate p. It chooses k' from $\mathcal{B}(n-1, p)$ and applies the $\text{flip}_k(\cdot)$ operator with $k = k' + 1$. (1 option).

`[3]` `eoShiftedBitMutation(`$p = 1/n$`)`: The "shifted" standard bit mutation with mutation rate p, suggested in [CD18]. It samples k' from the binomial distribution $\mathcal{B}(n, p)$. When $k' = 0$, it uses $k = 1$ and it uses $k = k'$ otherwise. (1 option).

`[4]` `eoNormalBitMutation(`p`,`σ^2`)`: The "normal" mutation operator suggested in [YDB19]. It samples k from the normal distribution $\mathcal{N}(pn, \sigma^2)$. When $k > n$, k is replaced by a value chosen uniformly at random in the set $[0..n]$. (1 option, $p = 1/n$ and $\sigma^2 = 1.5$).

`[5]` `eoFastBitMutation(`β`)`: The "fast" mutation operator suggested in [DLMN17]. It samples k' from the power-law distribution $\mathcal{P}[L = k] = (C^\beta_{n/2})^{-1} k^{-\beta}$ with $C^\beta_{n/2} = \sum_{i=1}^{n/2} i^{-\beta}$. When k' is larger than n, it samples a uniform value k in $[0..n]$, and it uses $k = k'$ otherwise. (1 option, $\beta = 1.5$).

`[6--10]` `eoDetSingleBitFlip(`k`)`: Deterministically applies $\text{flip}_k(\cdot)$. (5 different options, $k \in [1, 3, 5, 7, 9]$).

SelectM(P): Selection of one point for the mutation operation if crossover was not chosen (7 options) We essentially have the same selection operators as for crossover. The only difference is that we select only one point instead of two.

Replace(P, P', μ): Replacement of population (11 options)

`[0]` `eoPlusReplacement()`: The best μ points of the multiset $P \cup P'$ are chosen. (1 option).

`[1]` `eoCommaReplacement()`: The best μ points of the offspring multiset P' are chosen. (1 option).

`[2]` `eoSSGAWorseReplacement()`: The $min(\lambda, \mu)$ points of the offspring multiset P' replace the worst points in P. (1 option).

`[3--5]` `eoSSGAStochTournamentReplacement(`k`)`: Like `eoSSGADetTournamentReplacement`, k being the the tournament size as percentage of population. (3 different options, $k \in [0.51, 0.71, 0.91]$).

`[6--10]` `eoSSGADetTournamentReplacement(`k`)`: The μ points are selected through tournament selection. Each tournament involves k uniformly chosen points in $P \cup P'$ and the best ones of these k points is selected. This procedure is repeated μ times, each time removing an already selected point from the multi-set $P \cup P'$. (5 different options, $k \in [2, 4, 6, 8, 10]$).

This concludes our description of the high level operators of our family of $(\mu \dot{+} \lambda)$ GAs. The set of all combinations generates the algorithm design space on which we let irace search for the configuration(s) that best solve a given problem instance.

Baseline Algorithms We consider four baseline algorithms, against which we compare the results of the automated design. They were manually chosen without particular justification. **(1)** $(\lambda + \lambda)$ **EA:** no crossover, plus replacement, standard bit mutation, random selector for mutations. **(2)** $(\lambda+\lambda)$ **fEA:** no crossover, plus replacement, fast bit mutation, random selector for mutations. **(3)** $(\lambda+\lambda)$ **xGA:** sequential selections, uniform crossover, standard bit mutation, plus replacement, $p_c = 0.4$, $b_c = 0.4$. **(4)** $(\lambda+\lambda)$ **1ptGA:** sequential selections, 1-point crossover, standard bit mutation, plus replacement, $p_c = 0.4$, $b_c = 0.4$.

3.3 The W-Model Problems

We evaluate our automated algorithm design pipeline on the W-model functions originally suggested in [WW18]. In a nutshell, the W-model is a benchmark problem generator, which allows to tune different characteristics of the problems, see below for a description. We selected from this family of benchmark problems the 19 instances suggested in [WCLW20], which are summarized in Table 1. Note here that the description differs from that given in [WCLW20], since we used the implementation within IOHexperimenter, which was made available in the context of the work [DYH+20]. The problem instances listed in Table 1 are identical to those suggested in [WCLW20], it is only the representations that differ.

It was suggested in [DYH+20] to superpose the W-model transformations to different optimization problems. The instances selected in [WCLW20], however, were only selected from transformations applied to the OneMax problem $\mathrm{OM} : \{0,1\} \to [0..n], x \to \sum_{i=1}^{n} x_i$. The OneMax problem has a very smooth and non-deceptive fitness landscape. Due to the well-known coupon collector effect [FGT92], it is relatively easy to make progress when the function values are small, and the probability to obtain an improving move decreases considerably with increasing function values. The complexity of the OneMax problem can be considerably increased through the following W-model transformations.

(1) Neutrality $W(., \mu_W, ., .)$: The bit string $(x_1, ..., x_n)$ is reduced to a string $(y_1, ..., y_m)$ with $m := n/\mu_W$, where μ_W is a parameter of the transformation (we use the subscript W to indicate that these are parameters of the W-model problem generator). For each $i \in [m]$ the value of y_i is the majority of the bit values in the size-μ substring $(x_{(i-1)\mu_W}, x_{(i-1)\mu_W+1}, ..., x_{i\mu_W})$

Table 1: Test problems on which the pipeline is evaluated, taken from [WCLW20]. In column "best" we list the baseline algorithm with largest average AUC value, reported in column $\mathrm{AUC_b}$. $\mathrm{AUC_i}$ is the average of the 15 mean AUC of elite configurations, as suggested by 15 independent runs of irace with default hyperparameters. AUC-values are w.r.t. to at least 50 validation runs and "rel." indicates the relative gain $(\mathrm{AUC_i} - \mathrm{AUC_b})/\,\mathrm{AUC_b}$.

FID	dim	μ_W	ν_W	γ_W	$V_{\max}$	best	$\mathrm{AUC_b}$	$\mathrm{AUC_i}$	rel.
1	20	2	6	10	10	xGA	8378	8740	4%
2	20	2	6	18	10	fEA	8402	8754	4%
3	16	1	5	72	16	fEA	8352	8397	1%
4	48	3	9	72	16	EA	8299	8914	7%
5	25	1	23	90	25	fEA	8003	8510	6%
6	32	1	2	397	32	1pt	7055	7311	4%
7	128	4	11	0	32	1pt	6833	8183	20%
8	128	4	14	0	32	EA	6885	8499	23%
9	128	4	8	128	32	xGA	8154	8786	8%
10	50	1	36	245	50	fEA	7216	8122	13%
11	100	2	21	256	50	EA	8314	9139	10%
12	150	3	16	613	50	EA	8034	8730	9%
13	128	2	32	256	64	fEA	8076	9345	16%
14	192	3	21	16	64	fEA	6173	7677	24%
15	192	3	21	256	64	fEA	6797	8292	22%
16	192	3	21	403	64	fEA	7273	8592	18%
17	256	4	52	2	64	xGA	6935	9028	30%
18	75	1	60	16	75	EA	5958	7089	19%
19	150	2	32	4	75	EA	7399	8717	18%

of x. That is, $y_i = 1$ if and only if there are at least $\mu_W/2$ ones in this "block". When $n/\mu_W \notin \mathbb{N}$, the last bits of x are copied to y.

(2) Epistasis $W(.,.,\nu_W,.)$: Epistasis introduces local perturbations to the bit strings. It first "cuts" the input string $(x_1, ..., x_n)$ into subsequent blocks of size ν_W. Using a permutation $e_{\nu_W} : \{0,1\}^{\nu_W} \to \{0,1\}^{\nu_W}$, each substring $(x_{(i-1)\nu_W+1}, x_{(i-1)\nu_W+2}, ..., x_{i\nu_W})$ is mapped to another string $(y_{(i-1)\nu_W+1}, y_{(i-1)\nu_W+2}, ..., y_{i\nu_W}) = e_{\nu_W}((x_{(i-1)\nu_W+1}, x_{(i-1)\nu_W+2}, ..., x_{i\nu_W}))$. The permutation e_{ν_W} is chosen in a way that Hamming-1 neighbors are mapped to strings of Hamming distance at least $\nu_W - 1$, see [WW18] for examples.

(3) Ruggedness and Deceptiveness $W(.,.,.,\gamma_W)$: This layer perturbs the fitness values, by applying a permutation $\sigma(\gamma_W)$ to the possible fitness values $[0..n]$. The parameter γ_W can be thought of as a parameter which controls the distance of the permutation to the identity. The permutations $\sigma(\gamma_W)$ are chosen in a way such that the "hardness" of the instances monotonically increases with increasing γ_W, see [WW18] for details.

We convert these functions into a minimization problem by multiplying all values by -1.

3.4 Experimental Setup

Our test bed is the automated design of Algorithm 1 with the options specified in Sec. 3.2 and with the objective to maximize the AUC as defined in Sec. 3.1, and this for each of the 19 problems listed in Table 1. These instances of the W-model problem were suggested in [WCLW20] based on an empirical study using clustering of algorithm performance data, with the goal to select a diverse collection of benchmark problems. Note here that we tune the algorithms for each problem individually. That is, we apply our algorithm design pipeline 19 independent times.

For the sake of simplicity, we fix the population sizes to $\lambda = \mu = 5$, for the search performed by irace and for our baseline algorithms.

For each use-case, we set the budget of the algorithms to $5n$ function evaluations (FEs). To compute the AUC, we evaluate the performance at 100 linearly distributed budgets $b^1, \ldots, b^{100} \in [1, 5n]$ and at 100 linearly distributed target values $v^1, \ldots, v^{100} \in [0, V_{\max}]$. Linearization computes the bucket index $i = \lfloor (x - x_{\min})/(x_{\max} - x_{\min}) \cdot 100 \rfloor$ for both budgets and targets.

To find the best algorithm design, we allow irace a budget of 100 000 target runs. We ensure that irace performed at least 50 independent runs for the first ranked elite configuration, adding additional runs if needed, and keeping all runs if irace conducted more than 50 runs. We do not consider the other elite configurations ranked by irace, even in draw cases. We run irace 15 independent times, to check the robustness of its selection. We compare performance to the four baseline algorithms, which we run 50 independent times each on each of the 19 test problems.

In total, these experiments took around 15×3 hours on a computer with four Intel CPU cores i5-7300HQ at 2.50GHz and Crucial P1 solid-state disks.

4 Experimental Results

Comparison of AUC Values by Function Table 1 compares the AUC values of the best out of the four baseline algorithms against that of the elite configuration suggested by irace. We observe that, for each of the 19 functions, the elite configurations suggested by irace perform better than the best baseline algorithm. We report in Table 1 the average values, but the differences between the individual irace runs are very small, less than 2.1% difference in AUC value between the best and the worst elite configuration for all 19 problems, and less than 1% performance difference for 9 out of the 19 functions. The relative advantage of the irace recommendations over the best baseline algorithms varies between 1% and 30%. When looking at each of the 15 elite configurations suggested per function, the best relative advantage is 31% for F17, whereas two of the irace elites performed worse than the best of the four baseline algorithms on function F3. For all other functions, all 15 irace elites have a better AUC value than the best of the four baseline algorithms. However, although we see a clear advantage of the irace configurations, we should keep in mind that the irace configurations are specifically tuned for each function, whereas the configurations of the four baseline algorithms are identical for all 19 W-model functions.

Unfortunately, our pipeline does not yet allow to tune a single best solver, *i.e.*, a single configuration that maximizes the AUC under the aggregated ECDF curve. Adding this functionality is a straightforward extension of our framework, which we plan to address in future work. The key challenge here is that Paradiseo does not have the feature to easily reset on the fly the states of solvers between two runs on different problems.

Comparison of the Configurations. Table 2 summarizes the best of the 15 elite configurations that were suggested by irace and compares them against the four baseline algorithms. Table 3 shows the distribution of operators chosen by the 15 irace runs. For the latter, we have chosen problem 17 as an example because we observed here the largest relative gain (see Table 1). We have added problem 5 for comparison, because the distribution of operators suggested for it is very distinct from that of problem 17.

It is worth noting that each operator is selected at least once in the 19×15 elite configurations suggested by irace (Table 3, right), which seems to confirm that i) different operators work well on different problems and ii) that irace searches the full design space, giving some indication that it is not too large or too complex for automated tuning approaches.

We can see that, among all the best configurations proposed by irace across 15 runs, none are similar to one of the baseline algorithms. The probability of mutation p_m is most frequently

Table 2: Configuration of the best out of the elite recommendation suggested by 15 independent runs of irace, for each of the 19 benchmark problems specified in Table 1, and compared against the configuration of the four baseline algorithms. The "op." column gives the number of options per operator. All other integer values correspond to the indices with which the different options are listed in Sec. 3.2 and "-" indicates a non-applicable element (*e.g.,* no crossover operator is used when $p_c = 0$).

Operator	op.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	EA	fEA	xGA	1ptGA
p_c	5	1	4	1	2	4	0	3	0	2	4	3	2	3	1	2	2	3	4	4	0	0	2	2
SelectC	7	2	5	3	1	2	-	0	-	2	2	2	2	6	5	5	2	2	2	2	-	-	2	2
Crossover	11	1	2	8	1	2	-	3	-	2	2	10	5	2	9	2	10	2	2	2	-	-	2	5
p_m	5	2	3	2	2	4	-	4	-	4	4	4	4	4	4	4	4	4	4	4	-	-	2	2
SelectM	7	2	4	6	6	3	2	3	2	5	5	2	3	1	2	6	6	5	1	6	0	0	2	2
Mutation	11	8	9	3	9	7	6	10	10	10	9	10	9	10	8	8	10	10	8	9	1	5	1	1
Replace	11	8	9	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 3: Distribution of operators variants recommended by 15 runs of irace, for problems 5 (left), problem 17 (center) and all problems (right). The most selected indices are highlighted in bold and the darker the background color, the more often the operator instance is selected. Empty cells indicates that irace never selected the operator instance, cells with a "-" entry marks indices which are not defined for this operator.

	Problem 5											Problem 17										All problems										
Op. index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Pc	**5**	6	1	3		-	-	-	-	-	-				1	**14**	-	-	-	-	-	-	18	**74**	56	57	65	-	-	-	-	-	-
SelectC	1		**5**	2		4	3	-	-	-	-			**15**					-	-	-	-	18	16	**141**	19	18	30	28	-	-	-	-
Crossover	1	**5**	2	2	1	1				1	2			**15**									11	29	**78**	24	6	8	11	19	18	23	43
Pm			2	3	**10**	-	-	-	-	-	-					**15**	-	-	-	-	-	-	4	9	25	49	**183**	-	-	-	-	-	-
SelectM			**14**				1	-	-	-	-	**6**	2	1	2	2		2	-	-	-	-	26	17	61	36	27	41	**62**	-	-	-	-
Mutation	2	**5**		1	1	2	4												1	**9**	4	1	2	7		3	1	2	6	10	62	81	**96**
Replace	**15**											**15**											**245**	2	3	6	2	1	2	5	1	3	

set to higher values and the most often chosen mutation is deterministic bit flip with larger number of bits (index 10 in the *Mutation* slot, which is the flip_9 mutation operator). This indicates that larger mutation strengths could have been worth investigating, a result that has surprised us, since in most benchmark studies we see small mutation rates as defaults (albeit we do not consider various population sizes in this study). The results confirm the superiority of the *plus* replacement (id. 0 in the *Replace* slot) and support the use of an elitist selection for the crossover candidates (id. 2 in *SelectC*). We can also see that the uniform crossover with $b_c = 0.5$ (id. 2 in *Crossover*) is more often chosen, like a small probability of performing a crossover (id. 1 in p_c).

For some problems, irace almost always suggest a similar algorithm. On problem 17, for example, it often selects a GA with a large probability of applying uniform crossover in combination with deterministic bit flip mutations. For some other problems, a larger variance on the selected operators can be observed. For instance on problem 5, irace selects a high mutation probability along with an elitist mutation selection, but does not show a clear preference for the other slots.

These results support the idea that there is not always a single best solver (*i.e.,* "No Free Lunch"), even when considering limited design and benchmarking spaces. We also see that some problems seem to require certain design choices, whereas others can be solved well by a broad range of configurations. A more detailed analysis of how these preferences correlate with the characteristics of the problems should offer plenty of interesting insights, but is left for future work.

Fixed-budget solution qualities Figure 3 shows two examples of convergence plots, where we plot the values of the best solutions found so far against the number of objective function

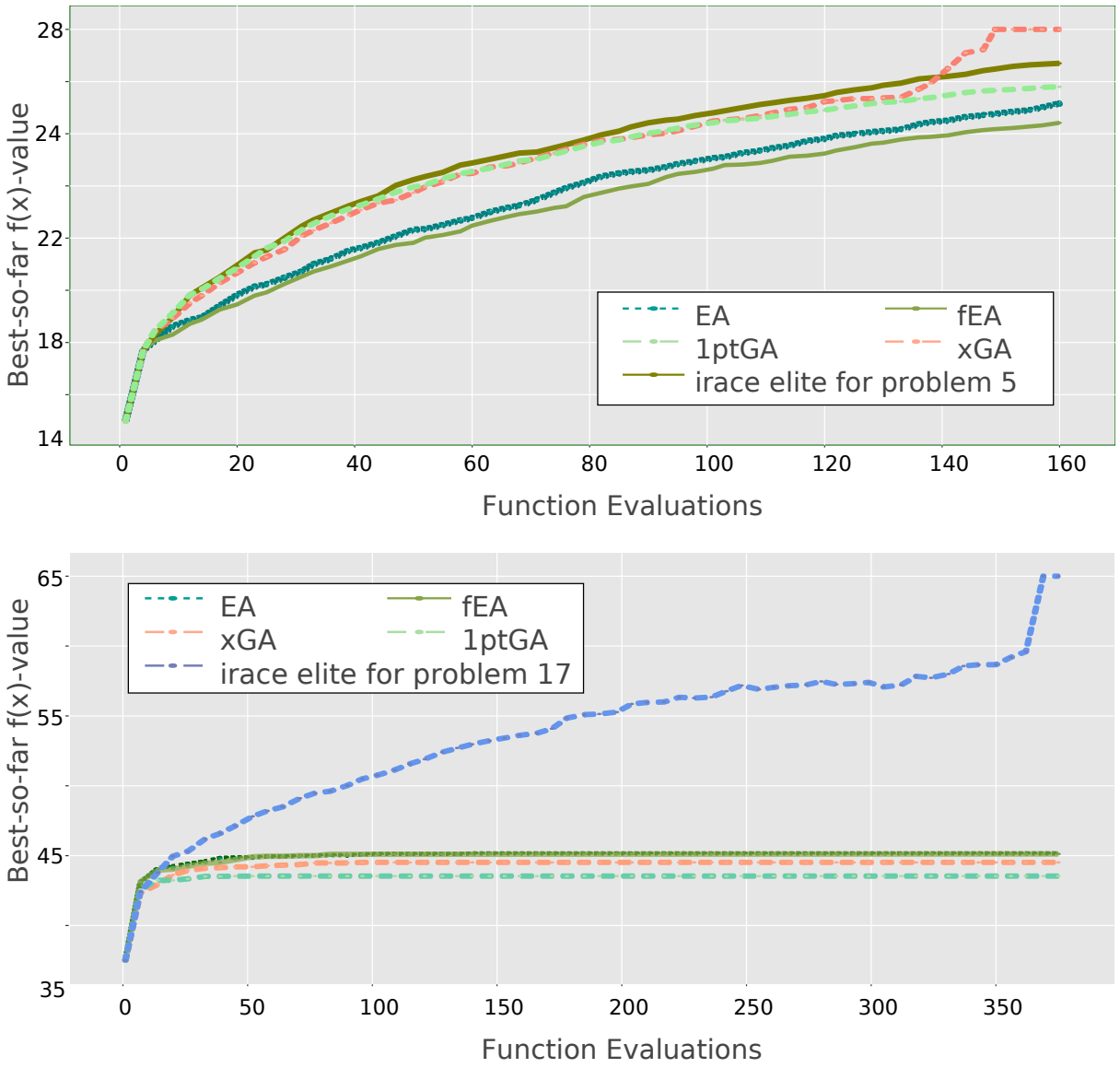

Figure 3: Convergence plots for the baseline algorithms and the elite configurations suggested by irace, on problem 5 (top) and problem 17 (bottom).

evaluations performed, for each baseline algorithm and for the best elite configuration selected by irace. Problems 5 and 17 are chosen to allow for comparison with Table 2.

We observe that the elite configuration on problem 17 is largely more efficient than any of the baseline algorithms. However, on problem 5, the elite configuration is only the most efficient until 140 evaluations. It is selected nonetheless, because we consider the AUC of the 2D ECDF, which takes into account the average performance (across all budgets and targets) rather than the terminal budget of the best target. We believe that, whatever the performance metric we choose, there will always exists such artifacts, where some algorithm would be the best, had we chosen another metric. It is clear, however, that even in this plot the elite configuration performs better *most of the time*.

5 Conclusions and Future Work

By interfacing the three state-of-the-art benchmarking modules from the evolutionary computation literature, irace [LDC+16], Paradiseo [KMRS02], and IOHprofiler [DWY+18], we have

introduced in this work a powerful pipeline for the automated design of sampling-based optimization algorithms. We have demonstrated its efficiency on the use-case of tuning a family of genetic algorithms on instances of the W-model [WW18] suggested in [WCLW20].

Our results supports the idea that automated algorithm design can lead to increased performances and that there are efficient designs which wait to be studied more thoroughly. We believe that efficient pipelines like the one we introduce has the potential to help raising the level of abstraction at which researchers are working. Using automated algorithm design, it becomes possible to check if a newly designed operator can actually be useful in some algorithms/problems coupling [XHHL12]. We also believe that such studies can help deriving generic rules about algorithm design and could probably help theoretical researchers by suggesting *where* to look for interesting structures.

The modular design of the pipeline and its components makes our approach very broadly applicable. It is not restricted to particular types of problems nor to specific algorithms. In particular, extensions to continuous or mixed-integer problems are rather straightforward. Indeed, the Paradiseo framework is designed to separate operators which are independent of the encoding (selection, replacement, etc.) from operator which depends on it (mutation, crossover, etc.), allowing for easy reuse of components and extensions to other algorithmic paradigms (estimation of distribution, local search, multi-objective, etc.). Additionally, the IOHexperimenter provides loggers for vectorial encodings and benchmarks for both numerical and bitstring encodings.

Our work is partially motivated by an industrial application that requires an automated configuration of hardware products. However, we believe that our pipeline is not only interesting for such practical purposes. For researchers, our pipeline offers an elegant way of assessing new algorithm operators and their interplay with already existing ones.

In terms of further development, we plan to add the necessary features which would i) allow for running the same algorithm on multiple problems, while using a single logger that aggregates the results and would ii) support irace's interface for numerical parameters (additionally to categorical and integer ones).

We then plan to test the approach on different algorithms families, with a possible extension to generic "bottom-up" hybridization grammars [MMLIS13] and studies on the most efficient algorithms design (*e.g.,* on the correlations between elite algorithms' operators).

We also plan to extend the framework by integrating feature extraction methods that use algorithm trajectory data [DLV+19, BPRH19] and/or samples specifically made for exploratory landscape analysis [MBT+11, KT16] to couple the algorithm design to such information, similar to the per-instance configuration approaches made in [HHHL06, BDSS17].

Our long-term vision is a pipeline for the automated design of algorithms which adjust their behavior during the optimization process, by taking into account information accumulated so far, similar to the dynamic algorithm configurations studied under the notion of *parameter control* [KHE15]. In contrast to the static designs considered in this work, the automated design of dynamic algorithms requires to select suitable update rules (e.g. based on time, on progress, on self-adaption, etc.).

Finally, we also consider interesting the idea to provide a user-friendly front-end which allows users to assemble a benchmark study by selecting (e.g. through a graphical user interface) one or more algorithms and problems, the budget, etc. and then passing on this study to an automated interface which tunes (if desired) and runs the algorithm(s) and then automatically directs its users to the data summary and visualization platform IOHanalyzer, where the results of the empirical study can be analyzed. We believe that such a pipeline would greatly improve the deployment of evolutionary methods in practice.

Acknowledgments. We thank the GECCO ECADA workshop reviewers for very constructive feedback and for pointers to [LS14].

This work has been financially supported by the Paris Ile-de-France region.

References

[AADD21] Amine Aziz-Alaoui, Carola Doerr, and Johann Dreo, *Towards Large Scale Automated Algorithm Design by Integrating Modular Benchmarking Frameworks — data and analysis scripts*, April 2021.

[AMS+15] Carlos Ansótegui, Yuri Malitsky, Horst Samulowitz, Meinolf Sellmann, and Kevin Tierney, *Model-based genetic algorithms for algorithm configuration*, Proc. of International Conference on Artificial Intelligence (IJCAI'15), AAAI Press, 2015, pp. 733–739.

[BBFKK10] Thomas Bartz-Beielstein, Oliver Flasch, Patrick Koch, and Wolfgang Konen, *SPOT: A toolbox for interactive and automatic tuning in the R environment*, Proc. of the 20. Workshop Computational Intelligence, Universitätsverlag Karlsruhe, 2010, pp. 264–273.

[BDB+20] Thomas Bartz-Beielstein, Carola Doerr, Jakob Bossek, Sowmya Chandrasekaran, Tome Eftimov, Andreas Fischbach, Pascal Kerschke, Manuel López-Ibáñez, Katherine M. Malan, Jason H. Moore, Boris Naujoks, Patryk Orzechowski, Vanessa Volz, Markus Wagner, and Thomas Weise, *Benchmarking in optimization: Best practice and open issues*, CoRR **abs/2007.03488** (2020).

[BDSS17] Nacim Belkhir, Johann Dreo, Pierre Savéant, and Marc Schoenauer, *Per instance algorithm configuration of CMA-ES with limited budget*, Proc. of Genetic and Evolutionary Computation Conference (GECCO'17), ACM, 2017, pp. 681–688.

[BLIS16] Leonardo C. T. Bezerra, Manuel López-Ibáñez, and Thomas Stützle, *Automatic component-wise design of multi-objective evolutionary algorithms*, IEEE Transactions on Evolutionary Computation **20** (2016), no. 3, 403–417.

[BLIS20] ———, *Automatically designing state-of-the-art multi- and many-objective evolutionary algorithms*, Evolutionary Computation **28** (2020), no. 2, 195–226.

[BPRH19] Lukás Bajer, Zbynek Pitra, Jakub Repický, and Martin Holena, *Gaussian process surrogate models for the CMA evolution strategy*, Evolutionary Computation **27** (2019), no. 4, 665–697.

[CD18] Eduardo Carvalho Pinto and Carola Doerr, *Towards a more practice-aware runtime analysis of evolutionary algorithms*, CoRR **abs/1812.00493** (2018).

[CEP08] T. Cloete, Andries Petrus Engelbrecht, and Gary Pampara, *Cilib: A collaborative framework for computational intelligence algorithms - part II*, Proc. of the International Joint Conference on Neural Networks (IJCNN'08), IEEE, 2008, pp. 1764–1773.

[CMT04] Sébastien Cahon, Nordine Melab, and El-Ghazali Talbi, *Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics*, J. Heuristics **10** (2004), no. 3, 357–380, Latest release available on https://nojhan.github.io/paradiseo/.

[CSC+19] Borja Calvo, Ofer M. Shir, Josu Ceberio, Carola Doerr, Hao Wang, Thomas Bäck, and Jose A. Lozano, *Bayesian performance analysis for black-box optimization benchmarking*, Proc. of Genetic and Evolutionary Computation Conference (GECCO'19, Companion), ACM, 2019, pp. 1789–1797.

[DDY20] Benjamin Doerr, Carola Doerr, and Jing Yang, *Optimal parameter choices via precise black-box analysis*, Theoretical Computer Science **801** (2020), 1–34.

[dFFH01] Viviane Grunert da Fonseca, Carlos M. Fonseca, and Andreia O. Hall, *Inferential performance assessment of stochastic optimisers and the attainment function*, Evolutionary Multi-Criterion Optimization, First International Conference, EMO 2001, Zurich, Switzerland, March 7-9, 2001, Proceedings (Eckart Zitzler, Kalyanmoy Deb, Lothar Thiele, Carlos A. Coello Coello, and David Corne, eds.), Lecture Notes in Computer Science, vol. 1993, Springer, 2001, pp. 213–225.

[DLMN17] Benjamin Doerr, Huu Phuoc Le, Régis Makhmara, and Ta Duy Nguyen, *Fast genetic algorithms*, Proc. of Genetic and Evolutionary Computation Conference (GECCO'17), ACM, 2017, pp. 777–784.

[DLV+19] Bilel Derbel, Arnaud Liefooghe, Sébastien Vérel, Hernán E. Aguirre, and Kiyoshi Tanaka, *New features for continuous exploratory landscape analysis based on the SOO tree*, Proc. of Foundations of Genetic Algorithms (FOGA'19), ACM, 2019, pp. 72–86.

[dSR18a] Marcelo de Souza and Marcus Ritt, *Automatic grammar-based design of heuristic algorithms for unconstrained binary quadratic programming*, Evolutionary Computation in Combinatorial Optimization - 18th European Conference, EvoCOP 2018, Parma, Italy, April 4-6, 2018, Proceedings (Arnaud Liefooghe and Manuel López-Ibáñez, eds.), Lecture Notes in Computer Science, vol. 10782, Springer, 2018, pp. 67–84.

[dSR18b] ———, *An automatically designed recombination heuristic for the test-assignment problem*, 2018 IEEE Congress on Evolutionary Computation, CEC 2018, Rio de Janeiro, Brazil, July 8-13, 2018, IEEE, 2018, pp. 1–8.

[DWY+18] Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck, *IOHprofiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics*, CoRR **abs/1810.05281** (2018), Available at http://arxiv.org/abs/1810.05281. A more up-to-date documentation of IOHprofiler is available at https://iohprofiler.github.io/.

[DYH+20] Carola Doerr, Furong Ye, Naama Horesh, Hao Wang, Ofer M. Shir, and Thomas Bäck, *Benchmarking discrete optimization heuristics with iohprofiler*, Applied Soft Computing **88** (2020), 106027.

[ECF] *Evolutionary Computation Framework (ECF), howpublished = http://ecf.zemris.fer.hr/, note = Last visited: 2021-02-04.*

[EPK20] Tome Eftimov, Gasper Petelin, and Peter Korosec, *Dsctool: A web-service-based framework for statistical comparison of stochastic optimization algorithms*, Appl. Soft Comput. **87** (2020), 105977.

[FDG+12] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné, *DEAP: Evolutionary algorithms made easy*, Journal of Machine Learning Research **13** (2012), 2171–2175.

[FGLP11] Carlos M. Fonseca, Andreia P. Guerreiro, Manuel López-Ibáñez, and Luís Paquete, *On the computation of the empirical attainment function*, Proc. of Evolutionary Multi-Criterion Optimization (EMO'11), LNCS, vol. 6576, Springer, 2011, pp. 106–120.

[FGT92] Philippe Flajolet, Danièle Gardy, and Loÿs Thimonier, *Birthday paradox, coupon collectors, caching algorithms and self-organizing search*, Discrete Applied Mathematics **39** (1992), no. 3, 207–229.

[GF02] V. Grunert da Fonseca and C.M. Fonseca, *A link between the multivariate cumulative distribution function and the hitting function for random closed sets*, Statistics & Probability Letters **57** (2002), no. 2, 179–182.

[GP06] Christian Gagné and Marc Parizeau, *Genericity in evolutionary computation software tools: Principles and case study*, International Journal on Artificial Intelligence Tools **15** (2006), no. 2, 173–194.

[HAR+20] Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tušar, and Dimo Brockhoff, *COCO: a platform for comparing continuous optimizers in a black-box setting*, Optimization Methods and Software (2020), 1–31.

[HHHL06] Frank Hutter, Youssef Hamadi, Holger H. Hoos, and Kevin Leyton-Brown, *Performance prediction and automated tuning of randomized and parametric algorithms*, Proc. of Principles and Practice of Constraint Programming (CP'06), LNCS, vol. 4204, Springer, 2006, pp. 213–228.

[HHLB11] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown, *Sequential model-based optimization for general algorithm configuration*, Proc. of Learning and Intelligent Optimization (LION'11), Springer, 2011, pp. 507–523.

[Jen] *Jenetics, howpublished = https://jenetics.io/, note = Last visited: 2021-02-04.*

[KHE15] Giorgos Karafotias, Mark Hoogendoorn, and A.E. Eiben, *Parameter control in evolutionary algorithms: Trends and challenges*, IEEE Transactions on Evolutionary Computation **19** (2015), 167–187.

[KHNT19] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann, *Automated algorithm selection: Survey and perspectives*, Evolutionary Computation **27** (2019), no. 1, 3–45.

[KMRS02] Maarten Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer, *Evolving objects: A general purpose evolutionary computation library*, Artificial Evolution **2310** (2002), 829–888, Latest release available on https://nojhan.github.io/paradiseo/.

[KT16] Pascal Kerschke and Heike Trautmann, *The r-package FLACCO for exploratory landscape analysis with applications to multi-objective optimization problems*, Proc. of IEEE Congress on Evolutionary Computation (CEC'16), IEEE, 2016, pp. 5262–5269.

[LDC+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** (2016), 43–58.

[LIKS17] Manuel López-Ibánez, Marie-Eléonore Kessaci, and Thomas G Stützle, *Automatic design of hybrid metaheuristic from algorithmic components*, Tech. report, 2017.

[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).

[LPC12] Nuno Lourenço, Francisco Pereira, and Ernesto Costa, *Evolving Evolutionary Algorithms*, Proc. of Genetic and Evolutionary Computation Conference (GECCO'12, Companion Material), ACM, 2012, pp. 51–58.

[LS12] Manuel López-Ibáñez and Thomas Stützle, *The automatic design of multiobjective ant colony optimization algorithms*, IEEE Trans. Evol. Comput. **16** (2012), no. 6, 861–875.

[LS14] ———, *Automatically improving the anytime behaviour of optimisation algorithms*, Eur. J. Oper. Res. **235** (2014), no. 3, 569–582.

[LW12] Per Kristian Lehre and Carsten Witt, *Black-box search by unbiased variation*, Algorithmica **64** (2012), 623–642.

[MBT+11] Olaf Mersmann, Bernd Bischl, Heike Trautmann, Mike Preuss, Claus Weihs, and Günter Rudolph, *Exploratory landscape analysis*, Proc. of Genetic and Evolutionary Computation Conference (GECCO'11), ACM, 2011, pp. 829–836.

[MLDS14] Franco Mascia, Manuel López-Ibáñez, Jérémie Dubois-Lacoste, and Thomas Stützle, *Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools*, Comput. Oper. Res. **51** (2014), 190–199.

[MMLIS13] Marie-Eléonore Marmion, Franco Mascia, Manuel López-Ibánez, and Thomas Stützle, *Towards the Automatic Design of Metaheuristics*, MIC 2013 - 10th Metaheuristics International Conference (Singapore, Singapore) (Hoong Chuin Lau, Günther Raidl, , and Pascal Van Hentenryck, eds.), Proceedings of the 10th Metaheuristics International Conference (MIC2013), August 2013, pp. 1–3.

[NDV15] Antonio J. Nebro, Juan J. Durillo, and Matthieu Vergne, *Redesigning the jmetal multi-objective optimization framework*, Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation (New York, NY, USA), GECCO Companion '15, Association for Computing Machinery, 2015, p. 1093–1100.

[PS19] Federico Pagnozzi and Thomas Stützle, *Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems*, Eur. J. Oper. Res. **276** (2019), no. 2, 409–421.

[RCN98] Conor Ryan, John James Collins, and Michael O Neill, *Grammatical evolution: Evolving programs for an arbitrary language*, European Conference on Genetic Programming, Springer, 1998, pp. 83–96.

[RT18] Jérémy Rapin and Olivier Teytaud, *Nevergrad - A gradient-free optimization platform*, https://GitHub.com/FacebookResearch/Nevergrad, 2018.

[SB15] Kate Smith-Miles and Simon Bowly, *Generating new test instances by evolving in instance space*, Comput. Oper. Res. **63** (2015), 102–113.

[SL19] Eric O. Scott and Sean Luke, *ECJ at 20: toward a general metaheuristics toolkit*, Proc. of Genetic and Evolutionary Computation Conference (GECCO'19, Companion Material), ACM, 2019, pp. 1391–1398.

[WCLW20] Thomas Weise, Yan Chen, Xinlu Li, and Zhize Wu, *Selecting a diverse set of benchmark instances from a tunable model problem for black-box discrete optimization algorithms*, Appl. Soft Comput. **92** (2020), 106269.

[WKB+14] Stefan Wagner, Gabriel Kronberger, Andreas Beham, Michael Kommenda, Andreas Scheibenpflug, Erik Pitzer, Stefan Vonolfen, Monika Kofler, Stephan Winkler, Viktoria Dorfer, and Michael Affenzeller, *Architecture and design of the heuristiclab optimization environment*, Topics in Intelligent Engineering and Informatics, vol. 6, Springer, 2014, pp. 197–261.

[WVY+20] Hao Wang, Diederick Vermetten, Furong Ye, Carola Doerr, and Thomas Bäck, *Iohanalyzer: Performance analysis for iterative optimization heuristic*, CoRR **abs/2007.03953** (2020).

[WW18] Thomas Weise and Zijun Wu, *Difficult features of combinatorial optimization problems and the tunable w-model benchmark problem for simulating them*, Proc. of Genetic and Evolutionary Computation Conference (GECCO'18, Companion Material), ACM, 2018, pp. 1769–1776.

[XHHL12] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown, *Evaluating component solver contributions to portfolio-based algorithm selectors*, Proc. of Theory and Applications of Satisfiability Testing (SAT'12), Lecture Notes in Computer Science, vol. 7317, Springer, 2012, pp. 228–241.

[YDB19] Furong Ye, Carola Doerr, and Thomas Bäck, *Interpolating local and global search by controlling the variance of standard bit mutation*, Proc. of IEEE Congress on Evolutionary Computation (CEC'19), IEEE, 2019, pp. 2292–2299.

[YWDB20] Furong Ye, Hao Wang, Carola Doerr, and Thomas Bäck, *Benchmarking a* $(\mu + \lambda)$ *genetic algorithm with configurable crossover probability*, Proc. of Parallel Problem Solving from Nature (PPSN'20), LNCS, vol. 12270, Springer, 2020, pp. 699–713.

[ZR20] Martin Zaefferer and Frederik Rehbach, *Continuous optimization benchmarks by simulation*, Proc. of Parallel Problem Solving from Nature (PPSN'20), Lecture Notes in Computer Science, vol. 12269, Springer, 2020, pp. 273–286.

A Comparison of frameworks

Table 4: Comparison of software frameworks for evolutionary computation. Rank is based on an adhoc aggregation of subjective metrics based on performance of the programming language, activity of the project (number of contributors), breadth of features (number of modules, number of lines of code), and ease of integration in industrial projects (license). Data has been gathered in 2019.

N	Perf	Activit	Featu	Licenc	Score	Name	Language	Type	Upda	License	Contrib.	kloc
1	98,9	79,2	90,0	100,0	92,0	**ParadisEO**	C++	Framework	2019	LGPLv2	33	82
2	97,1	72,5	40,0	100,0	77,4	**jMetal**	Java	Framework	2019	MIT	29	60
3	98,9	54,2	40,0	100,0	73,3	**ECF**	C++	Framework	2017	MIT	19	15
4	98,9	29,2	40,0	100,0	67,0	**OpenBeagle**	C++	Framework	2017	LGPLv3	4	48
5	97,1	40,8	20,0	100,0	64,5	**Jenetics**	Java	Framework	2019	Apachev2	10	47
6	97,1	78,3	80,0	0,0	63,8	**ECJ**	Java	Framework	2018	AFLv3	33	54
7	3,7	99,2	50,0	100,0	63,2	**DEAP**	Python	Framework	2019	LGPLv3	45	9
8	97,1	25,8	10,0	100,0	58,2	**GP.NET**	C#	Library	2019	MIT	1	
9	97,1	24,2	10,0	100,0	57,8	**DGPF**	Java	Framework	2007	LGPLv2	6	
10	97,1	22,5	10,0	100,0	57,4	**JGAP**	Java	Library	2015	LGPLv2	1	
11	97,1	22,5	10,0	100,0	57,4	**Watchmaker**	Java	Framework	2013	Apachev2	2	
12	97,1	17,5	10,0	100,0	56,1	**GenPro**	Java	Framework	2009	Apachev2	1	
13	98,9	8,3	10,0	100,0	54,3	**GAlib**	C++	Library	1998	MIT	1	
14	3,7	77,5	20,0	100,0	50,3	**PyBrain**	Python	Module	2017	MIT	33	
15	97,1	21,7	20,0	50,0	47,2	**JCLEC**	Java	Framework	2014	?	1	
16	97,1	57,5	30,0	0,0	46,2	**HeuristicLab**	C#	Library	2019	GPLv3	20	
17	97,1	14,2	10,0	50,0	42,8	**GPE**	C#	Framework	2005	?	1	
18	97,1	13,3	10,0	50,0	42,6	**JGAlib**	Java	Library	2004	?	1	
19	0,0	52,5	10,0	100,0	40,6	**CIlib**	Scala	Framework	2019	Apachev2	17	
20	3,7	30,8	20,0	100,0	38,6	**pycma**	Python	Solver	2019	BSD	4	
21	3,7	40,8	10,0	100,0	38,6	**PyEvolve**	Python	Framework	2015	PSF	12	
22	0,0	36,7	10,0	100,0	36,7	**GPLAB**	Matlab	Library	2018	LGPLv2	8	
23	80,9	52,5	10,0	0,0	35,9	**Clojush**	Clojure	Framework	2019	EPLv1	17	
24	3,7	20,8	10,0	100,0	33,6	**pySTEP**	Python	Framework	2013	MIT	1	
25	98,9	25,0	10,0	0,0	33,5	**µGP3**	C++	Framework	2016	GPLv2	2	
26	3,7	20,0	10,0	100,0	33,4	**Pyvolution**	Python	Framework	2012	Apachev2	1	
27	98,9	21,7	10,0	0,0	32,6	**PISA**	C++	Library	2008	*	4	
28	97,1	22,5	10,0	0,0	32,4	**EvoJ**	Java	Framework	2015	CC-BY-NC-SA-3	1	
29	97,1	20,8	10,0	0,0	32,0	**Galapagos**	Java	Framework	2013	GPLv2	1	
30	97,1	20,0	10,0	0,0	31,8	**branecloud**	C#	Framework	2012	AFLv3	1	
31	97,1	16,7	10,0	0,0	30,9	**JAGA**	Java	Framework	2008	GPLv2	1	
32	98,9	11,7	10,0	0,0	30,1	**PMDGP**	C++	Framework	2002	GPLv2	1	
33	98,9	9,2	10,0	0,0	29,5	**GPC++**	C++	Framework	1997	GPLv2	2	
34	3,7	25,0	10,0	50,0	22,2	**PonyGE**	Python	Framework	2014	?	3	
35	3,7	39,2	30,0	0,0	18,2	**Platypus**	Python	Framework	2019	GPLv3	9	
36	0,0	10,8	10,0	50,0	17,7	**DCTG-GP**	Prolog	Library	2001	?	1	
37	3,7	34,2	20,0	0,0	14,5	**Desdeo**	Python	Framework	2019	MPLv2	6	
38	3,7	38,3	10,0	0,0	13,0	**PonyGE2**	Python	Framework	2018	GPLv3	9	
39	3,7	25,8	0,0	0,0	7,4	**EvoGrad**	Python	Framework	2019	Proprietary	1	

B Average AUC values

Table 5: Average AUC values of the elites returned by irace in each of the 15 independent runs and of the four baseline algorithms on each of the 19 W-model instances

	AUC of the elite returned by irace run number															Baseline Algorithms			
FID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1ptGA	EA	fEA	xGA
1	8,696	8,700	8,716	8,728	8,731	8,732	8,735	8,742	8,748	8,749	8,753	8,753	8,755	8,759	8,804	8351.2	8340.6	8369	8377.8
2	8,704	8,734	8,738	8,739	8,740	8,742	8,754	8,754	8,760	8,761	8,765	8,769	8,770	8,779	8,794	8265.6	8318.2	8402.4	8311.4
3	8,298	8,350	8,354	8,359	8,368	8,386	8,395	8,401	8,411	8,411	8,426	8,437	8,441	8,443	8,472	8300.48	8341.5	8352.14	8328.02
4	8,862	8,882	8,886	8,886	8,888	8,892	8,909	8,912	8,929	8,930	8,930	8,940	8,943	8,949	8,970	8179.12	8298.52	8290.12	8252.38
5	8,478	8,478	8,480	8,488	8,491	8,498	8,498	8,501	8,502	8,507	8,533	8,546	8,547	8,548	8,554	7923.6	7975.6	8002.88	7983.12
6	7,241	7,261	7,276	7,276	7,281	7,288	7,304	7,315	7,316	7,325	7,330	7,342	7,367	7,373	7,374	7055.06	6666.08	6534.9	7054.04
7	8,120	8,155	8,159	8,161	8,173	8,175	8,181	8,186	8,188	8,188	8,195	8,195	8,201	8,214	8,252	6833.16	6828.8	6827.14	6698.18
8	8,451	8,468	8,474	8,480	8,482	8,486	8,491	8,503	8,506	8,506	8,509	8,510	8,520	8,524	8,574	6796.86	6884.5	6851.04	6750.42
9	8,758	8,761	8,771	8,774	8,779	8,783	8,787	8,790	8,793	8,794	8,796	8,796	8,796	8,807	8,807	8075.14	8089.02	8040.36	8153.66
10	8,062	8,079	8,090	8,096	8,102	8,108	8,108	8,110	8,124	8,137	8,146	8,155	8,162	8,167	8,177	7091.52	7187.64	7216.32	7139.88
11	9,117	9,120	9,123	9,125	9,127	9,132	9,139	9,140	9,143	9,144	9,149	9,156	9,158	9,158	9,159	8178.84	8314.34	8289.5	8183.16
12	8,690	8,696	8,717	8,718	8,721	8,728	8,733	8,734	8,735	8,739	8,744	8,746	8,747	8,749	8,759	7960.4	8033.76	8022.48	8000.08
13	9,304	9,324	9,333	9,337	9,338	9,347	9,348	9,351	9,352	9,353	9,353	9,356	9,359	9,360	9,368	8001.24	8068.56	8076.36	7969.56
14	7,610	7,628	7,634	7,636	7,640	7,665	7,670	7,688	7,691	7,693	7,695	7,702	7,705	7,724	7,729	6064.98	6137.96	6173.42	6138.36
15	8,255	8,265	8,272	8,279	8,279	8,284	8,288	8,295	8,297	8,298	8,302	8,306	8,315	8,318	8,330	6703.18	6768.46	6797.36	6765.32
16	8,552	8,565	8,573	8,580	8,588	8,591	8,594	8,600	8,601	8,601	8,602	8,603	8,609	8,609	8,619	7164.98	7237.94	7273.42	7238.36
17	8,995	8,997	9,012	9,013	9,015	9,024	9,024	9,026	9,028	9,030	9,031	9,037	9,057	9,064	9,067	6838.68	6910	6873.06	6935.24
18	7,042	7,047	7,057	7,069	7,070	7,078	7,091	7,094	7,103	7,103	7,104	7,105	7,108	7,133	7,134	5764.78	5957.88	5956.34	5885.52
19	8,685	8,698	8,700	8,701	8,702	8,706	8,709	8,718	8,720	8,722	8,727	8,733	8,734	8,751	8,756	7330.28	7398.6	7359.56	7287.74

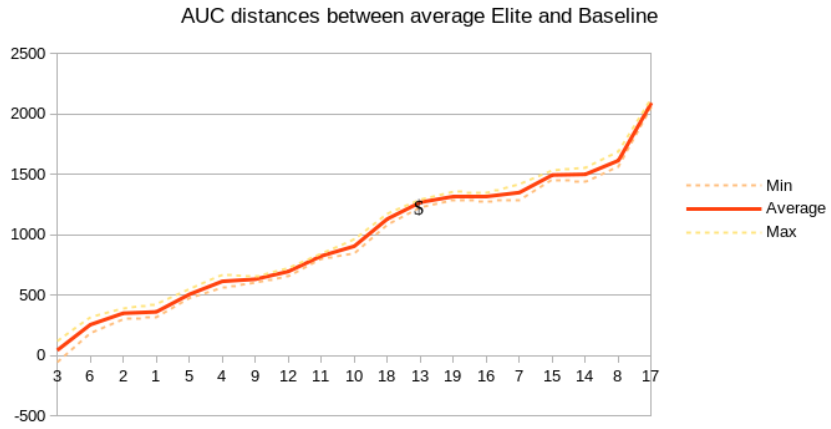

Figure 4: Distances between AUCs of elite algorithms and baseline algorithms.

C Diagram of Paradiseo classes

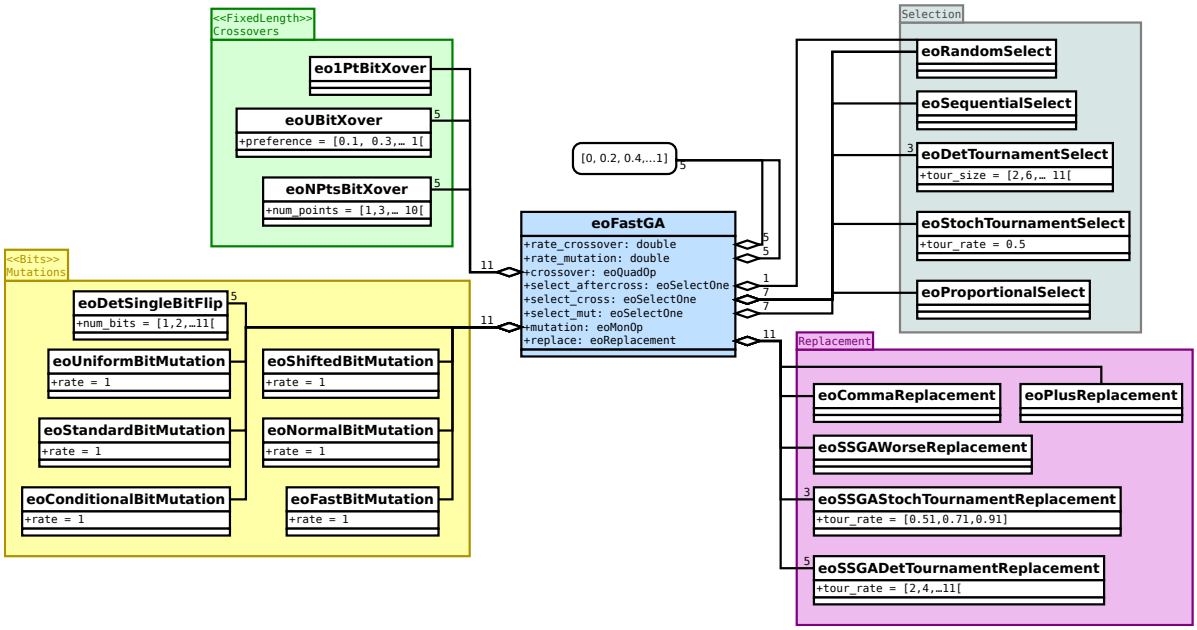

Figure 5: Summary UML diagram of the FastGA family of algorithms, as modeled with Paradiseo classes. Aggregation arrows shows the cardinality of instances (arrow tail side) and slots (arrow head side) involved in the final combination. No cardinality is indicated when it equals one.

D Convergence plots for all problems

The following 19 figures shows the convergence plots of the baseline algorithms against the best elite selected by irace. Algorithms are denoted in the legend by the set of indices for each slots, using the following code (see Table 2 for the corresponding algorithms):

- P: population size (always 5 in this study),
- C: crossover probability,
- s: crossover selector,
- c: crossover,
- a: selector after crossover (always 0 in this study),
- M: mutation probability,
- u: mutation selector,
- m: mutation,
- r: replacement,
- O: stopping criterion (always 0 in this study).

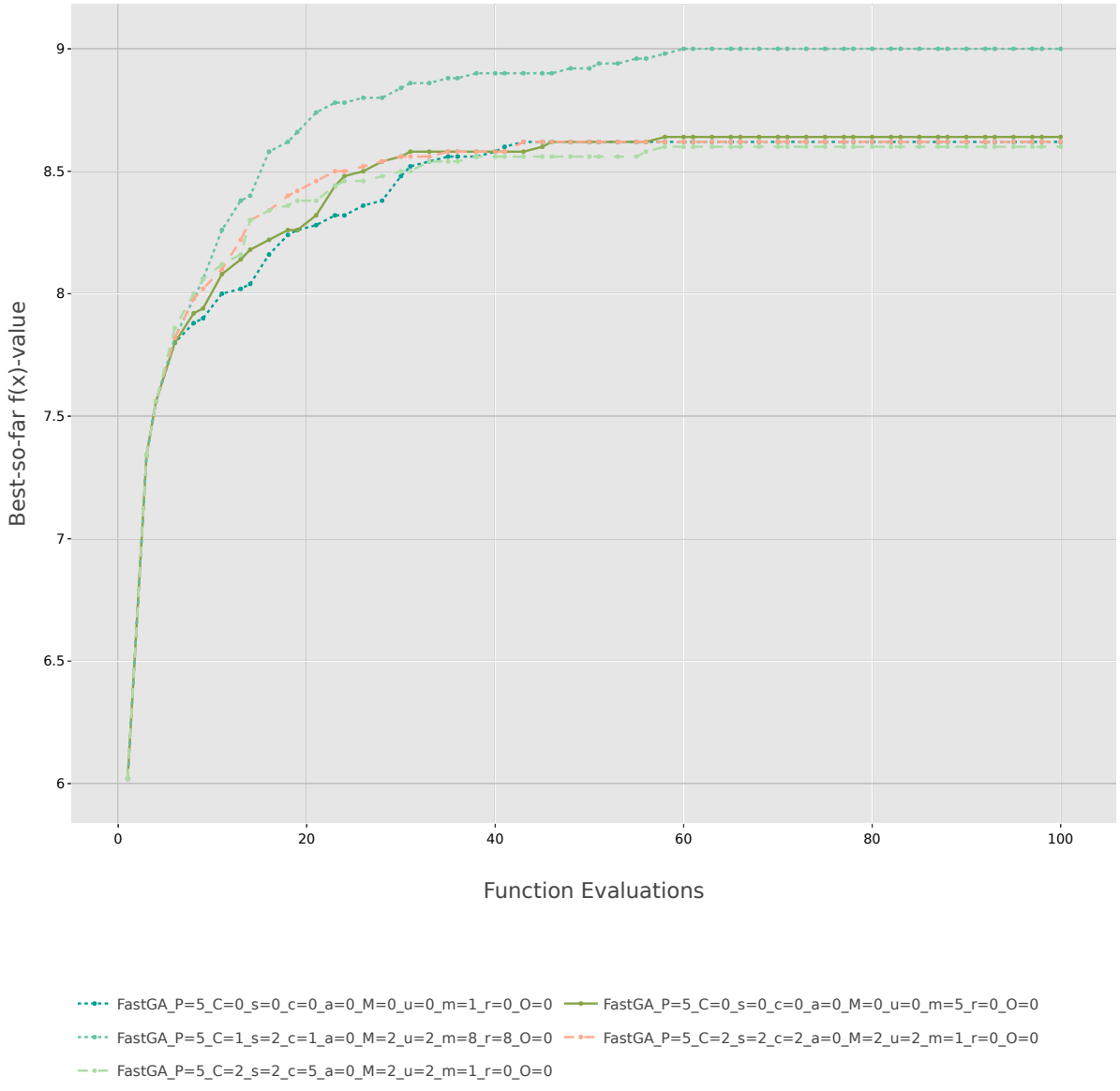

Figure 6: Convergence plot of baseline algorithms and elite, for problem 1.

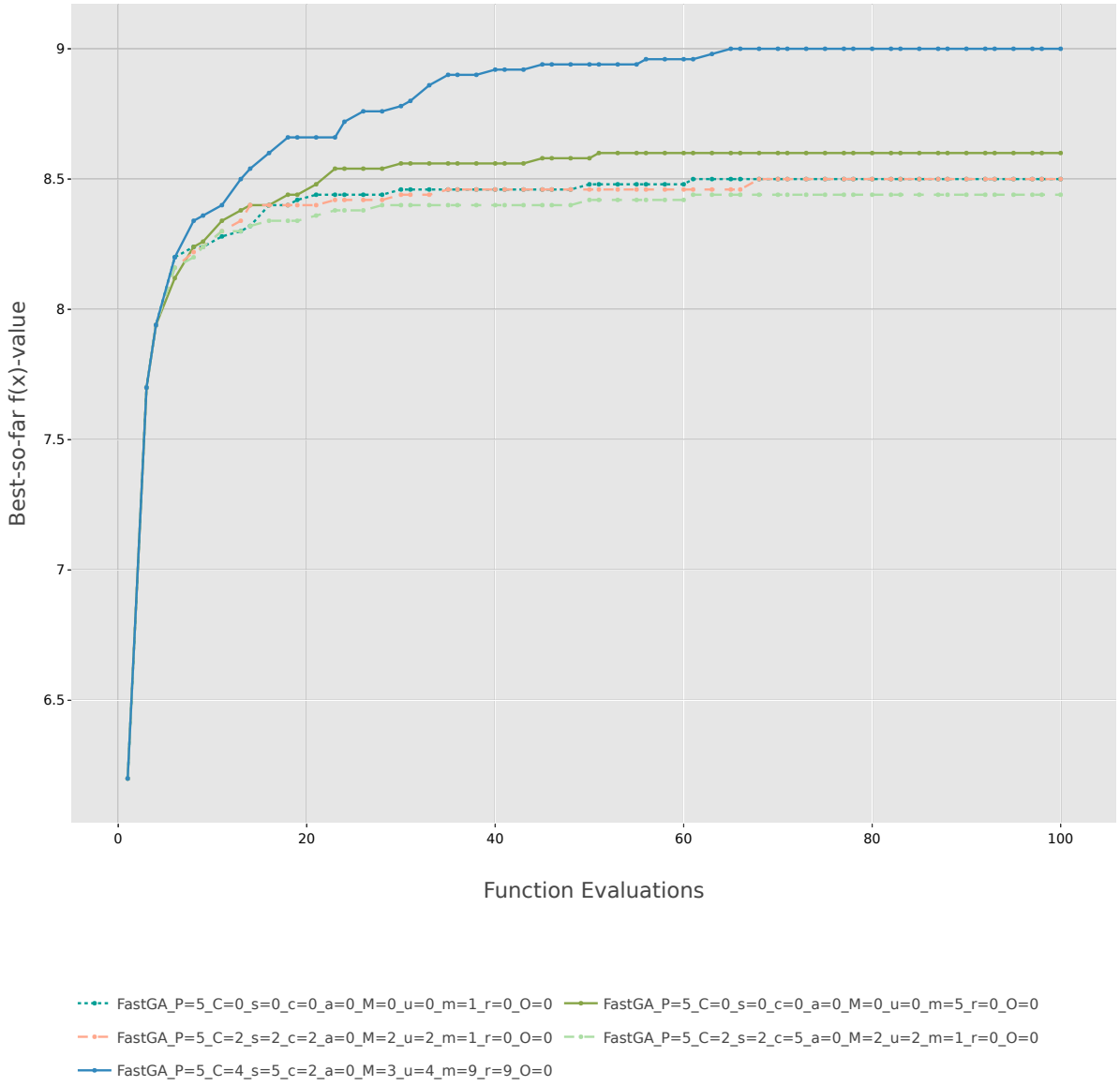

Figure 7: Convergence plot of baseline algorithms and elite, for problem 2.

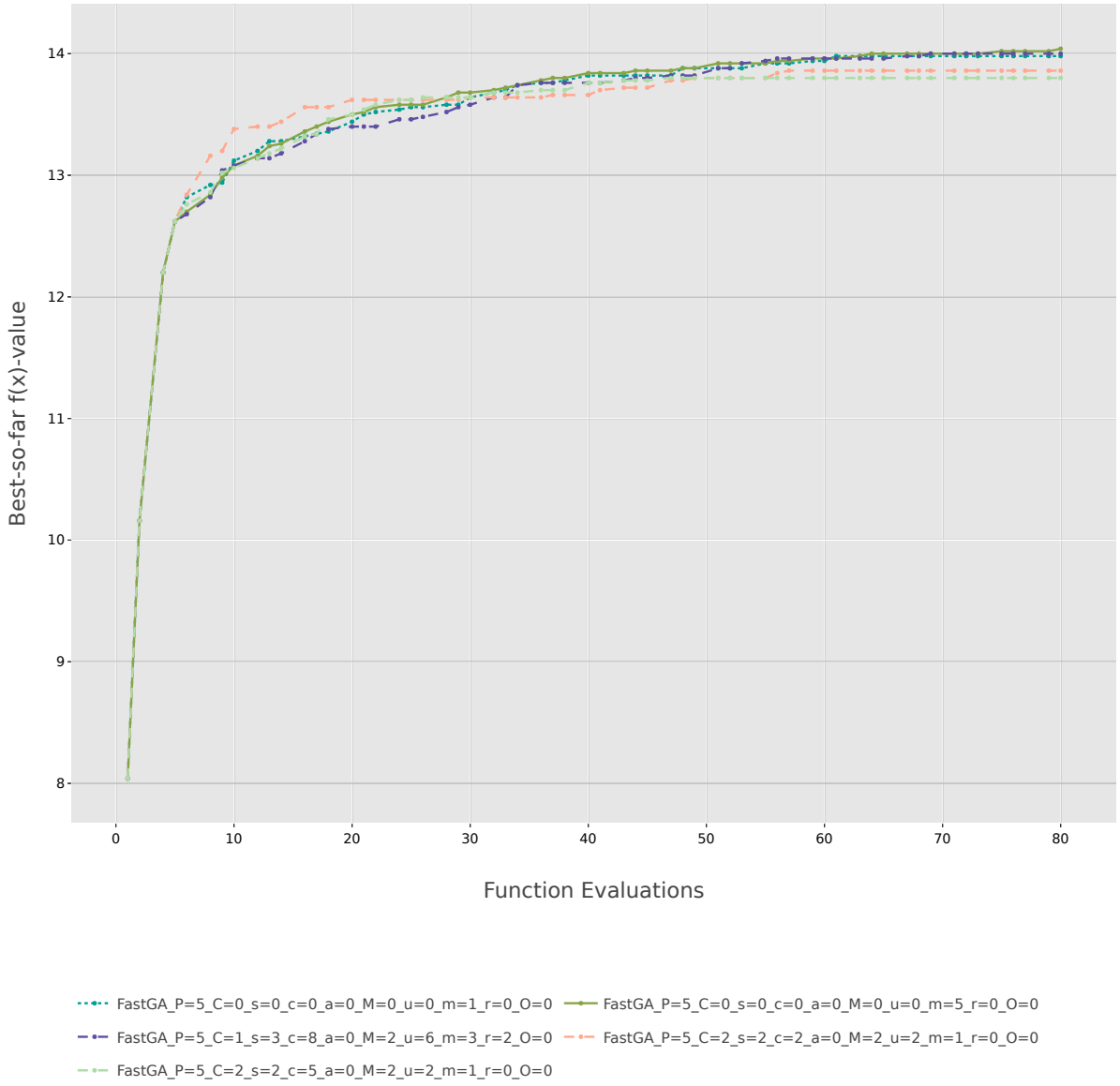

Figure 8: Convergence plot of baseline algorithms and elite, for problem 3.

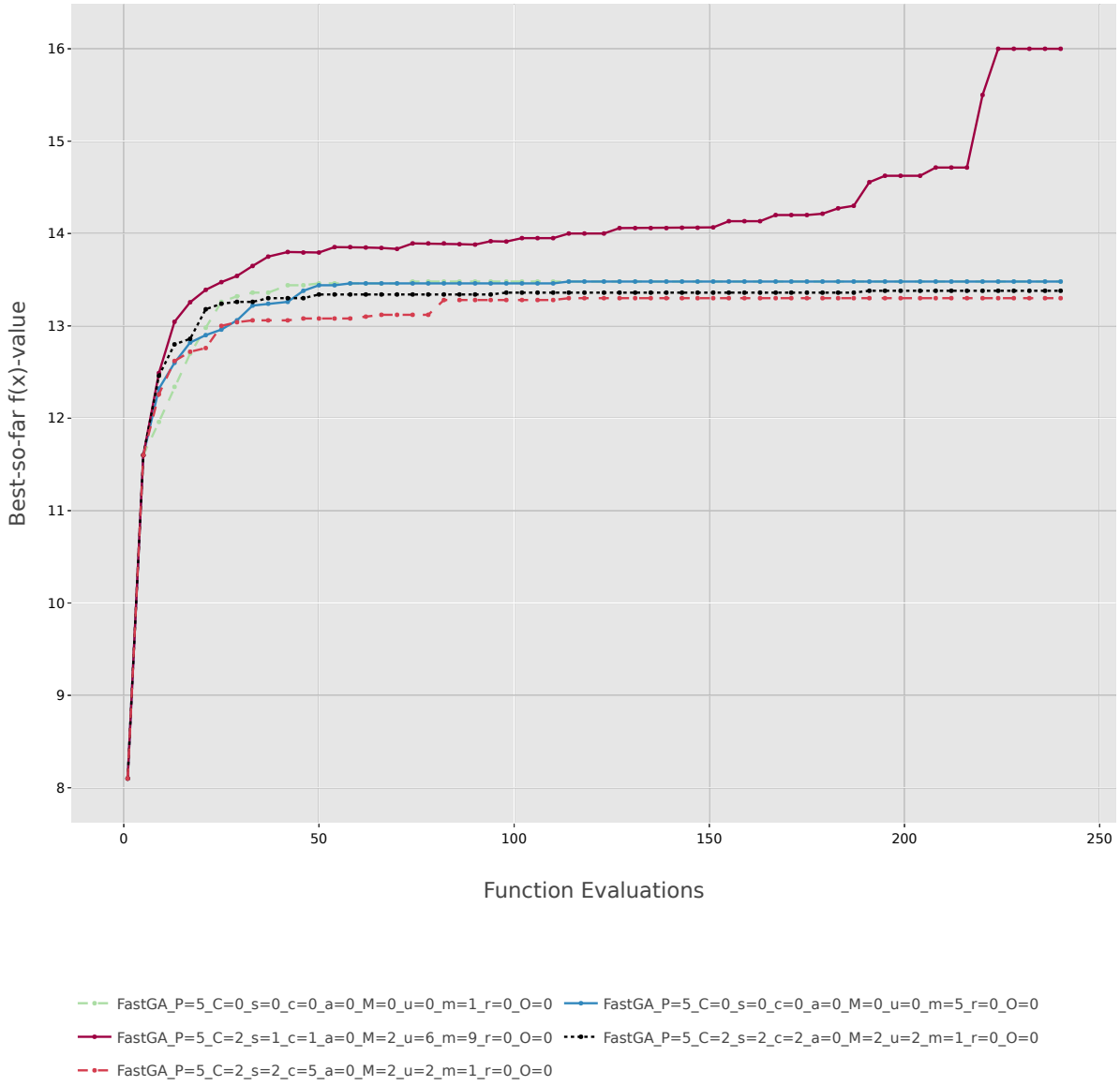

Figure 9: Convergence plot of baseline algorithms and elite, for problem 4.

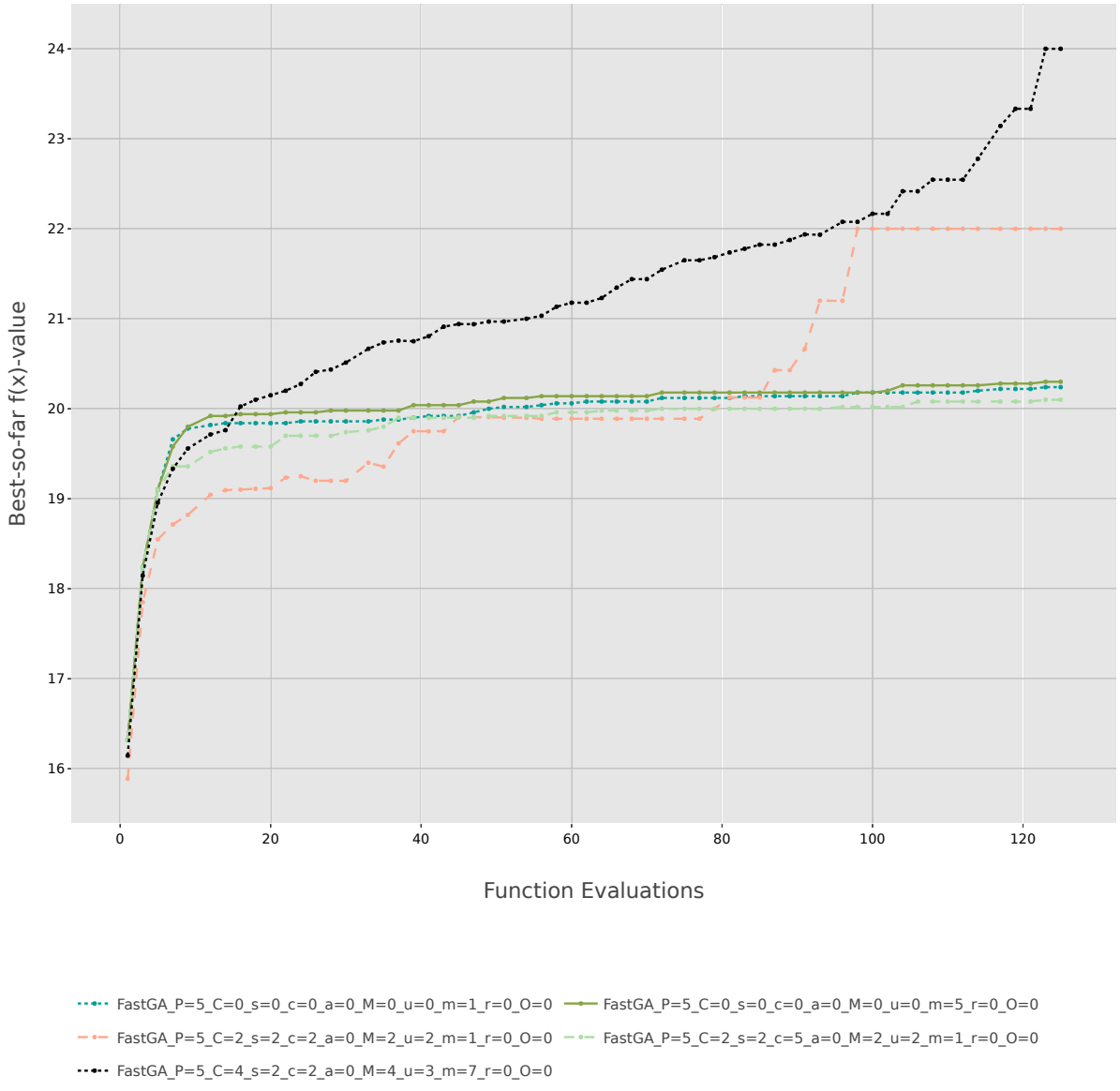

Figure 10: Convergence plot of baseline algorithms and elite, for problem 5.

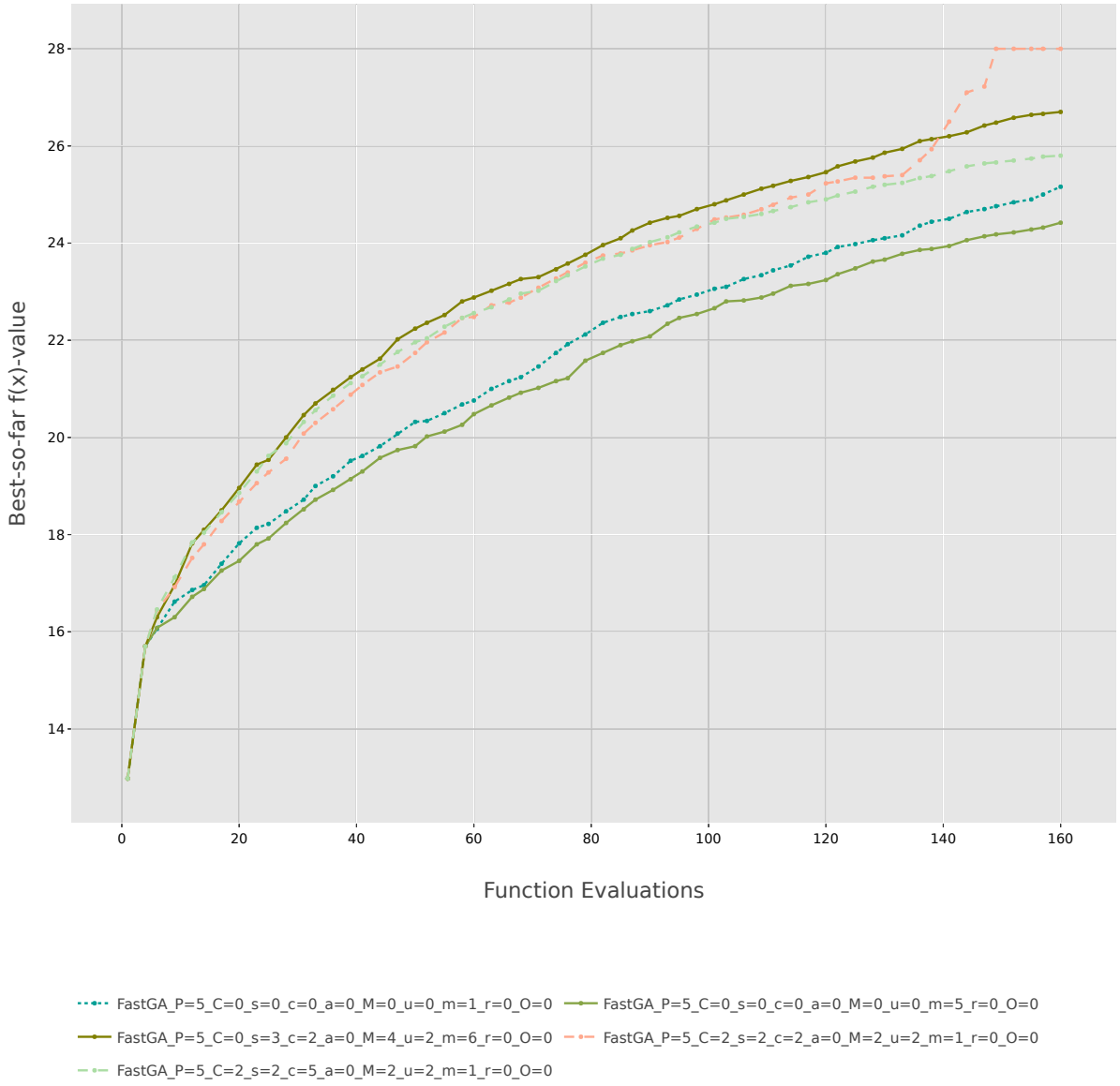

Figure 11: Convergence plot of baseline algorithms and elite, for problem 6.

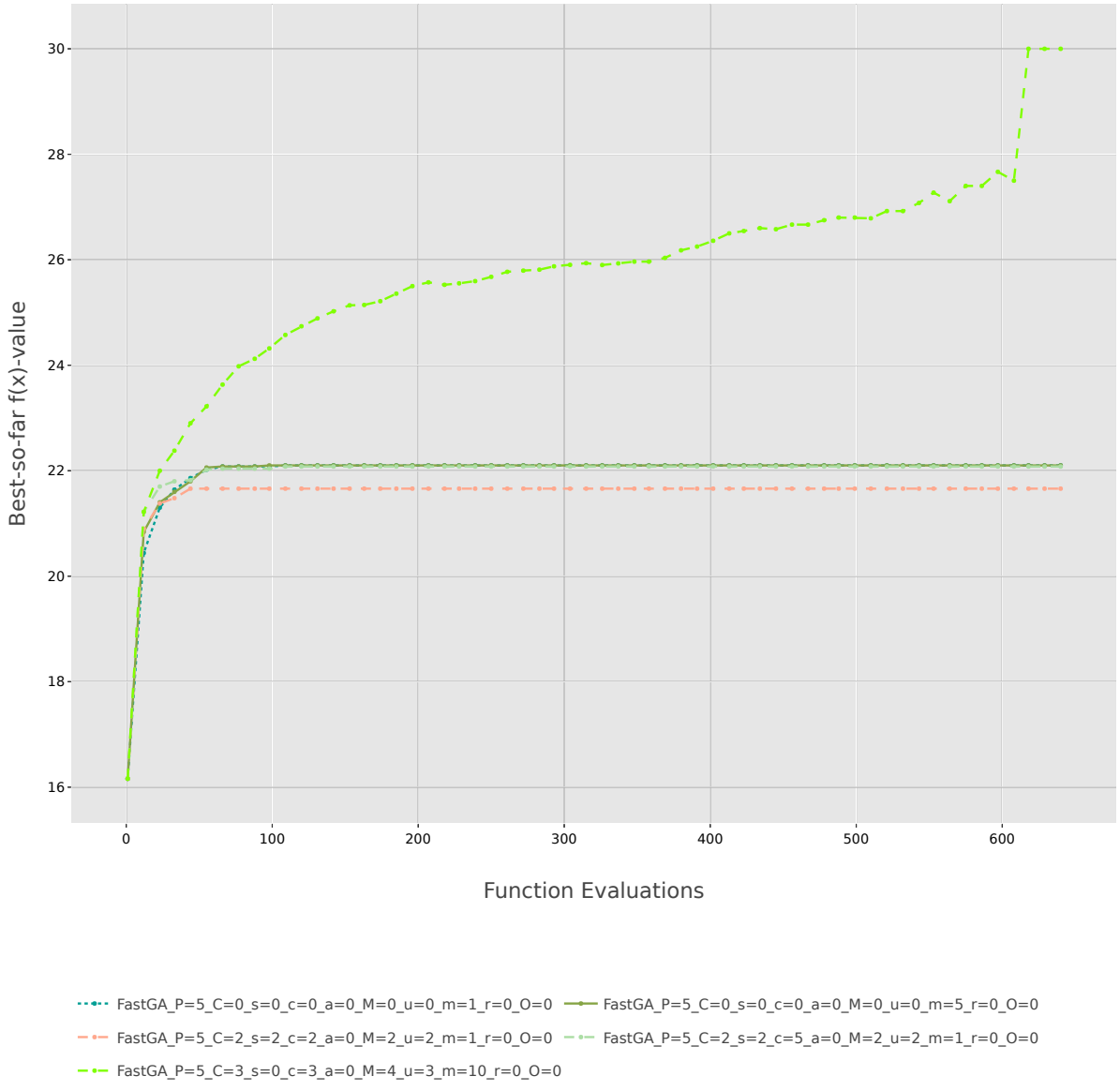

Figure 12: Convergence plot of baseline algorithms and elite, for problem 7.

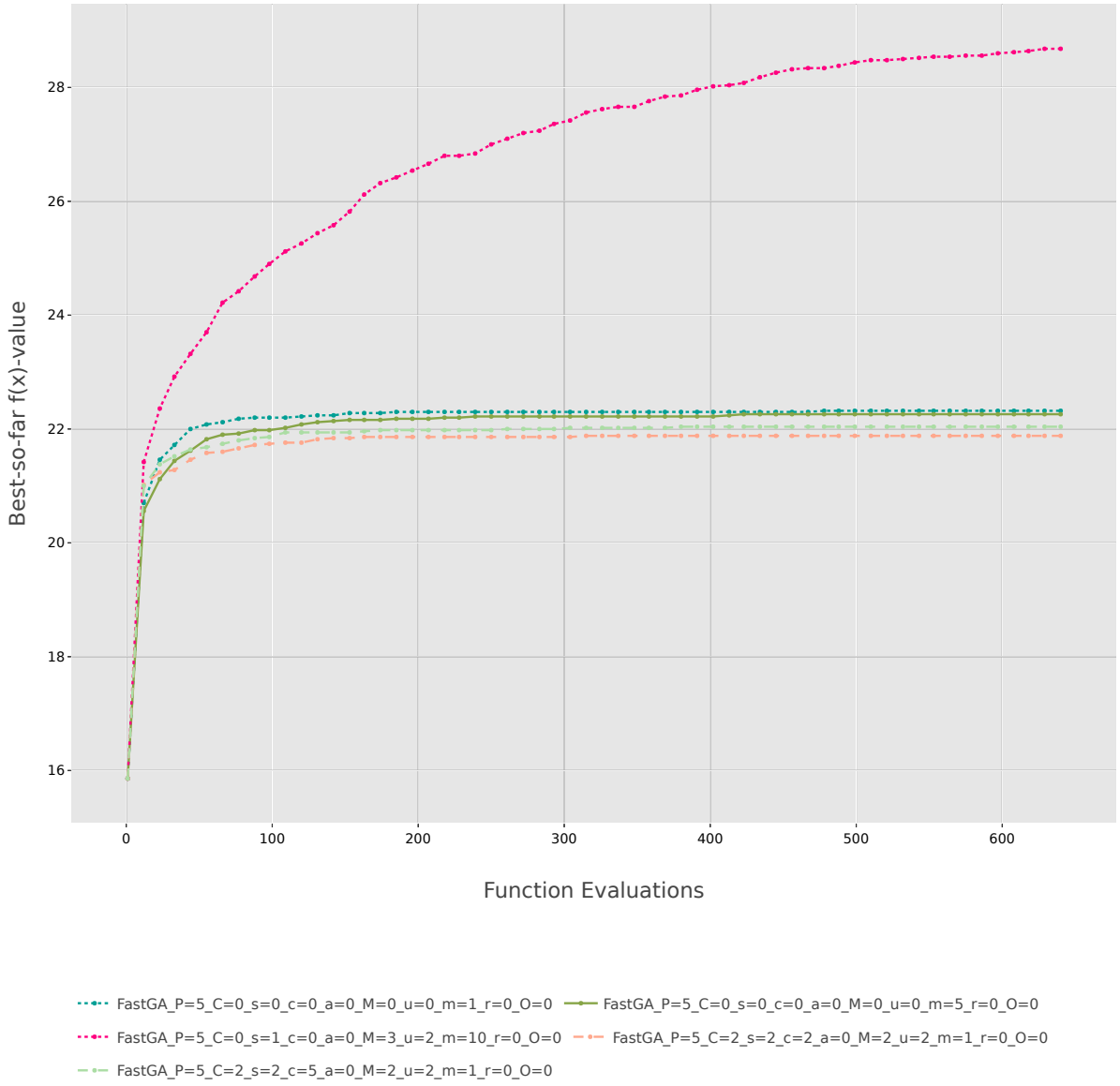

Figure 13: Convergence plot of baseline algorithms and elite, for problem 8.

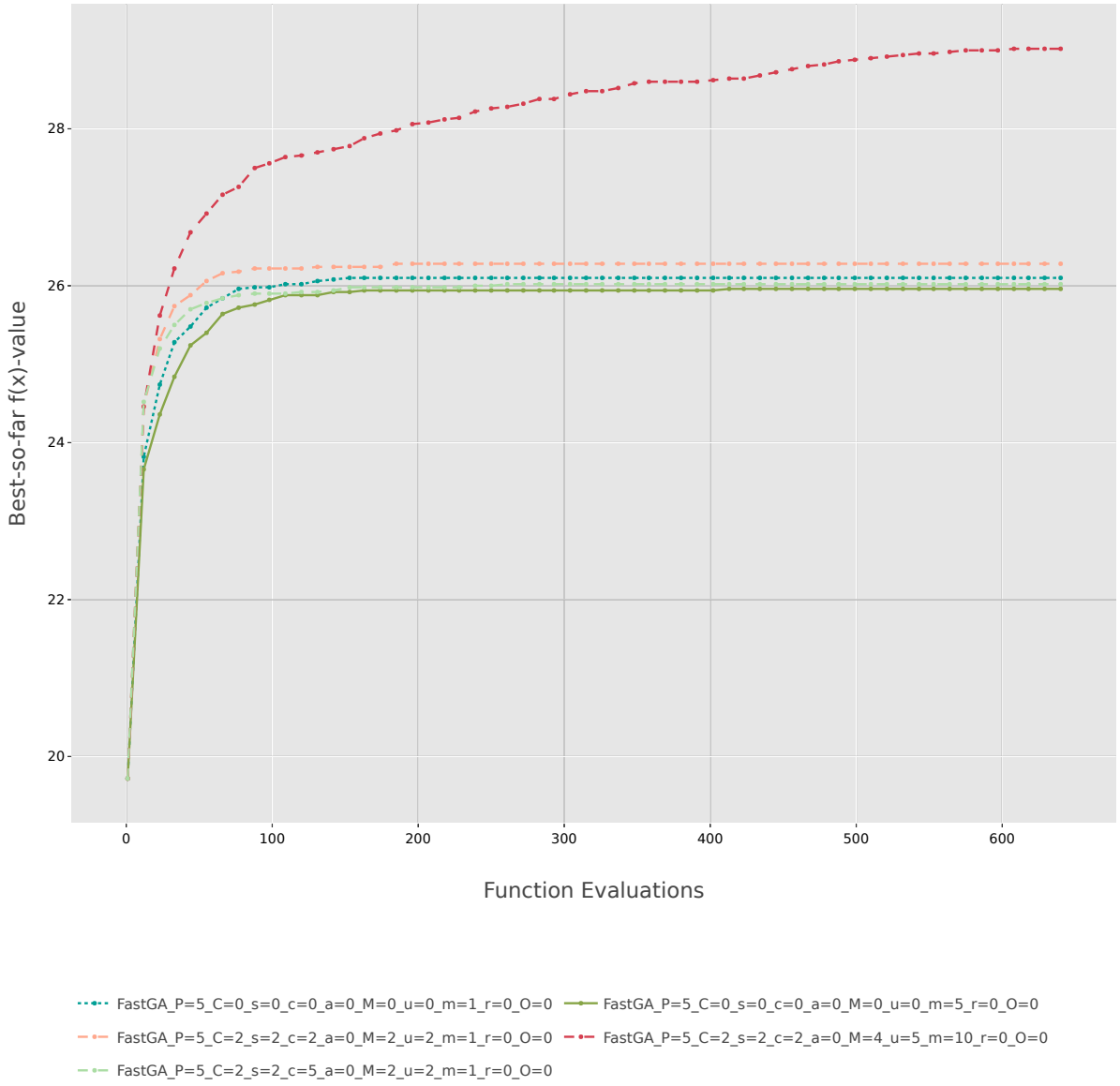

Figure 14: Convergence plot of baseline algorithms and elite, for problem 9.

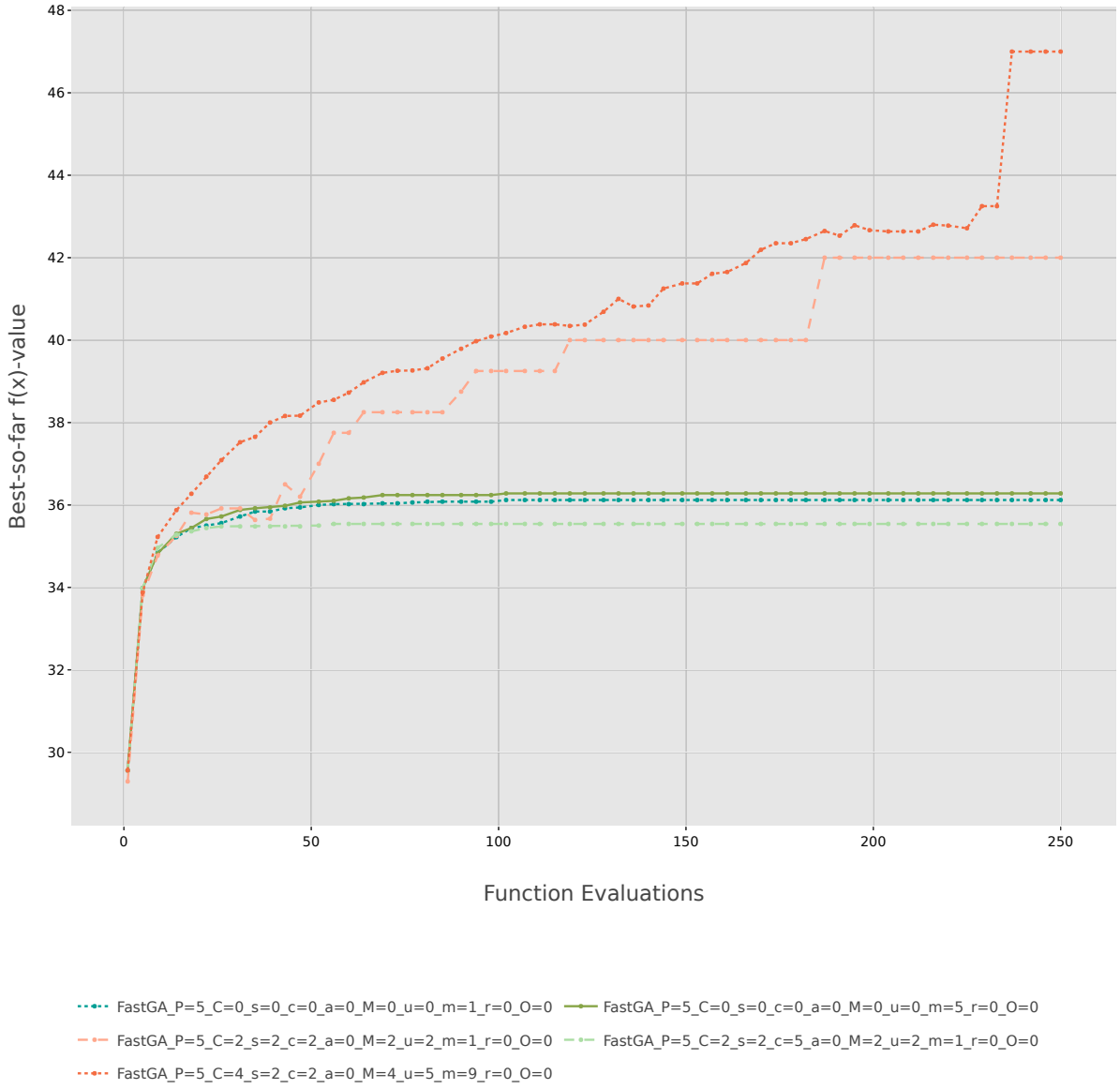

Figure 15: Convergence plot of baseline algorithms and elite, for problem 10.

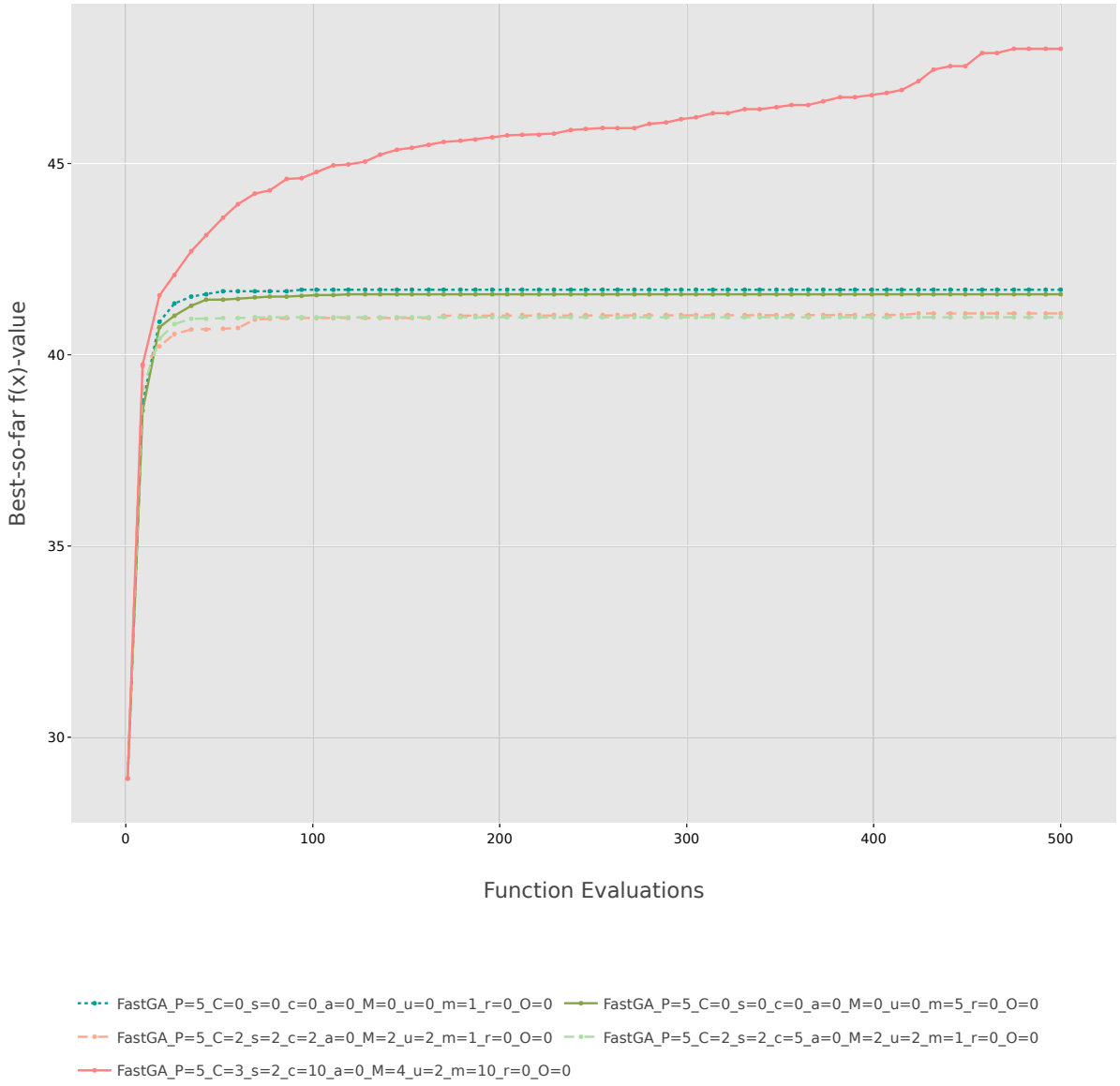

Figure 16: Convergence plot of baseline algorithms and elite, for problem 11.

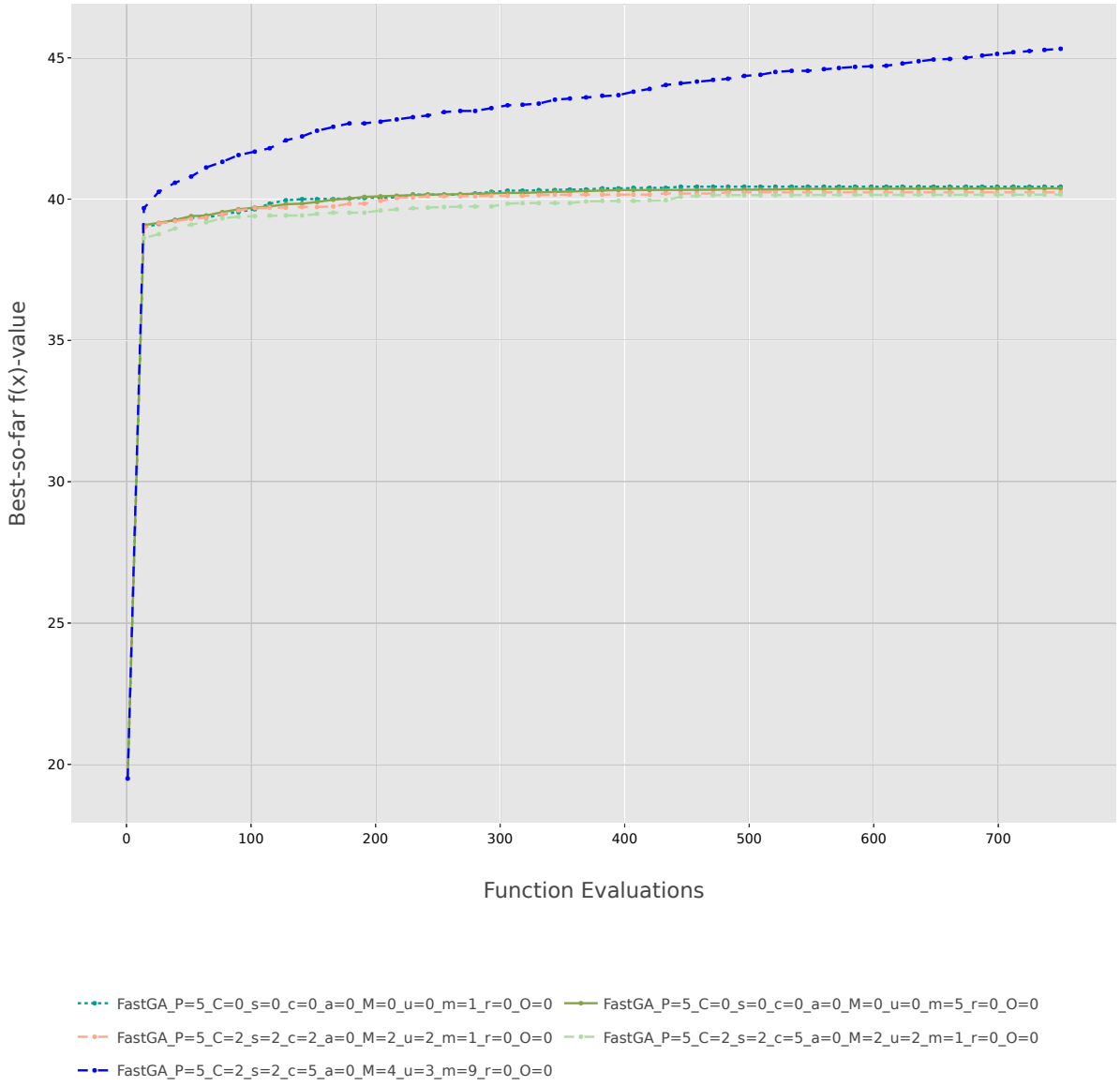

Figure 17: Convergence plot of baseline algorithms and elite, for problem 12.

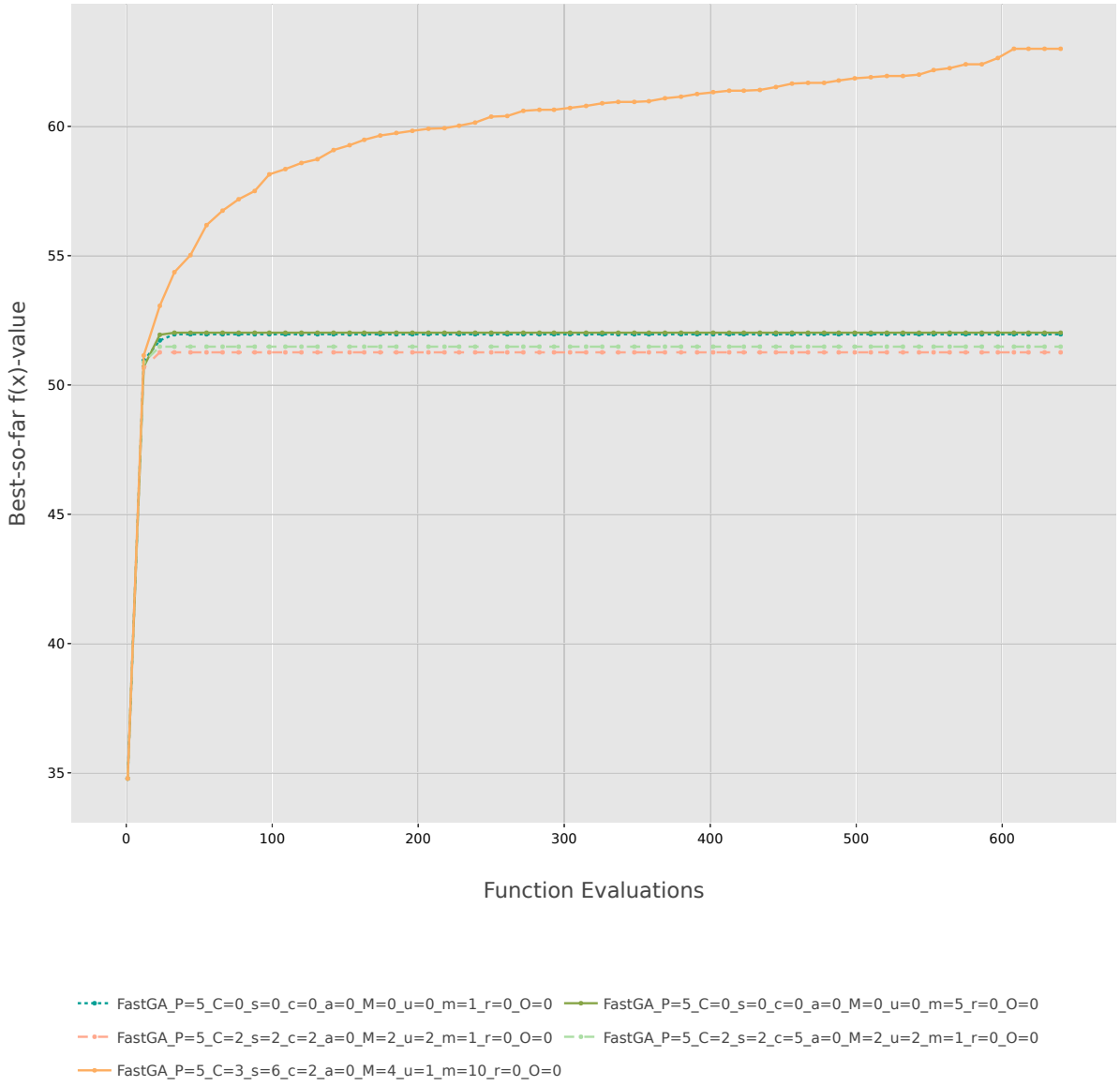

Figure 18: Convergence plot of baseline algorithms and elite, for problem 13.

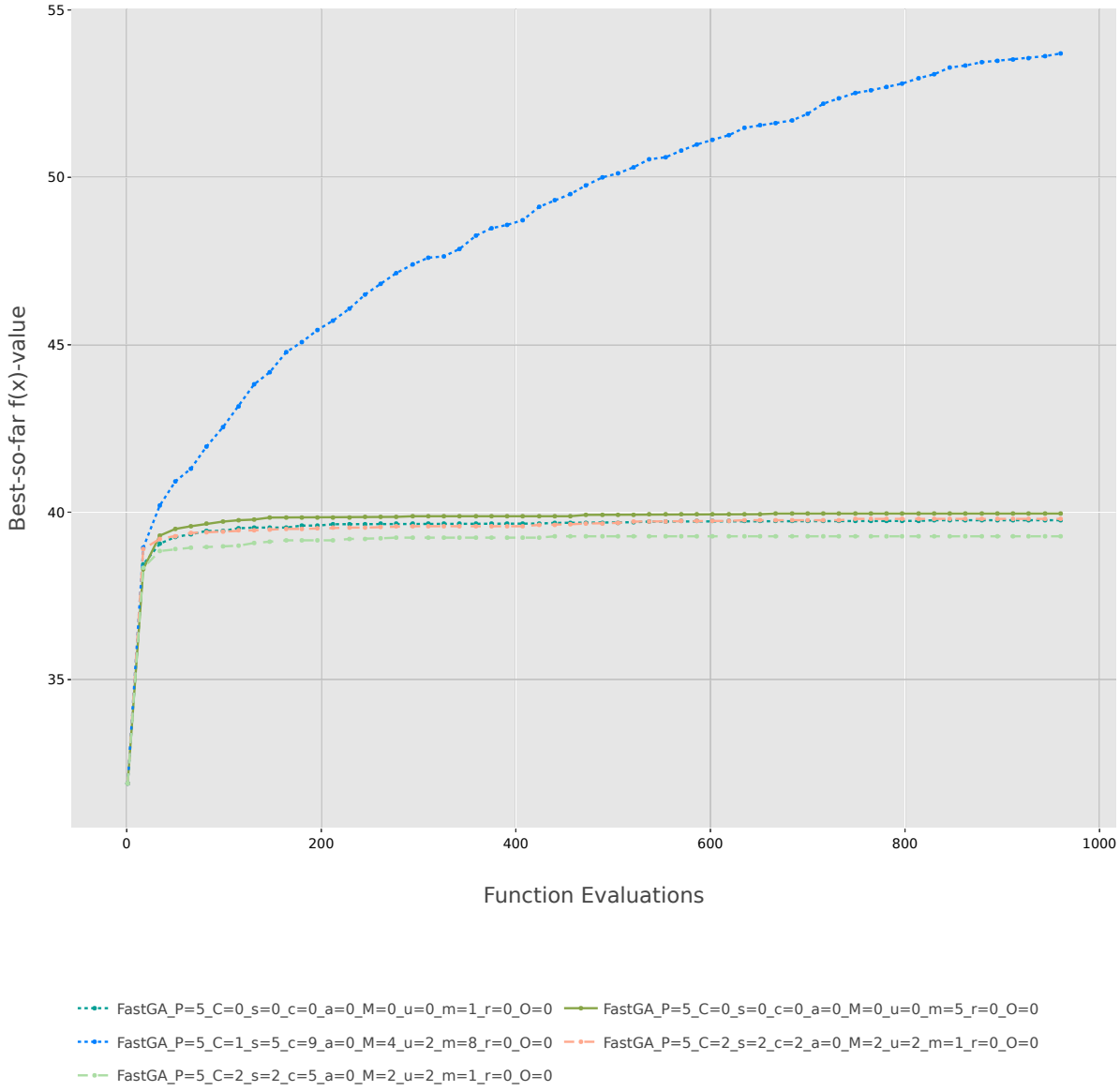

Figure 19: Convergence plot of baseline algorithms and elite, for problem 14.

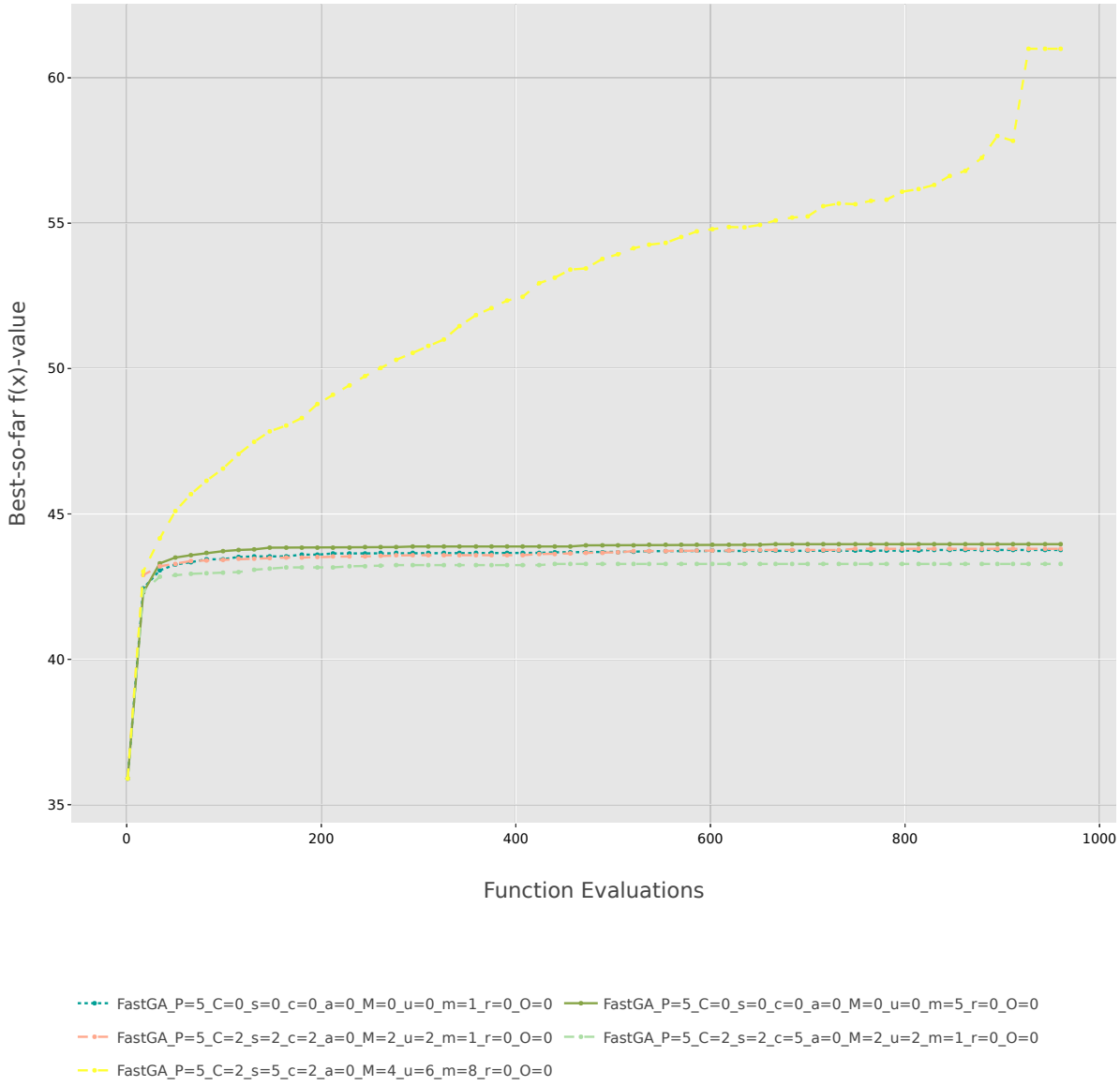

Figure 20: Convergence plot of baseline algorithms and elite, for problem 15.

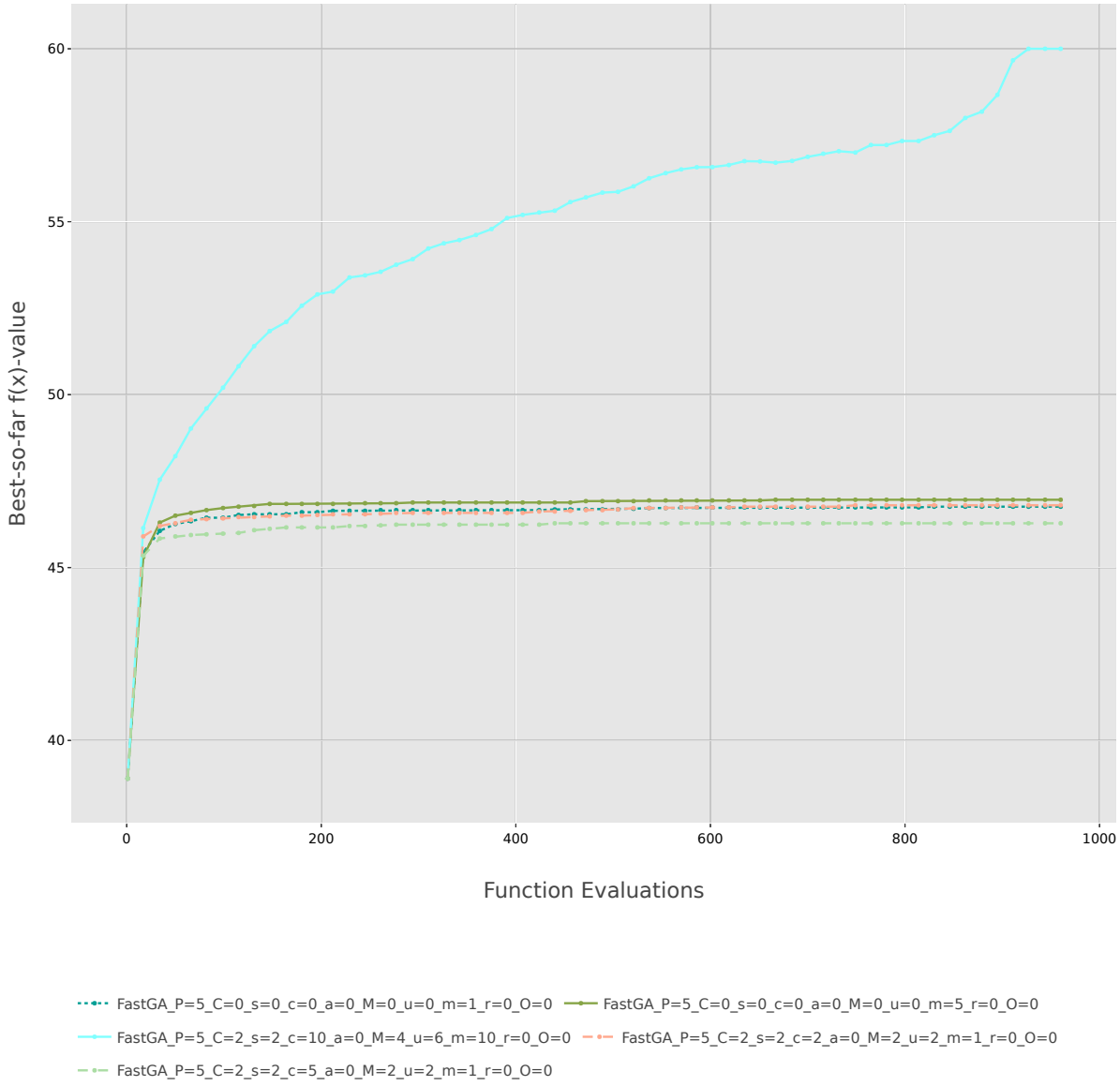

Figure 21: Convergence plot of baseline algorithms and elite, for problem 16.

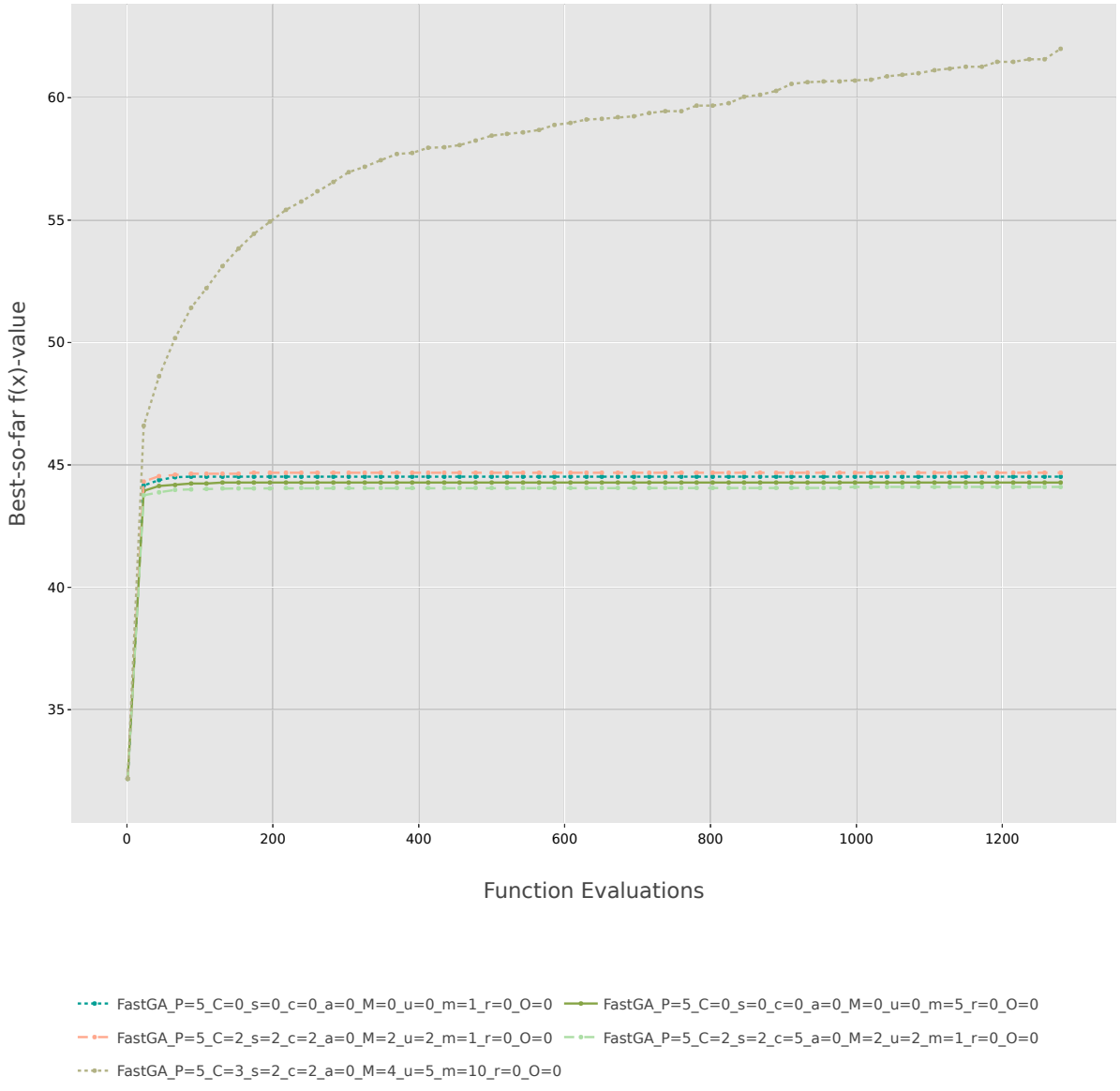

Figure 22: Convergence plot of baseline algorithms and elite, for problem 17.

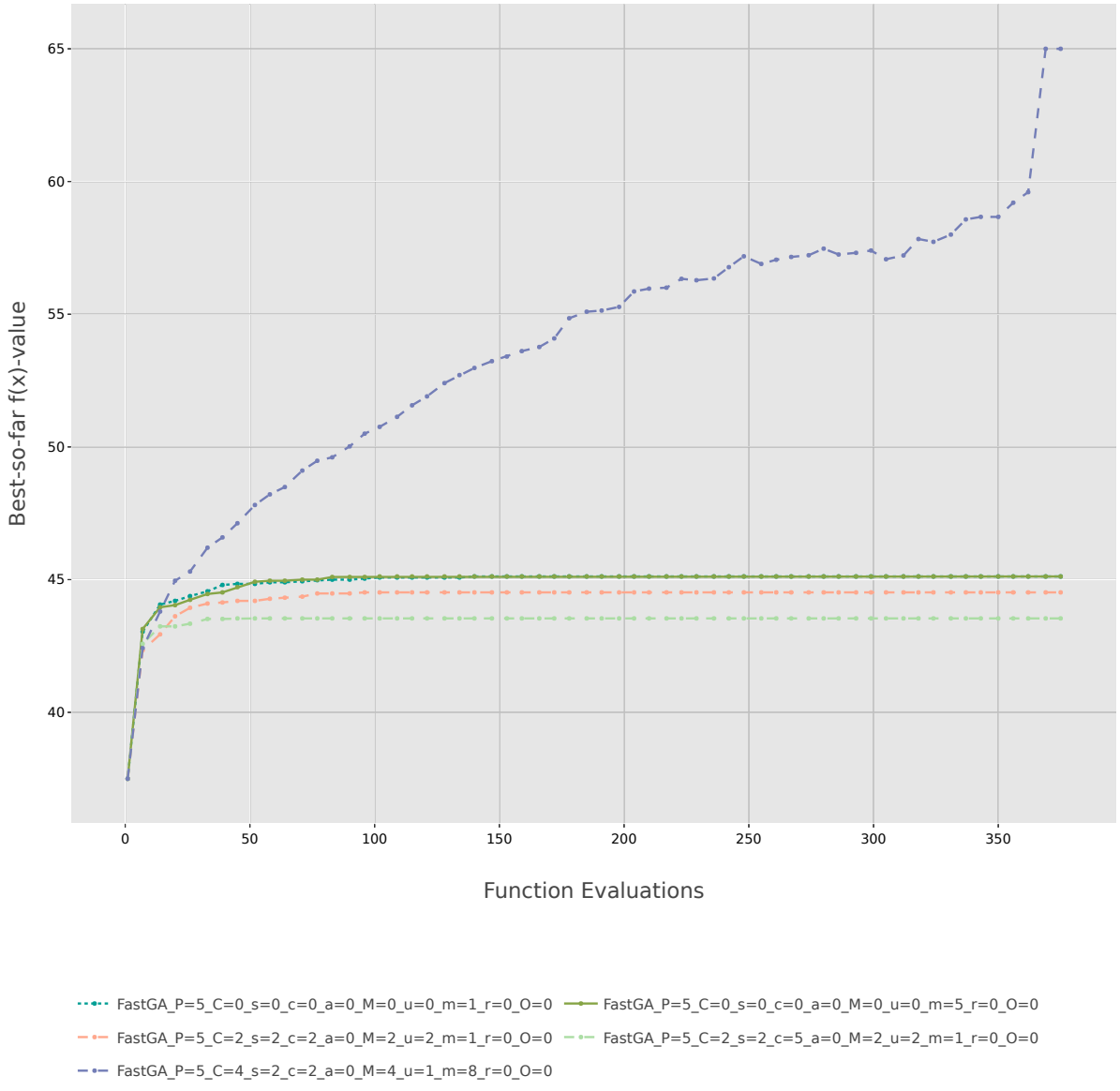

Figure 23: Convergence plot of baseline algorithms and elite, for problem 18.

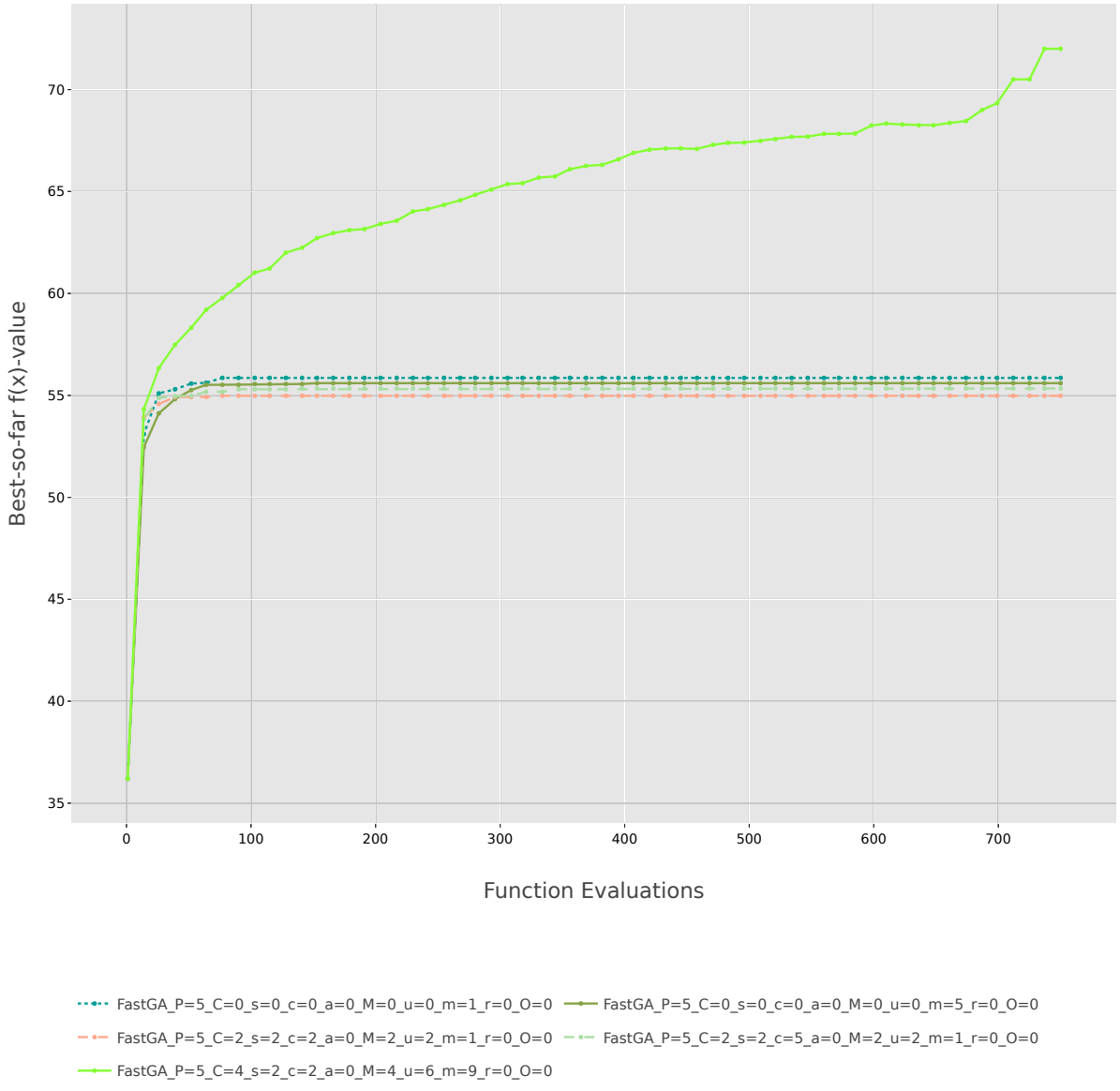

Figure 24: Convergence plot of baseline algorithms and elite, for problem 19.