LLaMEA: A Large Language Model Evolutionary Algorithm for Automatically Generating Metaheuristics

Stein, N. van; Bäck, T.H.W.

LLaMEA: A Large Language Model Evolutionary Algorithm for Automatically Generating Metaheuristics

Niki van Stein, *Member, IEEE*, and Thomas Bäck, *Fellow, IEEE*

***Abstract*—Large language models (LLMs), such as GPT-4 have demonstrated their ability to understand natural language and generate complex code snippets. This article introduces a novel LLM evolutionary algorithm (LLaMEA) framework, leveraging GPT models for the automated generation and refinement of algorithms. Given a set of criteria and a task definition (the search space), LLaMEA iteratively generates, mutates, and selects algorithms based on performance metrics and feedback from runtime evaluations. This framework offers a unique approach to generating optimized algorithms without requiring extensive prior expertise. We show how this framework can be used to generate novel closed box metaheuristic optimization algorithms for box-constrained, continuous optimization problems automatically. LLaMEA generates multiple algorithms that outperform state-of-the-art optimization algorithms (covariance matrix adaptation evolution strategy and differential evolution) on the 5-D closed box optimization benchmark (BBOB). The algorithms also show competitive performance on the 10- and 20-D instances of the test functions, although they have not seen such instances during the automated generation process. The results demonstrate the feasibility of the framework and identify future directions for automated generation and optimization of algorithms via LLMs.**

***Index Terms*—Automated code generation, evolutionary computation (EC), large language models (LLMs), metaheuristics, optimization.**

I. Introduction

FOR DECADES, algorithms for finding near-optimal solution candidates to global optimization problems of the form

$$\text{minimize } \mathcal{F} : \mathcal{S} \to \mathbb{R} \tag{1}$$

where $\mathcal{S} \subseteq \mathbb{R}^d$ and $\mathcal{S} = \times_{i=1}^{d}[l_i, u_i]$ is defined by box constraints ($l_i < u_i \quad \forall\, l_i, u_i \in \mathbb{R}$), have been developed based on inspirations gleaned from nature. Famous examples include the use of biological evolution in the field of evolutionary computation (EC) [1], [2] (with examples, such as genetic algorithms and evolutionary strategies), the swarming behavior of bird-like objects in particle swarm optimization [3], or the foraging behavior of ants in ant colony optimization [4]. For the highly advanced variants of such algorithms, impressive results have been reported for solving real-world optimization problems [5], [6], [7].

However, the number of metaphor-inspired algorithms that have been proposed by researchers, often as relatively small variations of existing methods or claimed as a completely new branch, is very large (e.g., [8] and [9] mentions more than 500 methods). A systematic empirical benchmarking of such methods against the state-of-the-art, as exemplified in [10], is typically not performed.

Realizing that the laborious, expert driven approach for improving existing and inventing new algorithms has become quite inefficient, researchers have recently started to develop modular frameworks for arbitrarily combining components (*modules*) from algorithm classes into new variants—thereby creating combinatorial algorithm design spaces in which thousands to millions of algorithms can be generated, benchmarked, and optimized. Examples include the modular covariance matrix adaptation evolution strategy (CMA-ES) [11], [12], modular differential evolution (DE) [13], modular particle swarm optimization [14], and a recent overview that summarizes all such approaches toward automated design [15].

Although they often provide algorithm design spaces of millions of potential module combinations plus their hyperparameter search spaces, modular frameworks also need to be created by experts in the field, by carefully selecting the included modules and providing the full infrastructure for configuring and searching through such algorithm spaces. Moreover, the modular approaches are limited to the design space created by the choices that have been made by the experts when selecting modules for inclusion.

In this work, we propose to overcome such limitations by using large language models (LLMs) within an evolutionary loop for automatically and iteratively evolving and optimizing the program code of such metaheuristic optimization algorithms for solving the optimization problem in (1). To evaluate the generated algorithms, we use a specific optimization benchmarking tool (IOHprofiler, consisting of the IOHexperimenter [16] module for systematically running algorithms on benchmark functions, and IOHanalyzer [17] for statistically analyzing the results of the runs) to automatically

Received 5 June 2024; revised 6 August 2024, 27 September 2024, and 11 November 2024; accepted 11 November 2024. Date of publication 13 November 2024; date of current version 2 April 2025. This article was approved by Associate Editor G. Iacca. *(Corresponding author: Niki van Stein.)*

The authors are with the Leiden Institute of Advanced Computer Science, Leiden University, 2300 RA, The Netherlands (e-mail: n.van.stein@liacs.leidenuniv.nl).

This article has supplementary downloadable material available at https://doi.org/10.1109/TEVC.2024.3497793.

Digital Object Identifier 10.1109/TEVC.2024.3497793

evaluate the quality of the generated optimization algorithms and to provide a corresponding feedback to the LLM. The specific contributions of this work are as follows.

1) We present the LLM-evolutionary algorithm (LLaMEA), an evolutionary algorithm that uses an LLM to automatically generate and optimize high-quality metaheuristic optimization algorithms for solving the optimization problem as stated in (1). LLaMEA combines in-context learning, a precise task prompt (including a sample algorithm, code interface requirements, and mutation instruction), error handling and two different selection strategies in a novel way. To the best of our knowledge, this is the first time that new metaheuristics for continuous optimization problems are successfully generated and optimized by LLMs to reach and exceed state-of-the-art optimization algorithm performance levels. In particular, we show that the generated algorithms are competitive with three baseline algorithms, namely a) the CMA-ES with default parameters; b) a hyperparameter-optimized CMA-ES; and c) DE.
2) We couple LLaMEA with the benchmarking tool, IOHexperimenter [17], to automatically evaluate the quality of the generated optimization algorithm, for providing feedback to the LLM. IOHexperimenter is a well-established benchmarking framework that supports evaluating optimization heuristics for continuous optimization problems (see [18], [19], [20], [21], [22]). It allows for comparing a new optimization algorithm automatically against a well-known set of state-of-the-art algorithms on a wide set of benchmark functions in a statistically sound way. It is only this automated benchmarking approach that allows us to close the evolutionary loop that includes the LLM and to run it automatically. We specifically use a standard set of 24 benchmark test problems (with multiple instances of each problem), the so-called closed box optimization benchmark (BBOB) [23], for representing the class of continuous optimization problem as stated in (1).
3) To evaluate the quality of a newly generated metaheuristic across a set of benchmark functions by a single, aggregated performance measure, we use the area over the convergence curve (AOCC) measure [24]. This enables the LLM to generate metaheuristics that perform well across a set of problem instances, rather than on a single problem, only.

We also would like to emphasize that the goal of this research is not to generate a specific algorithm for combinatorial or constraint optimization problems, but rather to combine the use of modern, sound benchmarking techniques for measuring the performance of algorithms with the generative capabilities of LLaMEA for achieving reliable results for the continuous optimization task defined in (1) and for outperforming state-of-the-art metaheuristics on such problems.

In the remainder of this article, we first discuss the state-of-the-art in LLM-based algorithm generation for direct, global optimization (Section II). We then introduce the newly proposed LLaMEA in Section III, and our experimental setup in Section IV. The experimental results are presented and discussed in Section V. We also analyze the best algorithms found in Section VI, as we are striving to understand the resulting algorithms proposed by LLaMEA, and why they might perform so well. The conclusions are presented in Section VII.

II. Related Work

The integration of LLMs into the optimization domain has recently received significant attention, resulting in various innovative methodologies. We can distinguish three classes of solutions that integrate LLMs and EC; *Prompt optimization* by EC methods, *LLMs* as the *EC method*, and optimization of *code generation*.

Prompt Optimization: Using optimization algorithms, such as genetic algorithms for optimizing prompts (EvoPrompt) [25] has shown impressive results in outperforming human-engineered prompts. This idea of directly evolving the prompts by an evolutionary algorithm has also recently been used to demonstrate an automated engineering design optimization loop for finding 3-D car shapes with optimal aerodynamic performance [26].

These approaches are limited, however, as they are usually based on a fixed prompt template and a finite set of strings as building blocks and need evolutionary operators that work on such patterns and strings. To overcome such limitations, an LLM can be used to generate the prompts for an LLM, by asking the prompting LLM to propose and iteratively improve a prompt, based on the (quantifiable) feedback concerning the quality of the answer generated by the prompted LLM. The automated prompt engineer (APE) [27] implements this loop by employing an iterative Monte Carlo search approach for prompt generation, in which the LLM is asked to generate new prompts similar to those with high scores. The authors illustrate the benefits of APE on a large set of instruction induction tasks [28] and by improving a "Chain-of-Thought" prompt ("Let's think step by step") from [29].

LLMs as EC: The direct application of LLMs as evolution strategies for optimization tasks represents another innovative approach, named EvoLLM [30]. In this proposal, the LLM generates the means of uni-variate normal distributions which are then used as sampling distributions for proposing solution vectors for closed box optimization tasks. To guide the LLM toward improvements, the best m solution vectors from the best k generations, each, are provided to the LLM, and it is prompted to generate the new mean values of the sampling distribution. A method, such as EvoLLM leverages the LLM to perform the sampling based on the information summarized above, and it shows reasonable performance on a subset of eight of the BBOB set of benchmarking functions for continuous optimization [31] for one 5-D instance of each problem. EvoLLM uses the idea of *in-context learning* [32], [33], which works by providing a set of examples and their corresponding scores to the LLM, which is then prompted to generate a score for a newly presented example.

Code Generation: The recently proposed *FunSearch* approach (searching in the function space) [34] closes the loop

and combines the LLM with a systematic evaluation of the quality of the generated programs, resulting in new solution algorithms for combinatorial optimization problems (the cap-set problem and bin packing) after 2.5 million calls of the LLM. In this approach, a distributed island-based Evolutionary Algorithm is used for maintaining diversity and prompt construction based on the already generated programs. In the approach called algorithm evolution using LLM (AEL) [35] and also the evolution of heuristics (EoH) [36], ideas from evolutionary algorithms are used explicitly to design novel optimization heuristics. AEL and EoH treat each algorithm as a solution candidate in a population, evolving them by asking the LLM to perform crossover and mutation operators on the heuristics generated by the LLM. These approaches have shown superior performance over traditional heuristic methods and domain-specific models in solving small instances of the traveling salesperson problem (TSP). The AEL approach, however, is very specific to the TSP and cannot be applied (trivially) to the continuous optimization problem as in (1), which is the subject of our work. The EoH approach can be generalized to the continuous optimization problem with some small modifications, and thus be compared with our proposed approach. Note that, the EoH approach was designed to generate and optimize small pieces of code (single functions) and the closed box optimization algorithms we aim to generate can be complex, relatively large and consisting of a complete class with variables and functions.

The concept of recursive self-improvement, where systems iteratively improve their performance by refining their own processes, has been a focal point in AI research. The self-taught optimizer (STOP) [37] framework exemplifies this by using LLMs to recursively enhance code generation. STOP begins with a seed improver that uses an LLM to generate and evaluate candidate solutions, iteratively refining its own scaffolding to improve performance across various tasks. This framework highlights the potential of LLMs to act as meta-optimizers, capable of self-improvement without altering their underlying model parameters.

When it comes to leveraging LLMs for the design of closed box optimization metaheuristics, results from a manual prompting approach with GPT-4 for selecting, explaining, and combining components of such algorithms have illustrated the capabilities of LLMs for generating such algorithms [38]. Automated generation and evaluation of their performance was not performed, however.

Overall, these advancements illustrate the diverse use of LLMs in an optimization context. However, we observe that the application of LLMs for generating novel direct global optimization algorithms from scratch is still at a very early stage. Often, the performance evaluation of the LLM (e.g., in EvoLLM) or the generated algorithms (e.g., in AEL) focuses on small test problems or does not benchmark against state-of-the-art algorithms. In-context learning, as in EvoLLM, and algorithm generation, as in STOP or AEL, are usually not combined, and the generation of new algorithms is typically based on mutation and crossover requests to the LLM, with a focus on solving combinatorial problems.

In our approach, described in Section III, we propose to overcome such limitations by presenting a framework that can be used to generate any kind of algorithm as long there is a way to assess its quality automatically and it stays within the token limits of current LLMs. To achieve this goal, we combine in-context learning and an evolutionary algorithm-like iteration loop that allows the LLM to either refine (i.e., *mutate*) the best-so-far algorithm or redesign it completely. The proposed approach not only uses numerical feedback scores (as AEL and EoH do), but also debugging information or other textual feedback that can be automatically provided to steer the search toward better solutions. We combine this approach with a sound performance evaluation of the generated algorithms, based on the well-established IOHprofiler benchmark platform.

III. LLaMEA

The proposed LLaMEA framework (Fig. 1) consists of four primary steps, which are iteratively repeated, following a similar structure as an evolutionary Algorithm with one parent and one child with an *initialization* step and a general optimization loop of *evaluation* (in this case, by using IOH-experimenter [16] to run the generated algorithms on a well defined set of benchmarking test functions) and *refinement (mutation) or redesign* of the algorithm. The *selection strategy* determines whether only improvements are accepted (in case of a $(1 + 1)$-strategy) or the newly generated algorithm is always accepted (in case of a $(1, 1)$-strategy). Since both selection strategies are integrated into LLaMEA, we use the term $(1 \overset{+}{,} 1)$-EA for describing both selection strategies in one notation.

Our approach, which we therefore abbreviate as $(1 \overset{+}{,} 1)$-LLaMEA, is presented in more detail in Algorithm 1. Here, initialization is performed in lines 1–6 by prompting the LLM with a task description prompt S (see Section III-A for a description of S), evaluating the quality $f(a_t)$ of the generated program code (text) $a_t \in \mathcal{A}$, and memorizing the initial program a_t, its mean quality y_t and standard deviation of quality σ_t over multiple runs, and potentially some runtime error information e_t as best solution $(a_b, y_b, \sigma_b, e_b)$ (see Section III-C for a variation of this approach, in which more detailed information is provided to LLaMEA).

The while-loop (lines 7–20) iterates through T (total budget) LLM calls to generate a new program each time (line 13). The prompt construction in lines 9 (in case of elitist $(1 + 1)$-selection) or line 11 (in case of nonelitist $(1, 1)$-selection) is the essential step, generating a prompt F (see Section III-D for a description of the template of F) that consists of a concatenation[1] of the following information: 1) task description S; 2) the algorithm names and mean quality values of all previously generated algorithms; 3) the current best algorithm's a_b (in case of $(1+1)$-selection) or most recent algorithm's a_t (in case of $(1, 1)$-selection) complete information (code, mean quality and standard deviation, execution error information); and 4) a short task prompt F_0, with an instruction to the LLM. The new algorithm is generated by the LLM (line 13) and its quality evaluated catching any runtime errors on the way (line 14), setting y_{t+1} explicitly to zero (worst possible quality) if

[1]We use tuple notation here for clarity to show the components that compose the prompt string.

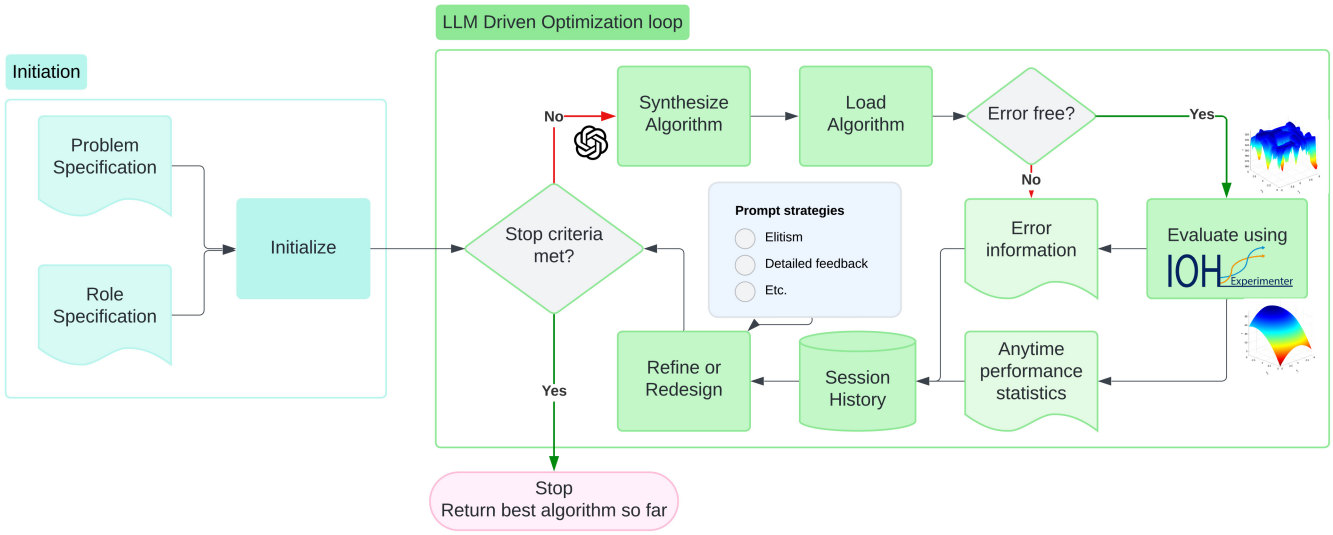

Fig. 1. Summary of the proposed LLM driven algorithm design framework LLaMEA. Full details of all steps are provided in the corresponding sections.

Algorithm 1 $(1\overset{+}{,}1)$-LLaMEA

1: $S \leftarrow$ task-prompt ▷ Task description prompt
2: $F_0 \leftarrow$ task-feedback-prompt ▷ Feedback prompt after each iteration
3: $t \leftarrow 0$
4: $a_t \leftarrow LLM(S)$ ▷ Initialize by generating first parent program
5: $(y_t, \sigma_t, e_t) \leftarrow f(a_t)$ ▷ Evaluate mean quality and std.-dev. of first program and catch errors if occurring
6: $a_b \leftarrow a_t; y_b \leftarrow y_t; \sigma_b \leftarrow \sigma_t; e_b \leftarrow e_t$ ▷ Remember best-so-far
7: **while** $t < T$ **do** ▷ Budget not exhausted
8: **if** $(1+1)$ **then** ▷ $(1+1)$-Variant: Construct new prompt, using best-so-far algorithm
9: $F \leftarrow (S, ((\text{name}(a_0), y_0), \ldots, (\text{name}(a_t), y_t)), (a_b, y_b, \sigma_b, e_b), F_0)$
10: **else** ▷ $(1,1)$-Variant: Construct new prompt, using latest parent algorithm
11: $F \leftarrow (S, ((\text{name}(a_0), y_0), \ldots, (\text{name}(a_t), y_t)), (a_t, y_t, \sigma_t, e_t), F_0)$
12: **end if**
13: $a_{t+1} \leftarrow LLM(F)$ ▷ Generate offspring algorithm by mutation
14: $(y_{t+1}, \sigma_{t+1}, e_{t+1}) \leftarrow f(a_{t+1})$ ▷ Evaluate offspring algorithm, catch errors
15: **if** $e_{t+1} \neq \emptyset$ **then** $y_{t+1} = 0$ ▷ Errors occurred
16: **end if**
17: **if** $y_{t+1} \geq y_t$ **then** $a_b \leftarrow a_{t+1}; y_b \leftarrow y_{t+1}; \sigma_b \leftarrow \sigma_{t+1}; e_b \leftarrow e_{t+1}$ ▷ Update best
18: **end if**
19: $t \leftarrow t+1$ ▷ Increase evaluation counter
20: **end while**
21: **return** a_b, y_b ▷ Return best algorithm and its quality

runtime errors occurred (line 15). The best-so-far algorithm is updated if an improvement was found (line 17).

It should be noted that in the general description of Algorithm 1 given above, we do not specify the quality measure $f : \mathcal{A} \rightarrow \mathbb{R}_{\geq 0}$; $f(a) \rightarrow \max$ in detail, as LLaMEA is a generic concept. In Section III-C, we define the specific quality measure used here for generating metaheuristics.

A. Starting Prompt

The initialization of the optimization loop is crucial for the framework to work, as it sets the boundaries and rules that the LLM needs to operate with. Through experimentation, we found that including a small example code into the task prompt S helps a lot in generating code without syntax and runtime errors. However, the example code could also bias the search toward similar algorithms. In this work we therefore choose to use a simple implementation of random search (RS) as the example code to provide. The code of this RS algorithm is provided in our Zenodo repository [39]. The LLM receives an initial prompt with a specific set of criteria, domain expertise, and problem description. This starting point guides the LLM in generating an appropriate algorithm. The starting prompt provides the role description first, followed by a detailed description of the problem to solve including a clear format

of the expected response. Our detailed task prompt S is given below:

Detailed task prompt S

```
Your task is to design novel metaheuristic
    algorithms to solve closed box
    optimization problems.
The optimization algorithm should handle a
    wide range of tasks,
which is evaluated on a large test suite of
    noiseless functions.
Your task is to write the optimization
    algorithm in Python code.
The code should contain one function `def
    __call__(self, f)`,
which should optimize the black box function
    `f` using
`budget` function evaluations.
The f() can only be called as many times as
    the budget allows.
An example of such code is as follows:
```
<initial example code>
```
Give a novel heuristic algorithm to solve
this task.
Give the response in the format:
# Name: <name of the algorithm>
# Code: <code>
```

B. Algorithm Synthesis (Initialization)

Using the prompt S, the LLM generates the code for a new metaheuristic optimization algorithm, considering the constraints and guidance provided. The LLM should provide the answer in the format given, using MarkDown formatting for the code-block. The generated algorithm and its name are extracted using regular expressions, including additional exception handling to capture small deviations from the requested format. In our experiments, 100% of the LLM responses lead to successfully extracted code. In addition, the LLM usually generates a small explanation in addition to what we ask, which we store for offline evaluation. In the case that we cannot extract the code (which in practice never happened but can theoretically occur), we provide the LLM feedback that the response did not follow the provided format, trying to enforce the format for the next iteration.

Exception Handling: Once the algorithm is extracted we dynamically load the generated Python code and instantiate a version of the algorithm for evaluation. The loading and instantiating of the code can lead to syntax errors, which we capture and store to be provided in the refinement prompt F. When these errors occur, the evaluation run cannot commence, and is therefore skipped. Evaluation metrics are set to the lowest possible value (0) for the particular candidate and error messages e_t are stored for additional feedback to the LLM.

C. Evaluation

The generated algorithm is evaluated on the BBOB suite (see [31] and the supplemental material for an overview of the functions). The suite consists of 24 noiseless functions, with an instance generation mechanism to generate a diverse set of different optimization landscapes that share particular function characteristics. The suite is divided into five function groups: 1) separable functions; 2) functions with low or moderate conditioning; 3) high conditioning unimodal functions; 4) multimodal functions with strong global structure; and 5) multimodal functions with weak global structure. The BBOB suite is considered a best-practice in the field of metaheuristic algorithm design for the evaluation of newly proposed algorithms. For the evaluation feedback provided to the LLM, we summarize the performance on the whole BBOB suite by taking an average any-time performance metric, namely the normalized AOCC [see (2)]. This results in one number y_t representing aggregated performance of the proposed algorithm over the complete benchmark suite, including multiple instances per function, and its standard deviation σ_t over multiple runs of the algorithm.

Each generated algorithm is run for a fixed budget B of function evaluations, and the IOHexperimenter suite makes sure that the algorithm is terminated after using the full budget of evaluations. In practice, this mechanism ensures that generated algorithms do not exceed the budget and always terminate. In addition to aggregating these values over all functions, we also experiment with a more detailed feedback mechanism where the average *AOCC* and its standard deviation are returned for each function group, resulting in ten performance values $(y_{t,1}, \ldots, y_{t,5})$ and $(\sigma_{t,1}, \ldots, \sigma_{t,5})$ (two per function group). Theoretically, this approach should provide the LLM with information on what kind of functions the proposed algorithm works well and on which it works less well.

D. Mutation, Selection, and Feedback

The mutation and selection step in the LLaMEA framework consists mostly of the construction of the feedback prompt to the LLM in order to generate a new solution. Depending on the selection strategy, either the current-best algorithm a_b is given back ($(1+1)$-strategy) to the LLM, including the score it had on the BBOB suite, or the final generated algorithm a_t ($(1, 1)$-strategy) is provided to the LLM in the feedback prompt F.

After the selection is made, a feedback prompt F (lines 9 and 11 of Algorithm 1) is constructed using the following template:

Feedback prompt template F

```
<Task prompt S>
<List of previously generated algorithm
    names with mean $AOCC$ score>
<selected algorithm to refine (full
    code) and mean and std $AOCC$
    scores>
Either refine or redesign to improve
    the algorithm.
```

The prompt includes the initial detailed task prompt S, a list $((\text{name}(a_0), y_0), \ldots, (\text{name}(a_t), y_t))$ of previously generated algorithm names and their mean scores, the selected algorithm a_b or a_t, including its score y_b (y_t) and standard deviation σ_b (σ_t), to mutate and a short task prompt F_0 = "Either refine or redesign to improve the algorithm" telling the LLM to perform

a mutation or redesign (restart) action. The list of previously tried algorithm names is included to make sure the LLM is not generating (almost) the same algorithm twice.

The construction of the feedback prompt includes a few choices, which in general can be seen in an EC context as follows.

Restarts and Mutation Rate: We ask the LLM to either make a (small) refinement of an algorithm or redesign it completely, where the latter is analogue to a restart or very large mutation rate in an evolutionary algorithm optimization run, increasing its exploration behavior. In the proposed framework we leave this choice to the LLM itself as our task is simply *Either refine* ***or*** *redesign to improve the algorithm*. In Section V-B, we analyze how the LLM makes this decision over time, in terms of the generated algorithm names as well as code similarity.

Plus and Comma Strategy: In the proposed framework we support both $(1+1)$- and $(1, 1)$-LLaMEA strategies, meaning the LLM either is asked to refine/redesign the best-so-far algorithm (in the elitist $(1+1)$-case), or the final generated algorithm (in the $(1, 1)$-case). Only the full code of the selected (best or final) algorithm is provided to the LLM in every iteration. In our experiments we demonstrate the differences between both strategies per LLM model.

Detailed Feedback: Instead of providing an overall score and standard deviation, we can provide the LLM with more performance details of the generated algorithms. In our case, we can provide additional metrics per BBOB function group, potentially giving the LLM information on what kind of functions the solution works well and on which ones it does not work well. This results in mean *AOCC* values and standard deviations for each of the five function groups, i.e., $(y_{t,1}, \ldots, y_{t,5})$ and $(\sigma_{t,1}, \ldots, \sigma_{t,5})$.

In our experiments we cover the inclusion of such details versus leaving them out. However, since adding such details never showed an increase in performance, we have left these results in the supplemental material to keep our main results concise and clear.

Session History: In the proposed framework we do not keep the entire run history in every iteration [including all codes $(a_0, \ldots, a_t)$], as this becomes more and more expensive as the list grows. However, the inclusion of a larger set of previous attempts by providing the previous solutions in code with their associated scores could in theory be beneficial. The LLM would be able to do multishot in-context learning, and potentially learn from previously generated solutions. This would translate to EC terms, as to keeping an archive of best solutions or the use of predictive machine learning models in machine learning assisted optimization [40], [41].

Instead of providing all codes, we keep a condensed list of algorithm names (generated by the LLM) and their respective scores, $((\text{name}(a_0), y_0), \ldots, (\text{name}(a_t), y_t))$, to facilitate in-context learning of the LLM [32], [33]. The purpose of keeping this list and providing it to the LLM is twofold, namely 1) the LLM could learn what kind of algorithms work well and which work less good (by analyzing the algorithm names only) and 2) it makes it less likely that the LLM is generating the same algorithm twice.

IV. Experimental Setup

To validate the proposed evolution framework for generating and optimizing metaheuristics, we designed a set of experiments to compare three different LLMs and the two different selection strategies for each of them. We then compare the best of these six combinations of LLM and selection strategy with the state of the art EoH algorithm (using the best-performing LLM) and an RS baseline (also using the best-performing LLM). In addition, the best generated algorithms are analyzed and compared to several state-of-the-art baselines on the BBOB suite using additional instances and additional dimension settings.

A. Large Language Models

In our experiments we have limited the number of LLMs to the ChatGPT family of models, including gpt-3.5-turbo-0125 [42], gpt-4-turbo-2024-04-09 [43], and the recently released gpt-4o-2024-05-13 [44]. The LLaMEA framework leverages the OpenAI chat completion API call for querying the LLM. Each model was run with default parameters (top_p equal to 1) and a temperature of 0.8. For abbreviating LLM-names, we drop the extension "turbo" in the following.

B. Benchmark Problems

As explained before, we use the BBOB benchmark function suite [31] within IOHexperimenter. For a robust evaluation we use three different instances per function, where an instance of a BBOB function is defined by a series of random transformations that do not alter the global function characteristics. In addition, we perform three independent runs per function instance with different random seeds (giving a total of nine runs per BBOB function). Each run has an evaluation budget of $B = 10\,000$ function evaluations. In our experiments we set the dimensionality of the optimization problems to $d = 5$. We run the main $(1\overset{+}{,}1)$-LLaMEA optimization loop in Algorithm 1 for $T = 100$ iterations.

C. Performance Metrics

To evaluate the generated algorithms effectively over a complete set of benchmark functions we use a so-called *anytime performance measure*, meaning that it quantifies performance of the optimization algorithm over the complete budget, instead of only looking at the final objective function value. For this we use the normalized AOCC, as introduced in [45]. The AOCC is given in

$$AOCC(\mathbf{y}_{a,f}) = \frac{1}{B}\sum_{i=1}^{B}\left(1 - \frac{\min(\max((y_i), lb), ub) - lb}{ub - lb}\right). \quad (2)$$

Here, $\mathbf{y}_{a,f}$ is the sequence of best-so-far log-scaled precision values (i.e., differences $\log(y_i - f^*)$ between actual function value y_i and function value f^* of the global minimum of the function [46]) reached during the optimization run of algorithm a on test function f and y_i its ith component, $B = 10\,000$ is the budget of function evaluations per run, lb and ub are the lower and upper bound of the function value range of interest. Here, we use the common setting of

$lb = 10^{-8}$ and $ub = 10^2$. Following best-practice [46], the precision values are log-scaled before calculating the *AOCC*. The AOCC is equivalent to the area under the so-called empirical cumulative distribution function (ECDF) curve with infinite targets between the chosen bounds [24].

Following common practice, we aggregate the *AOCC* scores of all 24 BBOB benchmark functions $f_1, \ldots, f_{24}$ by taking the mean over functions and their instances, i.e., for an algorithm a

$$AOCC(a) = \frac{1}{3 \cdot 24} \sum_{i=1}^{24} \sum_{j=1}^{3} AOCC\left(\mathbf{y}_{a,f_{ij}}\right) \tag{3}$$

where f_{ij} is the jth instance of function i. The final mean *AOCC* over $k = 5$ independent runs of algorithm a over all BBOB functions is given as feedback to the LLM in the next step, and is used as best-so-far solution in case an improvement was found. In other words, the quality measure $f(a)$ used in Algorithm 1 is defined as

$$f(a) = 1/k \sum_{i=1}^{k} AOCC(a). \tag{4}$$

In addition to this mean *AOCC* score, any runtime or compile errors that occurred during validation are also used to give feedback to the LLM. In case of fatal errors (no execution took place), the mean *AOCC* is set to the lowest possible score, zero.

D. Baselines

To assess the performance of the proposed framework, we compare it with two baselines, namely the state-of-the-art approach EoH and RS, also evaluated on the BBOB benchmark function suite using the same performance measure based on *AOCC*, given in (4). For EoH to work on our use-case, we modified the approach slightly, making as little changes as possible. We keep the same prompt as we used for the proposed approach, with the exception that it should generate a "small description" instead of a name and that it should generate "a single function with a specific name" (both required by EoH to work). We then linked the evaluation function to IOHexperimenter in the exact same way as for the proposed approach. All EoH hyper-parameters are kept at their default values. Since EoH is minimizing by default we multiply the evaluated fitness by -1 before returning it to EoH. The RS baseline is just prompting the LLM for 100 times using the same starting prompt. Both baselines use the GPT-4 LLM as this was the best-performing choice.

V. Results and Discussion

In Fig. 2, the mean best-so-far algorithm evaluation score [*AOCC* according to (4)] per configuration (LLM and strategy) is shown. Since we show only best-so-far values, all curves are monotonically increasing, also when (1, 1)-selection is used. When execution errors occur, which happens in 18.7% (843 out of 4500) of all runs, the corresponding *AOCC* value of zero is not plotted, as this would make the graphs unreadable.

The shaded region denotes the standard deviation over the five independent runs of Algorithm 1 that were performed for each of the nine combinations of LLMs and selection operator. Note that, of these nine combinations, only six are shown in Fig. 2, while the other three (with details) are provided in the supplemental material. This means that the commercial LLM-interfaces were called for a total of $5 \cdot 9 \cdot 100 = 4500$ times. Due to this costly (concerning computational effort and funds required for using the commercial LLMs) experimentation, we have limited the number of repetitions of the runs to 5, providing a good enough measure of the average performance and its variation. Since we evaluate the generated algorithms across a whole set of 24 continuous optimization problems in the BBOB test function set, and on the first three instances with three runs each, we perform $24 \cdot 9 = 216$ runs at $B = 10\,000$ function evaluations each, for evaluating a single metaheuristic generated by the LLM. This results in 2.16 million function evaluations, and the 4500 metaheuristics generated required a total of $9.72 \cdot 10^9$ function evaluations.

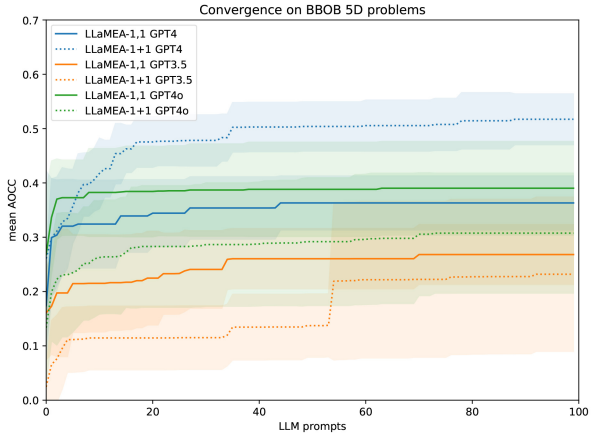

Fig. 2. Mean convergence curves (best-so-far algorithm scores) over the five different runs for each LLM and selection strategy. Shaded areas denote the standard deviation of the best-so-far. Note that the difference in initial performance already results from the fact that, although the starting prompt S is identical for all LLMs, the performance value shown here is the mean *AOCC* value of the first algorithm a_1 generated by each LLM in line 4 of Algorithm 1. Note that only best-so-far values are plotted, also for (1,1)-strategies, and infeasible algorithm results are also not plotted (as they would have an $AOCC = 0$ value as mentioned in Algorithm 1).

From Fig. 2 we conclude that different strategies yield a range of different results, depending on the model used. For example, the $(1 + 1)$-selection strategy seems to be beneficial for GPT-4, but is having a deteriorating effect when using GPT-4o. Overall GPT-4 seems to be better suited for the task overall, and GPT-3.5 is clearly a less favorable option. In Fig. 3, we compare our approach with the two baseline approaches, namely the EoH algorithm and RS (each using GPT-4, too). To use EoH, we extended it to work in tandem with IOHexperimenter in the same way as LLaMEA. The LLaMEA-$(1+1)$ GPT-4 shows better convergence (area under the *AOCC* curve) than the EoH approach and also results in an overall better algorithm in the end. Both EoH and the proposed approach show a clear improvement over just randomly sampling the LLM, indicating the advantages of using an evolutionary search framework. The EoH approach follows as the second best approach, where EoH is also run five times with a population size of 5 and default hyperparameters. EoH

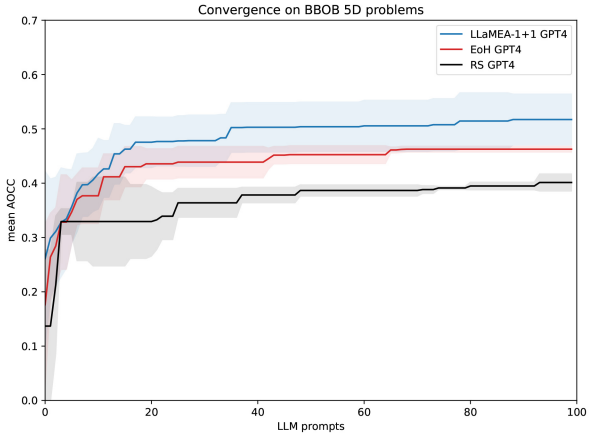

Fig. 3. Mean convergence curves (best-so-far algorithm scores) over the five different runs for the best strategy LLaMEA-variant, i.e., LLaMEA-(1+1) GPT-4 (same curve as in Fig. 2), including the state-of-the-art baseline EoH algorithm (red) and the RS baseline (black). Shaded areas denote the standard deviation of the best-so-far over five runs. Additional remarks as provided in the caption of Fig. 2 apply here, too.

performs a bit more stable (in terms of variation over different runs), due to the larger population size. However, it suffers from tries where the generated code is not executable since it has no self-debugging capabilities (unlike LLaMEA). Another limitation of EoH is that it can only deal with single functions, while in our proposed approach complete Python classes are generated, allowing for more complex interactions.

A. Novelty and diversity

Analyzing the generated codes and their generates names, it is immediately obvious (and not surprising) that the LLM uses existing algorithms, algorithm components, and search strategies in generating the proposed solutions.

In Fig. 4, a word-cloud is shown with all the substrings of generated algorithm names and their occurrence frequency visualized. Note that, the word-cloud serves purely as an intuitive visual representation to quickly get an overview of the different words the algorithm names contain. All algorithm names and codes are available in our open-source repository [39]. The generated algorithm names are in Python Camel-case style, such that it is easy to split the algorithm names into individual parts. In some cases, the algorithm name is just an abbreviation, and these abbreviations are kept as words in this analysis. The most used parts in algorithm names are rather generic, such as "evolution," "adaptive," and "dynamic." Some of the parts directly refer to existing algorithms, such as "harmony" and "firework," and other strings refer to existing strategies, such as "gradient," "local," "elite," etc. In general when observing the different generated solutions, we see interesting and novel combinations of existing techniques, such as *"surrogate assisted DE combined with covariance matrix adaptation evolution strategies"* and *"dynamic firework optimizer with enhanced local search"*. A full list of generated algorithms and their names is provided in our Zenodo repository [39]. The algorithm names are generated automatically by the LLM that is used by LLaMEA. We have performed manual checks and can confirm that the

Fig. 4. Word cloud of algorithm name parts generated over all different LLaMEA runs.

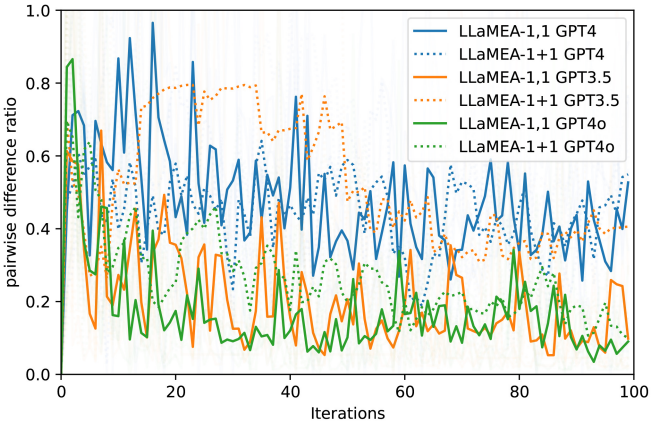

Fig. 5. Pairwise differences between parent and offspring for each iteration. Solid lines represent the mean over all runs per model and strategy, more transparent lines are individual runs. *w/ Details* denotes that we use a feedback mechanism that provided not just the plain average *AOCC* but also the average *AOCC* per BBOB function group as feedback to the LLM.

algorithm names are almost always in line with the actual code. This means that the LLM indeed creates a descriptive name for the generated algorithm, which is based on the key algorithmic components used in the algorithm.

B. Mutation Rates

To analyze how much the LLMs change (mutate) the algorithms over an entire optimization run, the code *diff* ratio (number of code lines that are different between a pair of programs divided by the length of the largest code) is calculated over each run between parent and offspring solutions. In Fig. 5, the mean *diff* over five different runs per LLM is shown.

There are a few interesting observations we can make as follows. The difference between parent and offspring for GPT-3.5 is on average smaller than for other models, and the $(1+1)$-GPT3.5-LLaMEA shows much higher differences in the beginning of the run than the other strategy with the same model. GPT-4-LLaMEA in general shows the largest differences (exploration) over the runs. It is very interesting to observe that the ratio of code differences in most cases seems to converge, indicating more exploration in the beginning of the search and more exploitation during the final parts of the search. This is interesting as the LLM has no information on the search budget T of Algorithm 1, and can therefore also not base its decision to make large or small refinements on the stage of the optimization run. When we look at specific code-diffs between parents and offspring generated by the $(1+1)$-GPT-4-LLaMEA, we observe that the LLM mutates both hyper-parameter values and higher level logic, like introducing a new crossover or mutation operator. In addition it mutates the comments in the code to reflect and argue about the changes. See the supplemental material for some specific examples.

We can furthermore see in the generated names of the algorithms that the LLM has either tried to refine the algorithm (adding "Improved" or "Refined" or "Enhanced" to the name, or generating names, such as `<algorithm>`V1, etc.) or redesign the algorithm using a different strategy (based on different existing algorithm names, such as DE, particle swarm optimization, etc.). We can therefore also look at the similarity between parent and offspring algorithm names to visualize the refinement process of the optimization runs.

To do so, we use the *Jaro similarity* [47] as it gives a ratio between 0 (completely different names) and 1 (completely matching). The Jaro similarity score between two algorithm names s and t is calculated as

$$\text{Jaro}(s,t) = \begin{cases} 0, & \text{if } m = 0 \\ \frac{1}{3}\left(\frac{m}{|s|} + \frac{m}{|t|} + \frac{m-t}{m}\right), & \text{otherwise} \end{cases} \tag{5}$$

where

1) s and t are the input strings.
2) m is the number of matching characters. Two characters from s and t are considered matching if they are the same and not farther apart than $\max(|s|, |t|)/2 - 1$.
3) t is half the number of transpositions. A transposition occurs when two matching characters are in a different order in s and t.
4) $|s|$ and $|t|$ are the lengths of strings s and t, respectively.

Fig. 6 illustrates the Jaro similarity mean (five runs) of the subsequently generated algorithm names (i.e., $\text{Jaro}(\text{name}(a'), \text{name}(a_{t+1}))$ for $t = 0, \ldots, T-1$, where $a' = a_b$ for $(1+1)$-selection and $a' = a_t$ for $(1,1)$-selection) over iterations t for each of the LLM-based optimization run configurations.

The Jaro similarity scores consistently show a behavior similar to the code difference ratios. It is clear that some steps only involve very small mutations (especially for GPT3.5) and sometimes a kind of restart occurs where the similarity score reaches zero in a single run.

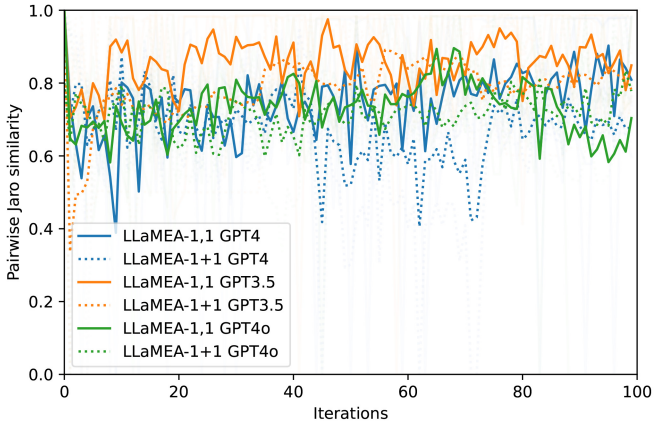

Fig. 6. Average Jaro similarity scores between parent and offspring over each optimization run for different models.

VI. Analysis of Best Algorithms

The experiment above resulted in 3657 algorithms (out of a maximum[2] of 4500) that were at least able to get an *AOCC* score larger than zero on the BBOB suite of optimization problems in $d = 5$ dimensions. In the next step to validate our proposed framework, we evaluate how much the best of these algorithms can generalize beyond the evaluation performed during the optimization loop, and subsequently we analyze in-depth the code and behavior of the best-performing algorithm that beats the state-the-art baselines. The evaluation is done in 5, 10, and 20 dimensions, using five different instances and five independent runs per instance (so 25 runs per BBOB function to optimize). For a fair comparison, we decided to rerun the generated algorithms for $d = 5$ in this setting (with 25 runs), too. The state-of-the-art baselines are the CMA-ES [48], DE [49], and an extensively optimized version of modular CMA-ES [45], [50] denoted as *CMA-best*. CMA-best is a CMA-ES algorithm with active update, a Gaussian base sampler, $\lambda = 20, \mu = 10$, IPOP, the mirrored sampling strategy, and CSA step-size-adaptation (see [50] for the details of these configurations). The optimized CMA-ES algorithm is the best-configured algorithm (in terms of *AOCC*) on BBOB in five dimensions and for a budget of 10 000 function evaluations, out of more than 52 128 CMA-ES and DE algorithm variants. Beating this optimized baseline is incredibly hard as it was the result of a very large modular optimization experiment. We use the CMA-best baseline to obtain a reasonable upper-bound of what is possible when optimizing an algorithm configuration to a specific benchmark set and fixed budget and dimensionality.

In Fig. 7 the empirical attainment functions (EAFs) [24] are shown for the best algorithm generated per LLM and strategy, resulting in six automatically generated algorithms plus the three baselines (CMA-ES, CMA-ES-best, and DE). The EAF estimates the fraction of runs that attain an arbitrary target value not later than a given runtime (i.e., number of function evaluations). Taking the partial integral of the EAF results in a more accurate version of the ECDF, since it does not rely

[2]Five runs with $T = 100$ iterations, each, for nine different models and strategies.

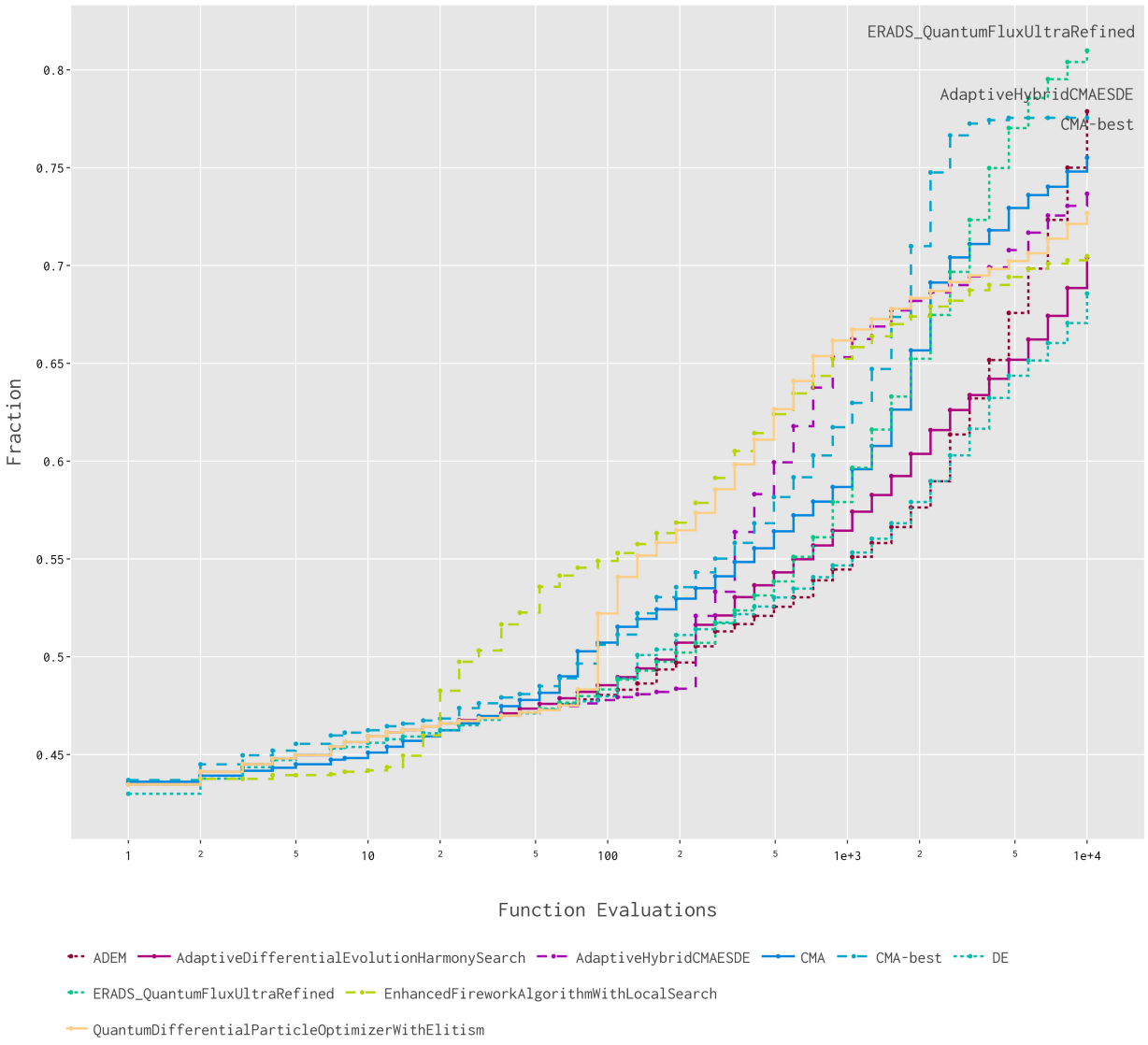

Fig. 7. EAFs estimate the fraction of runs that attain an arbitrary target value not later than a given runtime. The fraction (the y-axis) denotes the cumulative fraction of target values that have been reached by an optimization algorithm, as a function of the number of function evaluations (the x-axis). We show the EAF for the best algorithms per LLM configuration (i.e., six algorithms) and the three baseline algorithms (CMA-ES, best CMA-ES, and DE), averaged over all 24 BBOB functions in $5d$.

on discretization of the targets. The "fraction" (the y-axis in Fig. 7) denotes the cumulative fraction of target values that have been reached by an optimization algorithm, as a function of the number of function evaluations (the x-axis in Fig. 7).

Such EAF curves summarize how well an optimization algorithm performs over the complete run for the given set of target values, again on average over all instances, independent runs and objective functions. From the EAF curves and the area under these curves (see Table I), we can observe that LLaMEA has found one algorithm (*ERADS_QuantumFluxUltraRefined*, for short ERADS) with better performance for $d = 5$ than the state-of-the-art CMA-ES in terms of *AOCC*, two algorithms (ERADS and *AdaptiveDifferentialEvolutionHarmonySearch*) that on average perform better than the optimized CMA-best after the total budget in $5d$ (the EAF curve reaches higher), two other algorithms that perform better than CMA-best in the first 1000 evaluations (*EnhancedFireworkAlgorithmWithLocalSearch* and *QuantumDifferentialParticleOptizerWithElitism*), and seven algorithms that perform better than the DE baseline but worse than the CMA baseline. Detailed convergence curves per BBOB function (for $d = 5$) for ERADS_QuantumFluxUltraRefined and three baselines are shown in Fig. 8, these curves are also available for all best algorithms in the supplemental material. From this Figure we can observe that especially for BBOB f_{17} and f_{18}, the ERADS algorithm shows very promising search behavior.

A. "ERADS_QuantumFluxUltraRefined" Algorithm

While the name sounds rather futuristic and very sophisticated, upon a close inspection of the code the *ERADS_QuantumFluxUltraRefined* algorithm looks very similar a standard DE algorithm. According to the LLM, ERADS stands for *"Enhanced RADEDM with Strategic Mutation"*, and RADEDM stands for *"Refined*

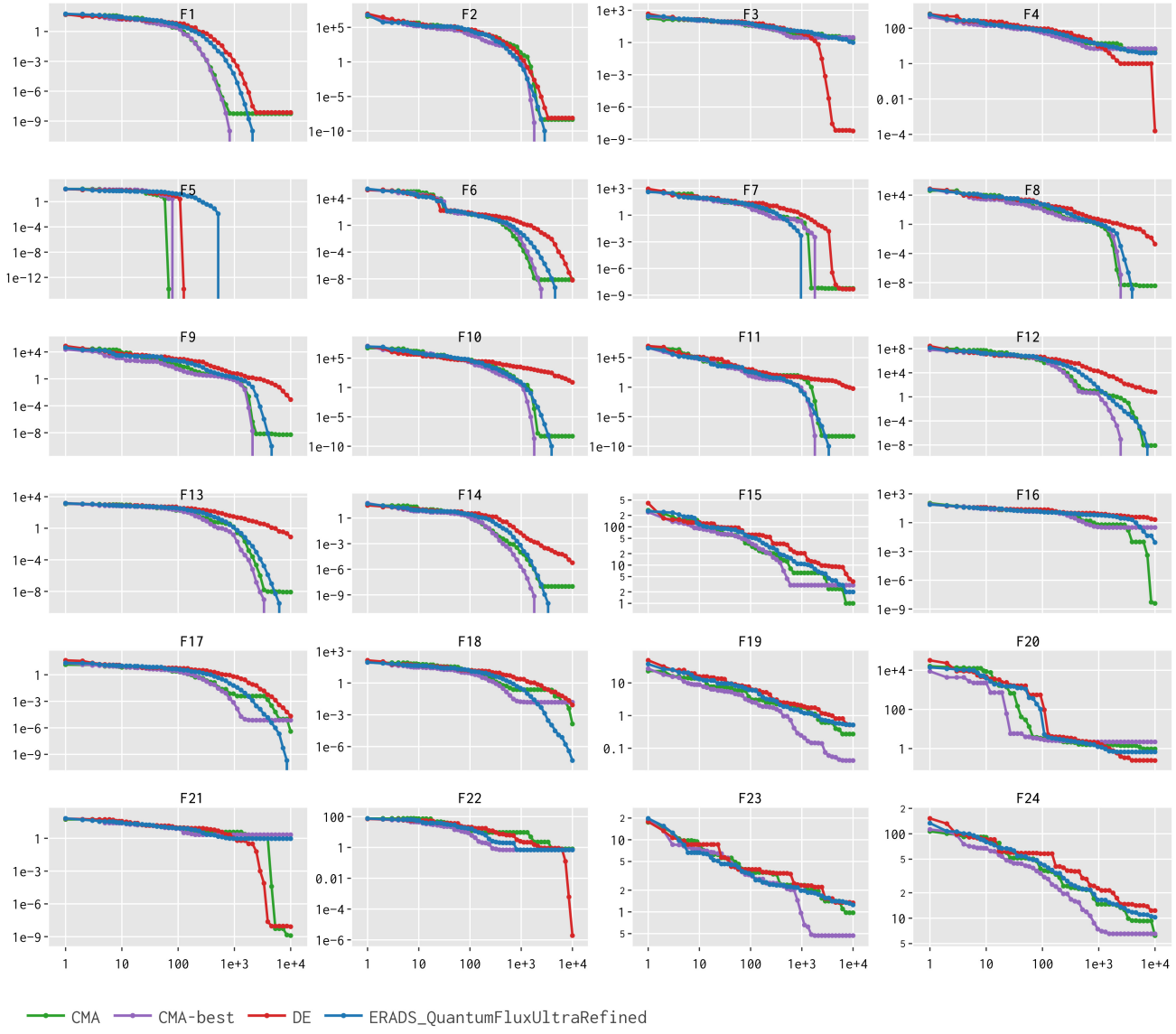

Fig. 8. Median best-so-far function value (the y-axis) over the 10 000 function evaluations (the x-axis) per BBOB function in $5d$ for the ERADS_QuantumFluxUltraRefined algorithm and the baselines: CMA-ES, DE, and CMA-best, for each of the 24 BBOB test functions.

Adaptive DE with Dynamic Memory". The pseudocode of *ERADS_QuantumFluxUltraRefined*, representing the key ideas of the generated Python code, was extracted manually and is shown in Algorithm 2. The full Python code can be retrieved from our online repository [39].

The main differences between ERADS and DE are the use of a memory factor to guide the mutation in certain directions and an adaptive F and CR control strategy. It is unclear, why the LLM generated the "Quantum Flux" component in the algorithm's name, but it can be assumed that LLM hallucinations [51] will also occur when applied to code generation tasks.

The proposed ERADS algorithm seems most similar to the existing JADE [52] algorithm from the literature. There are, however, some key differences which are, to the best of our knowledge, novel and have not been published before.

The key differences primarily revolve around the use of a memory vector and a memory factor (lines 4, 10, 18, and 31 of Algorithm 2) in combination with parameter adaptation (line 13 of Algorithm 2). Moreover, ERADS involves three randomly selected individuals from the population, the best individual, and the memory vector in the mutation operator (line 18) and then, with probability $CR = 0.95$, performs a crossover of the mutant and the actual individual (line 20). This crossover operator is the classical binomial crossover from DE, which copies a variable $\mathbf{v}_{i,j}$ from the mutant vector $\mathbf{v}_i$ with probability $CR = 0.95$, i.e., the *trial* vector in line 18 is almost always identical to the *mutant* vector. In classical DE notation, the mutant $\mathbf{v}_i$ (i being the index of individuals in the population) is generated in Algorithm 2 as follows:

$$\mathbf{v}_i = \mathbf{x}_{r_1} + F_g \cdot \left(\mathbf{x}_{best} - \mathbf{x}_{r_1}\right) + F_g \cdot \left(\mathbf{x}_{r_2} - \mathbf{x}_{r_3}\right) + F_g \cdot m_f \cdot \mathbf{m}.$$

As usual, the r_i indicate randomly sampled solution indices from the current population and F_g is the mutation factor (F_{current} in Algorithm 2), which in ERADS changes per generation g. The first two terms and crossover can best be

TABLE I
AREA UNDER THE EAF CURVE SCORES (AUC) FOR THE BEST THREE ALGORITHMS PER MODEL IN 5*d*, HIGHER AUC SCORES STANDS FOR A BETTER ANYTIME PERFORMANCE AND HAVE A SIMILAR MEANING AS THE *AOCC* METRIC USED EARLIER

ID	AUC
CMA-best (optimized baseline)	0.742
ERADS_QuantumFluxUltraRefined	0.733
CMA (baseline)	0.703
AdaptiveHybridCMAESDE	0.697
QuantumDifferentialParticleOptimizerWithElitism	0.695
EnhancedFireworkAlgorithmWithLocalSearch	0.684
AdaptiveHybridDEPSOWithDynamicRestart	0.684
ADEM	0.667
AdaptiveDifferentialEvolutionHarmonySearch	0.643
EnhancedDynamicPrecisionBalancedEvolution	0.641
DE (baseline)	0.628
QPSO	0.604

characterized as a DE/rand-to-best/1/bin strategy in classical DE notation [53], but the additional memory vector that contributes $0.3 \cdot F_g \cdot \mathbf{m}$ (remember that $m_f = 0.3$, as initialized in line 4, remains constant) is new. It should also be noted that ERADS updates $\mathbf{x}_{\text{best}}$ immediately within the current generation, if the newly generated individual improves $\mathbf{x}_{\text{best}}$. Similarly, $\mathbf{m}$ is updated immediately, if the newly generated individual improves $\mathbf{x}_{\text{best}}$, as follows:

$$\mathbf{m} \leftarrow 0.7 \cdot \mathbf{m} + 0.3 \cdot F_g \cdot (\mathbf{v}_i - \mathbf{x}_i).$$

It is interesting to observe that this update does not use the *trial* vector that was generated by crossover of $\mathbf{v}_i$ and $\mathbf{x}_i$ in line 20, and which is used to update $\mathbf{x}_{\text{best}}$ upon an improvement. Instead, it just uses the mutant vector $\mathbf{v}_i$, which, however, is almost identical to the *trial* vector due to $CR = 0.95$, which is the probability for each component of the mutant vector to be copied to the trial vector.

Finally, ERADS also includes a generational adaptation that linearly increases F_g from an initial value of 0.55 to the final value of 0.85, i.e., $F_g = 0.55 + 0.3 \cdot t/B$ (line 13). While this increases the mutation factor over time, it still remains below the maximum range of the mutation factor that is typically recommended, i.e., [0, 1] [52].

To further examine the hyper-parameters of the ERADS algorithm we include additional hyper-parameter optimization experiments in the supplemental material. The result clarifies that the LLM has already generated well performing hyper-parameter settings for $d = 5$ that can not be improved further.

In contrast to ERADS, JADE uses an optional archive with the best solutions which primarily serves to maintain diversity. Furthermore, JADE also features adaptive control parameters (F and CR) that adjust dynamically based on the success rates of the previous generations, aiming to optimize these parameters in real time to improve performance.

It is interesting to observe that most of the best algorithms proposed by the LLM are similar to DE (or hybrids of CMA and DE), which could indicate a bias toward this kind of algorithms in the LLM, or it could indicate that overall this kind of algorithms work well for the specific benchmark suite.

Algorithm 2 ERADS_QuantumFluxUltraRefined

1: $N \leftarrow 50$ ▷ Population size
2: $F_{init} \leftarrow 0.55, F_{final} \leftarrow 0.85$ ▷ Initial and final mutation scaling factors
3: $CR \leftarrow 0.95$ ▷ Crossover probability
4: $Memory\_factor \leftarrow 0.3$ ▷ Factor to integrate memory in mutation
5: $P \leftarrow$ u.a.r. population initialization within $(-5.0, 5.0)$
6: $\text{fit}_P \leftarrow f(P)$
7: $best\_index \leftarrow \text{argmin}(\text{fit}_P)$
8: $f_{opt} \leftarrow \text{fit}_P[best\_index]$
9: $\mathbf{x}_{opt} \leftarrow P[best\_index]$
10: $Memory \leftarrow \mathbf{0}$ ▷ Initialize memory for mutation direction
11: $t \leftarrow N$
12: **while** $t < B$ **do**
13: $F_{current} \leftarrow F_{init} + (F_{final} - F_{init}) \cdot \left(\frac{t}{B}\right)$
14: **for** each i in $[0, N-1]$ **do**
15: $indices \leftarrow$ select 3 distinct indices u.a.r. from $[0, N-1] \setminus \{i\}$
16: $\mathbf{x1}, \mathbf{x2}, \mathbf{x3} \leftarrow P[indices]$
17: $best \leftarrow P[best\_index]$
18: $mutant \leftarrow \mathbf{x1} + F_{current} \cdot (best - \mathbf{x1} + \mathbf{x2} - \mathbf{x3} + Memory\_factor \cdot Memory)$
19: $mutant \leftarrow$ clip $mutant$ to bounds $(-5.0, 5.0)$
20: $trial \leftarrow$ crossover $(mutant, P[i])$ w. prob. CR
21: $f\_trial \leftarrow f(trial)$
22: $t \leftarrow t + 1$
23: **if** $f\_trial < \text{fit}_P[i]$ **then**
24: $P[i] \leftarrow trial$
25: $\text{fit}_P[i] \leftarrow f\_trial$
26: **if** $f\_trial < f_{opt}$ **then**
27: $f_{opt} \leftarrow f\_trial$
28: $\mathbf{x}_{opt} \leftarrow trial$
29: $best\_index \leftarrow i$
30: **end if**
31: $Memory \leftarrow (1 - Memory\_factor) \cdot Memory + Memory\_factor \cdot F_{current} \cdot (mutant - P[i])$
32: **end if**
33: **if** $t \geq B$ **then**
34: **break**
35: **end if**
36: **end for**
37: **end while**
38: **return** $\mathbf{x}_{opt}, f_{opt}$ ▷ Return best solution and quality

B. Performance Analysis in Higher Dimensions

The best-found algorithms using the LLaMEA framework are evaluated against the most relevant state-of-the-art optimizers, namely the previously used baselines CMA-best, CMA-ES, and DE. CMA-ES and DE are using recommended hyperparameter settings. The baselines all originate from the IOHanalyzer benchmark data set [22]. In Fig. 9, the EAF of the proposed algorithms and baselines are shown for dimension $d \in \{10, 20\}$ using results from all BBOB functions, five instances per function and five random seeds (25 runs per BBOB function in total). It is interesting

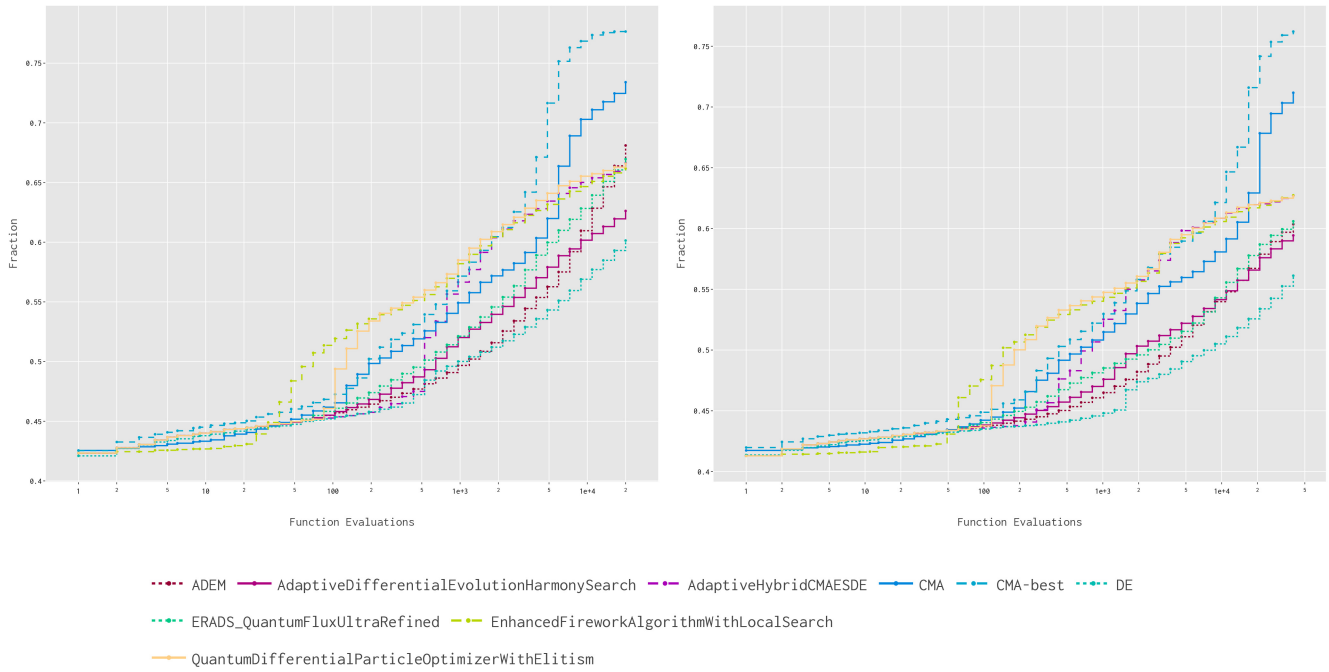

Fig. 9. EAF estimates the percentage of runs that attain an arbitrary target value not later than a given runtime. EAF for the best algorithm per configuration and the baselines; CMA-ES, CMA-best, and DE, averaged over all 24 BBOB functions in $10d$ (left plot) and $20d$ (right plot), respectively.

to observe that while ERADS performs well in five D problems, it fails to generalize well to higher dimensions, while *EnhancedFireworkAlgorithmWithLocalSearch* and *QuantumDifferentialParticleOptizerWithElitism* both have very good performance in higher dimensions for the first 2000 function evaluations (even better than CMA-best), but around 10 000 function evaluations CMA-ES outperforms the other algorithms clearly. It is also interesting to observe that the optimized (for $d = 5$) CMA-best performs less well than a CMA-ES with default hyper-parameters in $10d$ and $20d$. Just like the LLaMEA proposed algorithms, CMA-ES best was optimized on $5d$ and with a budget of 10 000 evaluations. When looking at convergence curves per function, we note that CMA-ES outperforms mainly on the very hard functions in the benchmark (f_{23} and f_{24}). Detailed convergence curves per BBOB function in $5d$, $10d$, and $20d$ are available in the supplemental material [39].

We would like to emphasize that the goal of our research is to show that, for a *specific* setting (here: BBOB in $5d$), an LLM can generate a superior algorithm automatically. This algorithm is not expected to scale to high-dimensional problems, as it was generated for the specific case of $5d$ problems. For this reason, we do not go beyond $d = 20$ for further evaluation of the generated algorithms.

VII. Conclusion and Outlook

This article introduced LLaMEA, a novel framework leveraging LLMs for the automatic generation and optimization of metaheuristic algorithms. Our approach automates the evolution of algorithm design, enabling efficient exploration and optimization within a computationally feasible framework. Our findings demonstrate that LLaMEA can effectively generate high-performing algorithms that rival and sometimes surpass existing state-of-the-art techniques.

The LLaMEA framework proved capable of generating and evolving algorithms that perform comparably to traditional state-of-the-art metaheuristics, demonstrating the potential of LLMs to innovate within the algorithmic design space effectively. Algorithms evolved through our framework successfully compete against established metaheuristics, highlighting the practical applicability of using LLMs for automated construction of optimization heuristics. Since LLMs have been trained on an exceedingly large code base, including the currently available metaheuristics, we can explain their observed strong performance by interpreting them as a *universal modular framework for algorithm construction* (see also Section I) that have access to a huge number of "modules" that can be combined to form new algorithms. In addition, we observed that the LLM is able to both fine-tune algorithm parameters, as well as introduce new logic, such as different mutation and crossover strategies in the generated algorithms.

Considering how much work goes into the manual design of "nature-inspired heuristics," which are often not novel [54], [55], [56], [57] and often cannot compete with RS [10], we argue that an automated design approach for specific application domains will likely be the method of choice from now on.

Challenges and Limitations: The proposed LLaMEA framework presents several limitations and challenges that can be addressed for further advancement. One significant challenge lies in the dependency on the quality and structure of the prompts used to guide the LLM, which can introduce biases or limit the diversity of the generated algorithms. Additionally, the execution reliability of the dynamically

generated code can be problematic, as errors during runtime can affect the evaluation of algorithm performance. Moreover, the computational cost associated with training and querying LLMs may pose scalability issues, particularly for extensive or multiobjective optimization problems. Addressing these challenges would enhance the robustness and applicability of the LLaMEA framework across different domains and more complex optimization scenarios.

Outlook: The promising results of the LLaMEA framework pave the way for several exciting directions for future research as follows.

1) Future work could explore the expansion of the LLaMEA framework to support a broader range of evolutionary strategies. While the proposed framework focuses on an $(1+1)$-EA approach, where we have one parent and generate one solution at a time, it is possible to generalize the framework to a $(\mu + \lambda)$-EA (i.e., an algorithm with parent population size μ and offspring population size $\lambda \gg \mu$), as in population-based evolutionary algorithms [2], keeping a larger population and generating multiple individuals. This would translate into generating multiple mutations and recombinations with LLMs by leveraging multiple random seeds and different temperature values in the generation process.
2) For generating diverse and innovative algorithm candidates, a population-based $(\mu + \lambda)$-EA could use different LLMs in parallel and control the *temperature parameter* of the LLMs to behave more explorative in the beginning of the search [58].
3) Applying the LLaMEA methodology to other algorithm classes for continuous optimization problems, such as Bayesian and more general surrogate-assisted algorithms, could further demonstrate the versatility of LLaMEA.

By continuing to develop and refine these approaches, we anticipate that the integration of LLMs in algorithmic design will significantly advance the field of EC, leading to more intelligent, adaptable, and efficient systems.

Declaration

Disclosure of Interests: The authors have no competing interests to declare that are relevant to the content of this article.

Reproducibility Statement: We provide an open-source documented implementation of our package at [39]. Intermediate results, generated algorithms and BBOB evaluation results are available as well in subdirectories of the repository.

References

[1] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Berlin, Germany: Springer, 2015.

[2] T. Bäck, D. B. Fogel, and Z. Michalewicz, *Handbook of Evolutionary Computation*, vol. 97. New York, NY, USA: Oxford Univ. Press, 1997.

[3] J. Kennedy and R. C. Eberhart, *Swarm Intelligence*. Burlington, MA, USA: Morgan Kaufmann, 2001.

[4] M. Dorigo and T. Stützle, *Ant Colony Optimization*. Cambridge, MA, USA: MIT Press, 2004.

[5] D. J. G. Sánchez, A. Gaspar-Cunha, J. D. H. Sosa, E. Minisci, and A. Zamuda, *Evolutionary Algorithms in Engineering Design Optimization*. Basel, Switzerland: MDPI, 2022.

[6] H. Iba and N. Noman, *Real-World Applications of Evolutionary Algorithms*. London, U.K.: Imperial College Press, 2011, pp. 211–262. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/9781848166820_0006

[7] R. Chiong, T. Weise, and Z. Michalewicz, *Variants of Evolutionary Algorithms for Real-World Applications*, vol. 2. Berlin, Germany: Springer, 2012.

[8] Z. Ma, G. Wu, P. N. Suganthan, A. Song, and Q. Luo, "Performance assessment and exhaustive listing of 500+ nature-inspired metaheuristic algorithms," *Swarm Evol. Comput.*, vol. 77, Mar. 2023, Art. no. 101248.

[9] J. Del Ser et al., "More is not always better: Insights from a massive comparison of meta-heuristic algorithms over real-parameter optimization problems," in *Proc. IEEE Symp. Ser. Comput. Intell. (SSCI)*, 2021, pp. 1–7.

[10] D. Vermetten, C. Doerr, H. Wang, A. V. Kononova, and T. Bäck, "Large-scale benchmarking of metaphor-based optimization heuristics," in *Proc. Genet. Evol. Comput. Conf.*, New York, NY, USA, 2024, pp. 41–49. [Online]. Available: https://doi.org/10.1145/3638529.3654122

[11] S. van Rijn, H. Wang, M. van Leeuwen, and T. Bäck, "Evolving the structure of evolution strategies," in *Proc. IEEE Symp. Ser. Comput. Intell. (SSCI)*, 2016, pp. 1–8.

[12] D. Vermetten, M. López-Ibáñez, O. Mersmann, R. Allmendinger, and A. V. Kononova, "Analysis of modular CMA-ES on strict box-constrained problems in the SBOX-COST benchmarking suite," in *Proc. Compan. Conf. Genet. Evol. Comput.*, New York, NY, USA, 2023, pp. 2346–2353. [Online]. Available: https://doi.org/10.1145/3583133.3596419

[13] D. Vermetten, F. Caraffini, A. V. Kononova, and T. Bäck, "Modular differential evolution," in *Proc. Genet. Evol. Comput. Conf.*, New York, NY, USA, 2023, pp. 864–872. [Online]. Available: https://doi.org/10.1145/3583131.3590417

[14] C. L. Camacho-Villalón, M. Dorigo, and T. Stützle, "PSO-X: A component-based framework for the automatic design of particle swarm optimization algorithms," *IEEE Trans. Evol. Comput.*, vol. 26, no. 3, pp. 402–416, Jun. 2022.

[15] C. L. Camacho-Villalón, T. Stützle, and M. Dorigo, "Designing new metaheuristics: Manual versus automatic approaches," *Intell. Comput.*, vol. 2, p. 48, Dec. 2023. [Online]. Available: https://spj.science.org/doi/abs/10.34133/icomputing.0048

[16] J. de Nobel, F. Ye, D. Vermetten, H. Wang, C. Doerr, and T. Bäck, "IOHexperimenter: Benchmarking platform for iterative optimization heuristics," 2022, *arXiv:2111.04077*.

[17] J. de Nobel, F. Ye, D. Vermetten, H. Wang, C. Doerr, and T. Bäck, "IOHexperimenter: Benchmarking platform for iterative optimization heuristics," *Evol. Comput.*, vol. 32, no. 3, pp. 205–210, Feb. 2024. [Online]. Available: https://doi.org/10.1162/evco_a_00342

[18] F. Neumann et al., "Benchmarking algorithms for submodular optimization problems using IOHprofiler," in *Proc. IEEE Congr. Evol. Comput., (CEC)*, Chicago, IL, USA, 2023, pp. 1–9. [Online]. Available: https://doi.org/10.1109/CEC53210.2023.10254181

[19] C. Doerr, F. Ye, N. Horesh, H. Wang, O. M. Shir, and T. Bäck, "Benchmarking discrete optimization heuristics with IOHprofiler," *Appl. Soft Comput.*, vol. 88, Mar. 2020, Art. no. 106027. [Online]. Available: https://doi.org/10.1016/j.asoc.2019.106027

[20] C. Doerr, H. Wang, F. Ye, S. van Rijn, and T. Bäck, "IOHprofiler: A benchmarking and profiling tool for iterative optimization heuristics," 2018, *arXiv:1810.05281*.

[21] C. Doerr, H. Wang, D. Vermetten, T. Bäck, J. de Nobel, and F. Ye, "Benchmarking and analyzing iterative optimization heuristics with IOHprofiler," in *Proc. Compan., Conf. Genet. Evol. Comput., (GECCO)*, Lisbon, Portugal, 2023, pp. 938–945. [Online]. Available: https://doi.org/10.1145/3583133.3595057

[22] H. Wang, D. Vermetten, F. Ye, C. Doerr, and T. Bäck, "IOHanalyzer: Detailed performance analyses for iterative optimization heuristics," *ACM Trans. Evol. Learn. Optim.*, vol. 2, no. 1, pp. 1–29, 2022. [Online]. Available: https://doi.org/10.1145/3510426

[23] N. Hansen and R. Ros, "Black-box optimization benchmarking of NEWUOA compared to BIPOP-CMA-ES: On the BBOB noiseless testbed," in *Proc. 12th Annu. Conf. Compan. Genet. Evol. Comput.*, 2010, pp. 1519–1526.

[24] M. López-Ibáñez, D. Vermetten, J. Dreo, and C. Doerr, "Using the empirical attainment function for analyzing single-objective black-box optimization algorithms," 2024, *arXiv:2404.02031*.

[25] Q. Guo et al., "Connecting large language models with evolutionary algorithms yields powerful prompt optimizers," 2024, *arXiv:2309.08532*.

[26] T. Rios, S. Menzel, and B. Sendhoff, "Large language and text-to-3D models for engineering design optimization," 2023, *arXiv:2307.01230*.

[27] Y. Zhou et al., "Large language models are human-level prompt engineers," in *Proc. NeurIPS Found. Models Decis. Making Workshop*, 2022, pp. 1–43. [Online]. Available: https://openreview.net/forum?id=YdqwNaCLCx

[28] O. Honovich, U. Shaham, S. R. Bowman, and O. Levy, "Instruction induction: From few examples to natural language task descriptions," in *Proc. 61st Annu. Meeting Assoc. Comput. Linguist.*, Toronto, ON, Canada, Jul. 2023, pp. 1935–1952. [Online]. Available: https://aclanthology.org/2023.acl-long.108

[29] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," 2022, *arXiv:2205.11916*. [Online]. Available: https://api.semanticscholar.org/CorpusID:249017743

[30] R. T. Lange, Y. Tian, and Y. Tang, "Large language models as evolution strategies," 2024, *arXiv:2402.18381*.

[31] N. Hansen, S. Finck, R. Ros, and A. Auger, "Real-parameter black-box optimization benchmarking 2009: Noiseless functions definitions," Inria Joint Centre, INRIA, Rocquencourt, France, Rep. RR6829, 2009.

[32] M. Luo, X. Xu, Y. Liu, P. Pasupat, and M. Kazemi, "In-context learning with retrieved demonstrations for language models: A survey," 2024, *arXiv:2401.11624*.

[33] Q. Dong et al., "A survey on in-context learning," 2023, *arXiv:2301.00234*.

[34] B. Romera-Paredes et al., "Mathematical discoveries from program search with large language models," *Nature*, vol. 625, pp. 468–475, Jan. 2024.

[35] F. Liu, X. Tong, M. Yuan, and Q. Zhang, "Algorithm evolution using large language model," 2023, *arXiv:2311.15249*.

[36] F. Liu et al., "Evolution of heuristics: Towards efficient automatic algorithm design using large language model," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2024, pp. 1–23.

[37] E. Zelikman, E. Lorch, L. Mackey, and A. T. Kalai, "Self-Taught Optimizer (STOP): Recursively self-improving code generation," 2024, *arXiv:2310.02304*.

[38] M. Pluhacek, A. Kazikova, T. Kadavy, A. Viktorin, and R. Senkerik, "Leveraging large language models for the generation of novel metaheuristic optimization algorithms," in *Proc. Compan. Conf. Genet. Evol. Comput.*, New York, NY, USA, 2023, pp. 1812–1820. [Online]. Available: https://doi.org/10.1145/3583133.3596401

[39] N. van Stein. "LLaMEA." Zenodo. Jun. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11358116

[40] R. Cheng, C. He, Y. Jin, and X. Yao, "Model-based evolutionary algorithms: A short survey," *Complex Intell. Syst.*, vol. 4, no. 4, pp. 283–292, 2018.

[41] M. I. E. Khaldi and A. Draa, "Surrogate-assisted evolutionary optimisation: A novel blueprint and a state of the art survey," *Evol. Intell.*, vol. 17, pp. 2213–2243, Aug. 2023.

[42] "ChatGPT-3.5-turbo, version 0125." OpenAI. 2022. Accessed: May 1, 2024. [Online]. Available: https://platform.openai.com/docs/models/gpt-3-5-turbo

[43] "ChatGPT-4-turbo." OpenAI. 2023. Accessed: May 1, 2024. [Online]. Available: https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4

[44] "ChatGPT-4o." OpenAI. 2023. Accessed: May 14, 2024. [Online]. Available: https://platform.openai.com/docs/models/gpt-4o

[45] N. van Stein, D. Vermetten, A. V. Kononova, and T. Bäck, "Explainable benchmarking for iterative optimization heuristics," 2024, *arXiv:2401.17842*.

[46] N. Hansen, A. Auger, D. Brockhoff, and T. Tušar, "Anytime performance assessment in blackbox optimization benchmarking," *IEEE Trans. Evol. Comput.*, vol. 26, no. 6, pp. 1293–1305, Dec. 2022.

[47] M. A. Jaro, "Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida," *J. Am. Statist. Assoc.*, vol. 84, no. 406, pp. 414–420, 1989. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/01621459.1989.10478785

[48] D. Brockhoff, A. Auger, and N. Hansen, "Comparing mirrored mutations and active covariance matrix adaptation in the IPOP-CMA-ES on the noiseless BBOB testbed," in *Proc. 14th Annu. Conf. Compan. Genet. Evol. Comput.*, 2012, pp. 297–304.

[49] P. Pošík and V. Klemš, "Benchmarking the differential evolution with adaptive encoding on noiseless functions," in *Proc. 14th Annu. Conf. Compan. Genet. Evol. Comput.*, 2012, pp. 189–196.

[50] J. de Nobel, D. Vermetten, H. Wang, C. Doerr, and T. Bäck, "Tuning as a means of assessing the benefits of new ideas in interplay with existing algorithmic modules," in *Proc. Genet. Evol. Comput. Conf. Compan. (GECCO)*, 2021, pp. 1375–1384. [Online]. Available: https://doi.org/10.1145/3449726.3463167

[51] L. Huang et al., "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," 2023, *arXiv:2311.05232*.

[52] J. Zhang and A. C. Sanderson, "JADE: Adaptive differential evolution with optional external archive," *IEEE Trans. Evol. Comput.*, vol. 13, no. 5, pp. 945–958, Oct. 2009.

[53] X. Xia et al., "NFDDE: A novelty-hybrid-fitness driving differential evolution algorithm," *Inf. Sci.*, vol. 579, pp. 33–54, Nov. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025521007726

[54] C. Aranha et al., "Metaphor-based metaheuristics, a call for action: The elephant in the room," *Swarm Intell.*, vol. 16, no. 1, pp. 1–6, 2022. [Online]. Available: https://doi.org/10.1007/s11721-021-00202-9

[55] C. L. Camacho-Villalón, M. Dorigo, and T. Stützle, "The intelligent water drops algorithm: Why it cannot be considered a novel algorithm-a brief discussion on the use of metaphors in optimization," *Swarm Intell.*, vol. 13, nos. 3–4, pp. 173–192, 2019. [Online]. Available: https://doi.org/10.1007/s11721-019-00165-y

[56] C. L. Camacho-Villalón, M. Dorigo, and T. Stützle, "An analysis of why cuckoo search does not bring any novel ideas to optimization," *Comput. Oper. Res.*, vol. 142, Jun. 2022, Art. no. 105747.

[57] C. L. Camacho-Villalón, T. Stützle, and M. Dorigo, "Grey wolf, firefly and bat algorithms: Three widespread algorithms that do not contain any novelty," in *Proc. Swarm Intell. (ANTS)*, 2020, pp. 121–133. [Online]. Available: https://doi.org/10.1007/978-3-030-60376-2_10

[58] M. Peeperkorn, T. Kouwenhoven, D. Brown, and A. Jordanous, "Is temperature the creativity parameter of large language models?" 2024, *arXiv:2405.00492*.