Proceedings of the 20th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 47–57
DOI: 10.15439/2025F1481
ISSN 2300-5963 ACSIS, Vol. 43

Optimizing the Optimizer: An Example Showing the Power of LLM Code Generation

Camilo Chacón Sartori
0000-0002-8543-9893
Artificial Intelligence Research Institute (IIIA-CSIC)
Campus UAB, Bellaterra, 08193 Barcelona, Spain
Email: cchacon@iiia.csic.es
*Corresponding author.

Christian Blum
0000-0002-1736-3559
Artificial Intelligence Research Institute (IIIA-CSIC)
Campus UAB, Bellaterra, 08193 Barcelona, Spain
Email: christian.blum@iiia.csic.es

***Abstract*—The integration of Large Language Models (LLMs) into optimization has created a powerful synergy, opening exciting research opportunities. This paper investigates how LLMs can enhance existing optimization algorithms. Using their pre-trained knowledge, we demonstrate their ability to propose innovative heuristic variations based on a semantic understanding of the algorithm's components. To evaluate this, we applied a non-trivial optimization algorithm, Construct, Merge, Solve & Adapt (CMSA)—a hybrid metaheuristic for combinatorial optimization problems that incorporates a heuristic in the solution construction phase. Our results show that an alternative heuristic proposed by GPT-4o outperforms the expert-designed heuristic of CMSA, with the performance gap widening on larger and denser graphs.**

I. Introduction

THERE is a wide variety of optimization algorithms of all kinds and flavors. A simple search using the term 'optimization algorithm' in databases like Scopus or platforms like GitHub yields thousands of results, with that number growing every year. Additionally, optimization researchers often maintain private collections of their algorithms. All these algorithms—open-source or proprietary, and implemented in diverse programming languages—can potentially be improved. By refining their original code with modern techniques and technologies, we can achieve more efficient designs and implementations that go beyond what their creators originally envisioned.

In recent years, the development and growth of Large Language Models (LLMs)—popularized by models such as OpenAI's GPT-4 [24], Anthropic's Claude [31], Google's Gemini [32], Meta's Llama 3 [33], and recently DeepSeek [11]—has opened the door to a wealth of new possibilities. Among the most transformative advancements is code generation. Tools like GitHub Copilot[1], which integrates seamlessly with code editors like Visual Studio Code[2], Windsurf[3], and Cursor[4], an editor with built-in LLM capabilities, have become essential tools for many software developers, revolutionizing their daily workflows. For example, a Python developer could request an LLM to generate a template for invoking external APIs, and the LLM would automatically produce the required code. LLMs have become invaluable for streamlining coding tasks, especially those that are routine and highly repetitive [14]. Therefore, it is natural to wonder: if LLMs excel at simple programming tasks, could they also aid in improving sophisticated optimization algorithms?

One of the recent examples of leveraging LLMs in optimization algorithms is the development of frameworks for generating new black-box metaheuristics [30]. Finding an effective metaheuristic, for example, for a combinatorial optimization problem can be a significant challenge. However, using LLMs to build upon an existing metaheuristic as context and employing them as tools to discover new heuristics or more efficient implementations (e.g., reducing RAM usage or computation time) in a given programming language remains a largely unexplored area. Implementing metaheuristics, unlike other algorithms, requires a focus on computational efficiency, mathematical expressions in scientific computing, and efficient data structure selection. Thus, designing new algorithms in this field demands expert implementation skills.

In this paper, we demonstrate how LLMs like GPT-4o can be leveraged to enhance a sophisticated optimization algorithm, specifically the Construct, Merge, Solve & Adapt (CMSA) hybrid metaheuristic [1, 3, 2]. Starting with an expert-developed C++ implementation of CMSA for the Maximum Independent Set (MIS) problem (of approximately 400 code lines), we employed an in-context prompting strategy combined with interactive dialogue (see Figure 1). Our results show that the LLM successfully comprehended the complex logic and parameter interactions within the CMSA implementation for MIS, demonstrating algorithmic insight to discover novel heuristics while suggesting improvements to the C++ codebase. This successful example opens up new possibilities for enhancing existing complex optimization algorithms by using LLMs as assistants.

The paper unfolds as follows. In Section II, we introduce code generation using LLMs, making it accessible for readers without prior experience in this field. We also explain the MIS problem and provide a brief overview of the CMSA algorithm. Next, in Section III,We present our methodology for enhancing

[1]https://github.com/features/copilot
[2]https://code.visualstudio.com/
[3]https://windsurf.com/editor
[4]https://www.cursor.com/

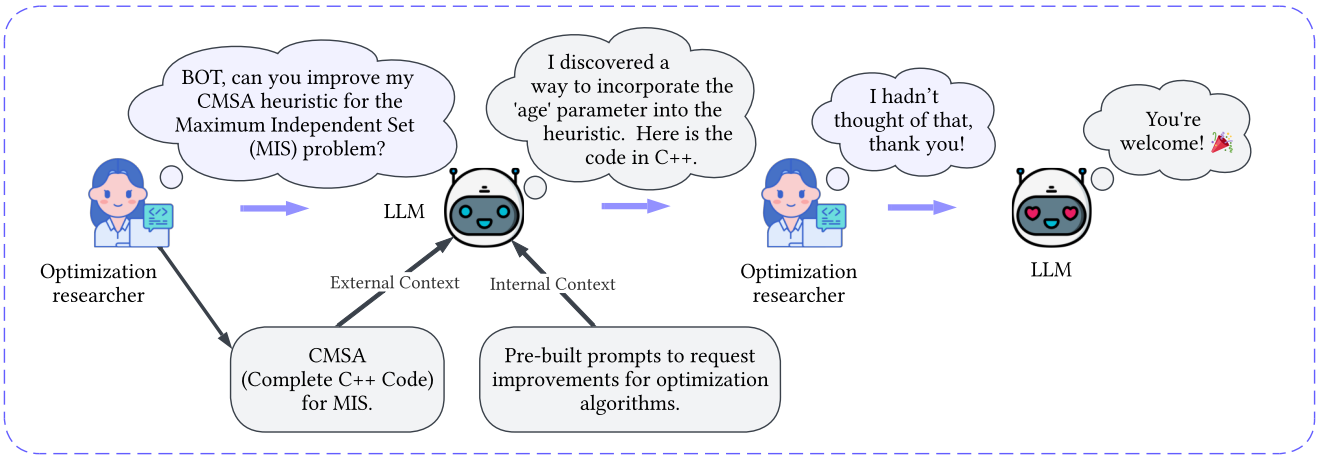


Fig. 1: : A dialogue showing how a chatbot applies our approach to improving optimization algorithms.

CMSA for the MIS problem using LLMs, detailing the process and providing steps for result reproducibility. Section IV presents our experimental results and their interpretation. The limitations of our approach and future research directions are discussed in Section V. The paper concludes by summarizing our key findings and emphasizing the potential impact of LLMs on existing metaheuristics.

II. Background

A. Code Generation with LLMs

Code generation is one of the primary research areas in LLMs [14, 16, 6]. The concept is straightforward: given a prompt, such as *"I need an algorithm to sort a list of numbers in Python"*, the LLM is expected to return a corresponding algorithm (e.g., `Quicksort`) implemented in Python. This capability arises because LLMs are trained on massive amounts of data, which include code from diverse programming languages available on the internet. These codes are sourced not only from GitHub repositories but also from StackOverflow, technical documentation, scientific articles, and publicly available books.

A more explicit prompt can lead to more sophisticated responses. For example, the prompt *"I need an efficient sorting algorithm for a list of numbers in Python, which can be processed in parallel, utilizing modern optimization techniques"* in GPT-4o generates a `ParallelMergeSort`. While efficient, it can be further refined by interacting with the model. For instance, asking *"Find new ways to improve it"* results in an enhanced version using techniques like optimized `Quicksort` for small arrays, better memory management with `NumPy`, and heap merge for combining sorted arrays. This approach leverages `MergeSort`'s ease of parallelization. However, the model might have suggested a different algorithm if the focus was on memory optimization without parallelism. This emphasizes the importance of prompt design and iterative refinement of responses [39].

LLM response quality depends on the data used for training, and with the refinement of datasets and the increase in size, their performance in code generation has improved [38]. Moreover, no LLM is flawless when it comes to generating error-free code. Continuing with the `ParallelMergeSort` example, a model might return code with bugs, so interaction with the model (e.g., copying and pasting error messages from the Python interpreter) is necessary to refine the responses. One could even ask the model to generate its own test suite to verify its algorithm. Different LLMs for these tasks are generally evaluated through benchmarks. One of the most widely used is HumanEval [7], which features programming challenges that assess language comprehension, algorithmic competencies, and basic mathematics—some of which are comparable to straightforward software developer interview questions.

Furthermore, *code generation using LLMs* is a broad area, as code may originate or be destined for very different domains, including the following: data science code for analyzing data and building predictive models [35]; systems code for managing hardware and low-level operations [17, 10]; frontend code development for web applications and UI [37]; and optimization code for solving complex computational problems. Each domain presents its own unique challenges and requirements, with our focus being on the latter, specifically in the field of metaheuristics.

Concerning the automatic generation of metaheuristics with LLMs, given the famous "No Free Lunch Theorem" [36], researchers understand that no single metaheuristic algorithm can outperform all others across all optimization problems. This fundamental principle naturally leads to an interesting possibility: LLMs could serve as powerful automatic generators of black-box metaheuristics, significantly reducing the time needed to find the 'best' implementation of a metaheuristic for a specific problem. In this context, LLMs could be tasked with discovering novel variations and operators that

create new metaheuristics—potentially even surpassing state-of-the-art algorithms for particular problems [30].

The first notable demonstration of heuristic discovery using LLMs was conducted by FunSearch [27], which showcased the potential of leveraging LLMs to generate novel heuristics for the Bin Packing (BP) Problem through a heuristic evolution process. However, a recent study [29] points out that the heuristics developed by FunSearch face challenges in generalizing across diverse BP problem instances. While FunSearch relies on an incomplete or suboptimal base program as a starting point, our approach starts with a complete and carefully designed implementation of an optimization algorithm. Other related works can be summarized as follows. Stein and Bäck [30]'s LlaMEA is a framework that integrates evolutionary algorithms with LLMs to iteratively generate and refine novel black-box metaheuristics during runtime. Similarly, Hemberg et al. [13] proposed LLM GP, combining genetic programming with LLMs to evolve operators through LLM assistance. Meanwhile, Pluhacek et al. [26] demonstrated via prompt engineering that LLMs can generate innovative metaheuristics by identifying and decomposing six high-performing swarm algorithms for continuous optimization.

While these studies already give a glimpse of possible use cases for LLMs in optimization, they primarily focus on creating new algorithms rather than improving existing ones—that is, building upon pre-existing complex code. This distinction is significant given the vast landscape of published optimization algorithms—a simple GitHub search for 'optimization algorithm' returns over 20,000 results, representing a wealth of algorithm implementations that could benefit from improvement. This is why we believe an opportunity exists to create a subfield focused on generating code from existing optimization algorithms. Building on this, our work explores using LLMs to enhance implemented algorithms, enabling the models to uncover new heuristics overlooked by experts in the original implementations. In this sense, our approach treats LLMs as assistants to researchers, not replacements. By doing so, we leverage LLMs to work with existing code (as context) and improve it using the current knowledge these black-box model models have gained during their pre-training phase.

B. Maximum Independent Set (MIS) Problem

To validate our hypothesis that LLMs can enhance existing optimization algorithms, we will employ an algorithm (detailed in the following subsection) to solve the Maximum Independent Set (MIS) problem, a well-studied and NP-hard combinatorial optimization problem with applications in network design, scheduling, and bioinformatics. Formally, the MIS problem is defined as follows: Given an undirected graph $G = (V, E)$, the objective is to find a largest subset $S \subseteq V$ such that no two vertices in S are adjacent in G, i.e., there is no edge $(u, v) \in E$ for all pairs $u \neq v \in S$. Figure 2 shows three examples of optimal MIS solutions (non-white nodes) in different graphs.

C. CMSA

Construct, Merge, Solve & Adapt (CMSA) is a hybrid metaheuristic, also known as a matheuristic, that combines elements of classical metaheuristics with exact solvers (such as Integer Linear Programming (ILP) solvers) for combinatorial optimization [3]. Each CMSA iteration is a sequence of four fundamental phases:

1) **Construct:** Generates solutions probabilistically through a probabilistic greedy mechanism (remember this phase; it will be key in the next section).
2) **Merge:** Combines solution components from the generated solutions to form a reduced subproblem.
3) **Solve:** Applies an exact solver (in the case of the MIS: an ILP solver) to optimally solve the reduced subproblem.
4) **Adapt:** Updates parameters and data structures based on the quality of the solution returned by the solver.

This hybrid architecture combines the efficiency of metaheuristics for search space exploration with the precision of exact methods in reduced spaces. The CMSA algorithm is controlled by some key parameters. First, and most importantly, each solution component has an associated *age* value, which is initialized to zero when a component is added to the subproblem. Moreover, a solution component in the subproblem is subject to an increase of its age value, in case it does not form part of the subproblem's solution generated by the exact solver. In this context, the parameter age_{max} plays a crucial role, defining the maximum age of solution components before they are removed from the incumbent subproblem, thereby preventing stagnation and encouraging diversity. Other parameters include n_a, which sets the number of solution constructions per iteration, t_{max}, the total CPU time, t_{limit}, the time limit for the ILP solver per iteration, and $0 \leq d_{rate} \leq 1$, which controls the determinism rate for solution construction. Note that higher values of d_{rate} lead to more deterministic solution constructions, while lower values increase randomness. Well-chosen values for these parameters ensure a balanced exploration of the search space, effective exploitation of promising areas via exact subproblem solving, and efficient use of computational resources.

We selected CMSA for the purpose of this paper due to its relative complexity in implementation as compared to simple metaheuristics. For instance, implementing CMSA for the MIS problem in C++ presents significant technical challenges. The four phases must be precisely defined, properly integrated, and implemented efficiently to leverage the framework's full potential. This complexity would only increase when dealing with more sophisticated optimization problems. In essence, a CMSA implementation demands expertise not only in metaheuristics and exact methods but also proficiency in C++ (or whatever programming language is chosen for implementation).

For our study, we utilize the original C++ implementation to solve the MIS problem, which was provided by CMSA's inventor and can be downloaded from our repository (https://github.com/camilochs/optimizing-the-optimizer). This choice

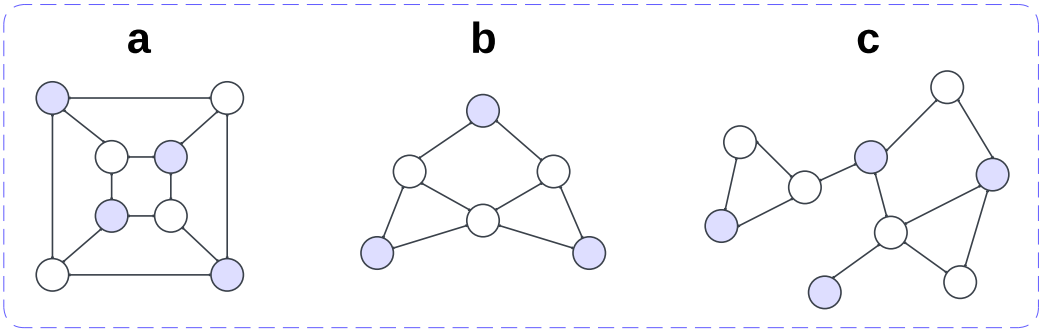


Fig. 2: Examples of maximum independent sets.

serves two crucial purposes: first, it ensures we are working with a well-implemented version of the algorithm, and second, it presents an interesting opportunity to test LLMs' capability to enhance expert-written code. This scenario allows us to evaluate whether LLMs can identify potential improvements even in code developed by domain experts. Our findings are presented in the following section.

III. LLM-Improved CMSA for MIS

In this section, we present a methodology for leveraging LLMs to improve existing optimization algorithms. By "improve existing optimization algorithms," we specifically mean expert-crafted implementations that have already been optimized for performance, rather than simplistic or baseline versions. In contrast, prior approaches often start from scratch—generating code from an initial prompt and refining it iteratively—or rely on overly simplified implementations that fail to reflect the complexity of real-world optimization problems. Our approach centers on dialog-based interaction with an LLM via a chatbot interface, uncovering novel improvements in the code that even an expert can benefit from. We detail this in the following subsections.

A. Discovering New Heuristics

LLMs are pattern-recognition machines [23, 28], and code is a rich source of structured textual patterns. This allows LLMs to identify underutilized elements—such as variables or functions—that could play a strategic role within a heuristic. Such insights may be missed by human experts, especially in complex codebases. These discoveries require a deep syntactic and semantic understanding of both the code and the optimization problem, enabling them to suggest meaningful modifications to heuristic strategies, going beyond superficial code changes. In this way, LLMs can serve as advanced assistants for optimization algorithm designers, offering novel heuristic improvements informed by reasoning over their vast pretraining knowledge.

To demonstrate our methodology, we employ an LLM to improve one of the heuristics in the CMSA code used to solve the MIS problem. Specifically, the function `generation_solution()` implements the solution construction phase of the CMSA framework [1]:

```
while (int(positions.size()) > 0) {
    double dec = standard_distribution(generator);
    int position = 0;

    if (dec <= determinism_rate) {
        position = *(positions.begin());
    } else {
        int max = candidate_list_size;
        if (max > int(positions.size())) {
            max = int(positions.size());
        }

        double rnum = standard_distribution(generator);
        int pos = produce_random_integer(max, rnum);

        set<int>::iterator sit2 = positions.begin();
        for (int i = 0; i < pos; ++i) {
            ++sit2;
        }
        position = *sit2;
    }

    greedy_sol.score += 1;
    greedy_sol.vertices.insert(increasing_degree_order[
     position]);

    if (age[increasing_degree_order[position]] == -1) {
        age[increasing_degree_order[position]] = 0;
    }

    positions.erase(position);

    for (auto sit = neigh[increasing_degree_order[position
     ]].begin();
         sit != neigh[increasing_degree_order[position]].
     end(); ++sit) {
        positions.erase(position_of[*sit]);
    }
}
```

Listing 1: Probabilistic greedy algorithm for MIS in CMSA.

It implements a greedy randomized construction heuristic that selects exactly one vertex $v_i \in \tilde{V}$ at each step (where $\tilde{V}$ is the set of nodes that can feasibly be chosen), until the MIS solution is complete:

$$v_i = \begin{cases} v_{\text{min}} & \text{if } r \leq \alpha \\ v_{\text{random}} \in \text{CL}(k) & \text{otherwise} \end{cases}$$

where

- v_{min} is the vertex with minimum degree (among the ones from $\tilde{V}$)

- r is a random number between $[0, 1]$
- α is the determinism rate
- CL(k) is a candidate list of size k
- v_{random} is a randomly selected vertex from the candidate list

The heuristic combines deterministic greedy selection (based on vertex degree) with randomization to create diverse solutions. It selects vertices either by taking the best available vertex (greedy choice) with probability α, or by randomly selecting from a restricted candidate list with probability $(1 - \alpha)$.

1) New heuristic from the LLM: The original greedy heuristic, while efficient in terms of runtime, fails to incorporate the `age` variable—a key element of the CMSA framework. It only uses the `age` variable to restart the solutions, meaning it is not utilized in the candidate selection mechanism. In Figure 3, we present two example dialogues of interactions with the LLM according to our approach. Let us begin with the first one. In case (a), the LLM proposes an enhanced heuristic that takes into account both node degrees and the current age values. The vertex selection mechanism suggested by the LLM can be technically described as follows:

$$v_i = \begin{cases} \text{argmin}\{P_w(v_j) \mid v_j \in \tilde{V}\} & \text{if } r \leq \alpha \\ \text{roulette-wheel selection w.r.t. } P_w(.) \text{ values} & \text{otherwise} \end{cases}$$

where $P_w(v_j)$ is a weighted probability:

$$P_w(v_j) = \frac{w(v_j)}{\sum_{v_l \in V} w(v_l)}, \quad w(v) = \frac{1}{2 + age(v)} + \frac{1}{1 + \text{degree}(v)} \quad (1)$$

The weight function $w(v)$ favors vertices with low age and degree values to promote diversity in the selection process. Thus, the LLM was able to change the original heuristic by incorporating the `age` values to diversify the node selection process, while balancing it with degree information through a composite weight function. It makes a lot of sense to decrease the probability of solution components with high age values being incorporated in newly constructed solutions.

Although the LLM successfully identifies an overlooked use of the `age` variable within the CMSA construction heuristic, any dialog-based system ultimately requires human feedback [5]. As such, the LLM occasionally makes mistakes. For example, as shown in Figure 3 (a), the line `double weight = 1.0 / (1 + age[v])` may result in a division by zero, since `age` values in CMSA are set to `-1` for solution components that do not belong to the subproblem. The human-provided fix consisted in replacing `1 + age[v]` with `2 + age[v]`. Notably, this kind of use of the `age` variable—despite its initial flaw and the correction provided through human feedback—had never been considered by any researchers working on CMSA algorithms.

> This new CMSA variant is henceforth called `LLM-CMSA-V1`. It was obtained by replacing the `generate_solution()` of the original CMSA implementation with the new function provided by the LLM.

2) Improving `LLM-CMSA-V1` with the LLM: Since our methodology relies on dialog-based interactions with the LLM via a chatbot interface, it is also possible to request improvements to previously generated code—still available in the current session context. For example, one might ask: *"Are there ways to enhance the dynamic selection heuristic to allow for a more diverse and advanced search?"* The LLM responds with several suggestions, one of which involves incorporating the concept of entropy. The C++ code provided by the LLM is characterized by a corresponding change, which involves replacing the definition of $P_w(.)$ (see Equation 1) with the following entropy-adjusted probabilities:

$$P_H(v_j) = \frac{P_w(v_j) + H}{\sum_{v_l \in V} P_w(v_l) + H}$$

where

$$H = -\sum_{v_l \in V} P_w(v_l) \log(P_w(v_l))$$

The entropy adjustment increases selection diversity by adding the system's uncertainty to each probability.

> This CMSA variant is henceforth called `LLM-CMSA-V2`. We directly replaced the `generate_solution()` function of the original CMSA with the LLM-generated code.

B. Code Optimization Strategies

After discovering the two new CMSA variants with the help of the LLM, it is also possible to request a different kind of improvement—not in terms of proposing new algorithmic heuristics, but rather in enhancing the underlying C++ code. These enhancements may involve the use of more efficient data structures, changes in data types, or other low-level optimizations that preserve the algorithmic logic. For instance, one might ask: *"Could the LLM create an improvement at the C++ code level?"* In other words, is there a more efficient way to implement the new CMSA variants without altering their core behavior?

It is reasonable to assume that, since the LLM has been pretrained on vast amounts of source code, it could suggest highly optimized C++ implementations—even for contexts such as optimization algorithm design (e.g., metaheuristics), where low-level improvements to data structures or code organization are not typically the focus of human designers. This leads us to the next prompt, which we apply to both previously generated heuristics, `LLM-CMSA-V1` and `LLM-CMSA-V2`:[5]

[5]This prompt uses the "C++ code" checkbox, as shown in Figure 3 (b).

Fig. 3: Two LLM interaction patterns: (a) a direct request to improve a heuristic using CMSA's `age` parameter, and (b) an iterative dialogue to enhance both heuristic quality and C++ performance through error correction. Both use in-context learning as a prompting strategy [12, 20].

Human prompt

Given the following heuristic implemented in C++ (`generate_solution()`), without altering its core logic or functionality, please analyze the code to identify potential **improvements in the use of data structures, cache optimization, and other low-level optimizations. Focus on enhancing performance by suggesting more efficient data structures, reducing memory overhead, improving data locality, and leveraging modern C++ features where applicable.** Please provide an updated version of the code with comments explaining each optimization.

However, dialog-based interactions with the LLM are not free from hallucinations—that is, the model may generate code with bugs or memory management issues in C++ (as shown in Figure 3 (b)). One effective way to address these problems is through human feedback, engaging in a trial-and-error refinement process with the model—a common practice in code generation workflows (see [5]). Eventually, the LLM produces code that compiles successfully. Upon reviewing the final output, it becomes evident that the generated code is more complex, less readable, and incorporates unconventional or less familiar programming techniques. For example, it replaces the `set` and `vector` data structures with more advanced, low-level alternatives:[6]

```
#include <bitset>
...
// Constants for optimization
constexpr size_t BLOCK_SIZE = 64;
...
// Bitset for boolean operations
std::bitset<32768> available;
// Adjust size based on max n_of_vertices
available.set();
// Aligned vector to optimize cache usage
alignas(BLOCK_SIZE) std::vector<int> active_vertices;
active_vertices.reserve(n_of_vertices);
```

Listing 2: Fragment code optimization suggested by LLM.

Meanwhile, the original code is shown in the following listing.

```
set<int> positions;
vector<int> position_of(n_of_vertices, 0);
for (int i = 0; i < n_of_vertices; ++i) {
    positions.insert(i);
    position_of[increasing_degree_order[i]] = i;
}
```

Listing 3: Fragment of original code from CMSA.

The logic remains the same, except for changes in variable names: `positions` is replaced with `available`, and `position_of` is substituted with `active_vertices`.[7]

Although the changes do not affect the heuristic's logic but rather the underlying C++ structures, they do not always lead to measurable improvements in efficiency, specifically in reducing the runtime, which in practice would allow us to explore better candidates when building a valid CMSA solution. Nevertheless, the generated code compiles and runs correctly.

> The two new CMSA variants with these performance improvements will be named: `LLM-CMSA-V1-PERF` and `LLM-CMSA-V2-PERF`.

This iterative process with the LLM, asking it to identify underutilized variables or functions in the existing code, demonstrates its potential as a sophisticated assistant for optimization experts. For instance, in this case, the LLM proposes a new CMSA construction heuristic utilizing the `age` parameter. Unlike simple code generation, the LLM enhances not only the algorithm itself but also the existing code (e.g., C++ implementations), suggesting improvements without altering its logic. This opens the door to using LLMs not just for developing optimization algorithms but also for updating and refining sophisticated legacy code, leveraging the extensive knowledge embedded in LLMs.

C. Reproducibility

Although it is not possible to replicate the exact results of an LLM, due to their autoregressive nature that predicts the most probable token based on a probability distribution (with the next token being determined stochastically) [18], it is possible to reproduce similar responses by using the same prompts, the same LLM, and its parameters. For this reason, our repository (https://github.com/camilochs/optimizing-the-optimizer) includes a *chatbot* that implements the same prompts used in our research (as shown in Figure 3). In fact, each element in Figure 3 (textbox and checkbox) loads pre-built prompts, known as in-context prompts [12, 20], to eliminate the need for manual input. Thus, our chatbot features two types of in-context prompts: (1) external ones, related to the C++ CMSA code for the MIS, and (2) internal ones, focused on improving the heuristic, the C++ code, and specifying which function in the code requires enhancement. Next, we will assess the quality of the heuristics proposed by the LLM.

IV. Empirical Evaluation

This section is divided into two parts: the preliminary phase (setup, benchmark, and CMSA parameter tuning) and the experimental results.

A. Preliminary

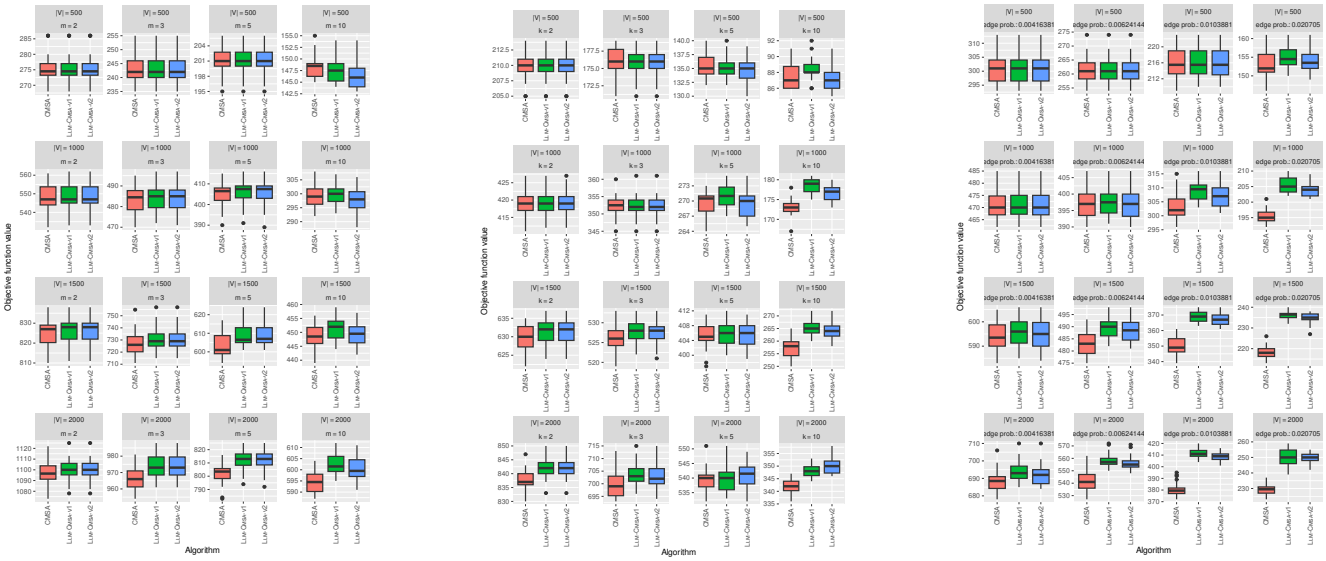
First, we used Chatbot Arena [9] to test various LLMs without incurring costs. [8] For our experiments, we finally selected GPT-4o (version: 2024-11-20)[9] as it is one of the top-performing models to date. Experiments concerning algorithm

[6]The comments in the code were generated by the LLM.

[7]Explicitly instructing the prompt to retain the variable names might have avoided this issue.

[8]https://lmarena.ai/

[9]The parameters used for the LLM were the default values: `temperature = 0.7` (controls randomness in responses), `top-p = 1` (nucleus sampling threshold for token probability), and `max-output-tokens = 2048` (maximum number of tokens in the output).



(a) Barabási–Albert (b) Watts–Strogatz (c) Erdős–Rényi

Fig. 4: Comparative analysis of solution quality: Original CMSA vs. LLM-CMSA variants (V1 and V2).

variants `CMSA`, `LLM-CMSA-V1`, and `LLM-CMSA-V2` were conducted on a cluster equipped with Intel® Xeon® CPU 5670 processors (12 cores at 2.933 GHz) and 32 GB of RAM.

Our benchmark set consists of three types of graphs: Barabási-Albert, Watts-Strogatz, and Erdős-Rényi graphs, with four different sizes and four density levels (see Figure 4). For each combination of size and density level, the benchmark set contains 1 tuning instance and 30 testing instances. This makes a total of 48 tuning instances and 1440 testing instances. We used a limited tuning set to find robust, general parameters, deliberately avoiding the overfitting that would result from tuning on a set identical to our evaluation instances. The parameters of all three CMSA variants (original, V1, V2) were tuned using `irace`, a tool for tuning algorithm parameters based on problem instances [22]. Due to the lack of space (and not being the important point of this paper), we do not report the final values here. We used 150, 300, 450, and 600 CPU seconds as computation time limits for graphs of the four different sizes.

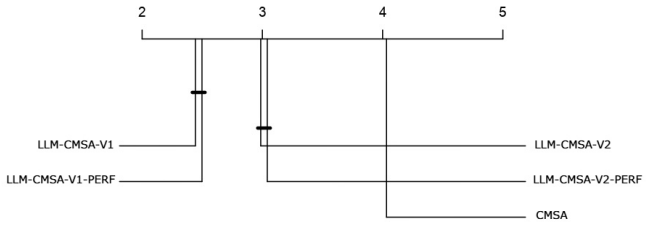
B. Numerical Results

In addition to the three tuned algorithms, we also tested two variants obtained by using the efficiency-improved C++ codes described in Section III-B. These two variants are henceforth called `LLM-CMSA-V1-PERF` and `LLM-CMSA-V2-PERF`. Each of the five algorithms was applied exactly once to each testing instance. The results of the main three variants are reported employing box plots in Figure 4. The considered graph size and density level are indicated at the top of each box plot. The main key observation is that both LLM-generated CMSA variants outperform the standard CMSA variant with growing graph size and density. This clearly shows that the suggestion of the LLM to make use of the age values for solution component selection during solution construction was a very good one.

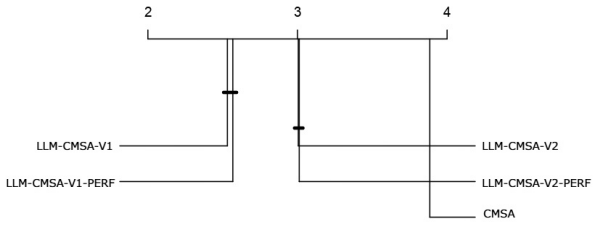
To be able to make statistical claims, we produced so-called critical difference (CD) plots (see Figure 5), in which the algorithms' whiskers show their average ranking concerning a set of problem instances. Moreover, algorithm whiskers are connected by a bold horizontal bar in case they perform statistically equivalent.[10] Figure 5 shows three CD plots, one for each graph type. In these plots we also included the efficiency-optimized LLM-generated CMSA variants. The following observations can be made. Even though both `LLM-CMSA-V1` and `LLM-CMSA-V2` outperform `CMSA` with statistical significance for all three graph types, in all three cases the first variant outperforms the second variant. This means that the idea of using the entropy of the selection probabilities was not fruitful. Moreover, the efficiently-optimized algorithm variants are statistically equivalent to their non-optimized counterparts. This means that, even though they might save RAM, which is a non-tested hypothesis, they do not benefit by producing better results. Furthermore, preliminary checks indicated no significant runtime improvements from these C++ optimizations in our test environment, suggesting the original expert implementation was already highly efficient or that the specific low-level changes were not impactful in this context.

Finally, Figure 6 shows two representative examples of the convergence behavior based on 10 runs of the main three CMSA variants. In both examples, the LLM-generated CMSA

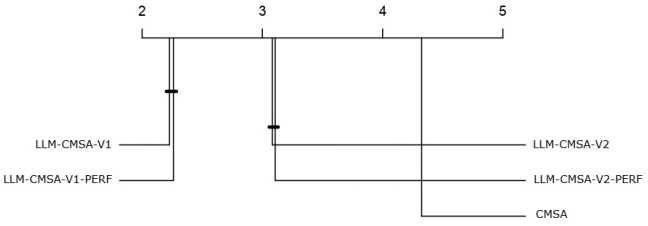
[10]The critical difference (CD) plot, which uses the Friedman test for overall differences and the Nemenyi post-hoc test for pairwise comparisons, is a recognized standard for the statistical comparison of multiple randomized algorithms for combinatorial optimization.



(a) Barabási–Albert

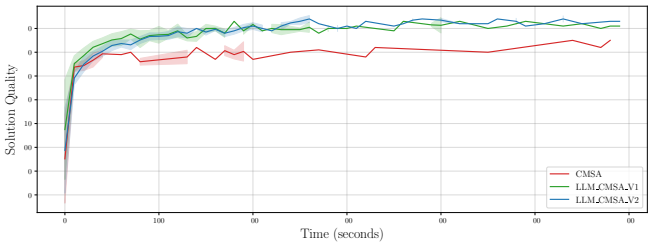


(b) Watts–Strogatz

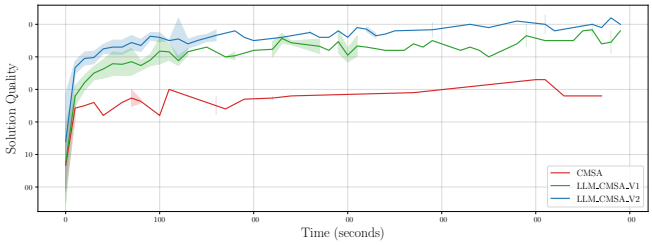


(c) Erdős–Rényi

Fig. 5: Critical difference (CD) plots for all graph types.



(a) Watts–Strogatz (n2000, k10)



(b) Erdős–Rényi (n2000, 020705)

Fig. 6: Examples of algorithm evolution over time.

variants quickly produce solutions of a quality that standard CMSA does not even obtain at the end of its runs.

V. Discussion

Our study highlights the potential for LLMs to engage in algorithmic reasoning, successfully identifying conceptual enhancements for CMSA, such as the `LLM-CMSA-V1` heuristic which outperformed the expert baseline. Interestingly, the LLM's subsequent suggestion incorporating entropy (`LLM-CMSA-V2`), while plausible, was less effective. Perhaps its added complexity was not beneficial, or even detrimental, within the specific CMSA/MIS context, making the simpler LLM-proposed V1 superior. This observation reinforces the need for rigorous empirical validation of any proposed heuristic. However, our study also has the following limitations: due to space constraints, we tested only one LLM (GPT-4o) for CMSA improvements, and future work could benefit from comparing additional LLMs, especially open-weight ones. Additionally, we did not explore using GPT-4o to enhance other complex algorithms for additional optimization problems. These limitations could be addressed in an extended article. Despite these limitations, our comprehensive analysis of CMSA with new heuristics for solving MIS instances presents promising opportunities for new lines of research emerging from our work:

1) **Specialized benchmarks.** While our study focuses on CMSA for the MIS problem, the field lacks benchmarks tailored to optimization. Just as general-purpose code generation relies on standard benchmarks, we need domain-specific ones to evaluate LLMs' ability to discover improved heuristics.
2) **LLM-based agent integration.** Figure 3 illustrates manual interaction, but tasks like execution and debugging could be handled by autonomous agents. A platform where such agents collaborate on improving existing optimization algorithms may yield major advances [15, 21].
3) **Adapt code to the target domain.** Optimization algorithms often require adaptation—not only by switching programming languages to improve performance and maintainability, but also by transitioning from single-threaded execution to parallelism (e.g., using CUDA). LLMs can support this process [25, 19, 8], though domain-specific fine-tuning may be necessary.

An underlying issue in using LLMs for code generation is how we deal with errors. Although they may produce similar outcomes, errors made by a human programmer (or optimization researcher) and those produced by an LLM differ fundamentally in origin. Understanding these conceptual differences is essential for developing sound methodologies for integrating human- and machine-generated code [4].

On the other hand, a fundamental question arises: should the LLM be credited for discovering a superior heuristic compared to the best implementation by a human expert? While prompt design clearly influences the outcome, it raises important questions about authorship, ownership, and the role

of AI in scientific discovery (see [34]).

VI. Conclusions

In our research, we demonstrate that LLMs can be effectively applied to enhance existing optimization algorithms. We used the non-trivial Construct, Merge, Solve, and Adapt (CMSA) algorithm implemented in C++ to solve the classical Maximum Independent Set problem. By leveraging in-context prompts with GPT-4o, the model successfully understood the operational context of the CMSA implementation and proposed conceptually new heuristics for the probabilistic construction phase. After a thorough comparative analysis, the heuristics proposed by the LLM outperformed those manually designed by an expert in CMSA. This highlights the potential of LLMs not merely as tools but as artificial collaborators capable of identifying unused code segments through semantic analysis and contributing meaningfully to the complex task of algorithm design.

Acknowledgment

C. Chacón Sartori and C. Blum were supported by grant PID2022-136787NB-I00 funded by MCIN/AEI/10.13039/501100011033. Moreover, thanks to FreePik, from where we extracted the icons used in Figures 1 and 3.

References

[1] C. Blum. *Construct, Merge, Solve & Adapt: A Hybrid Metaheuristic for Combinatorial Optimization*. Computational Intelligence Methods and Applications. Springer Nature Switzerland, 2024. ISBN 9783031601026. URL https://books.google.es/books?id=ENCt0AEACAAJ.

[2] C. Blum, J. Puchinger, G. R. Raidl, and A. Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, 11(6):4135–4151, 2011. ISSN 1568-4946. doi: https://doi.org/10.1016/j.asoc.2011.02.032. URL https://www.sciencedirect.com/science/article/pii/S1568494611000962.

[3] C. Blum, P. Pinacho, M. López-Ibáñez, and J. A. Lozano. Construct, merge, solve & adapt a new general algorithm for combinatorial optimization. *Computers & Operations Research*, 68:75–88, 2016. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2015.10.014. URL https://www.sciencedirect.com/science/article/pii/S0305054815002452.

[4] C. Chacón Sartori. Architectures of error: A philosophical inquiry into ai-generated and human-generated code, May 2025. URL https://ssrn.com/abstract=5265751. Available at SSRN.

[5] A. Chen, J. Scheurer, T. Korbak, J. A. Campos, J. S. Chan, S. R. Bowman, K. Cho, and E. Perez. Improving code generation by training with natural language feedback. *arXiv [cs.SE]*, Mar. 2023.

[6] L. Chen, Q. Guo, H. Jia, Z. Zeng, X. Wang, Y. Xu, J. Wu, Y. Wang, Q. Gao, J. Wang, W. Ye, and S. Zhang. A survey on evaluating large language models in code generation tasks, 2024. URL https://arxiv.org/abs/2408.16498.

[7] M. Chen and et al. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

[8] W. Chen, J. Zhu, Q. Fan, Y. Ma, and A. Zou. Cuda-llm: Llms can write efficient cuda kernels, 2025. URL https://arxiv.org/abs/2506.09092.

[9] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, H. Zhang, B. Zhu, M. Jordan, J. E. Gonzalez, and I. Stoica. Chatbot arena: An open platform for evaluating llms by human preference, 2024. URL https://arxiv.org/abs/2403.04132.

[10] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve, and H. Leather. Large language models for compiler optimization. *arXiv [cs.PL]*, Sept. 2023.

[11] DeepSeek-AI et al. Deepseek-v3 technical report, 2024. URL https://arxiv.org/abs/2412.19437.

[12] Q. Dong, L. Li, D. Dai, C. Zheng, J. Ma, R. Li, H. Xia, J. Xu, Z. Wu, T. Liu, B. Chang, X. Sun, L. Li, and Z. Sui. A survey on in-context learning, 2024. URL https://arxiv.org/abs/2301.00234.

[13] E. Hemberg, S. Moskal, and U.-M. O'Reilly. Evolving code with a large language model, 2024. URL https://arxiv.org/abs/2401.07102.

[14] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim. A survey on large language models for code generation. *arXiv [cs.CL]*, June 2024.

[15] X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*, 33(7):1–30, Sept. 2024.

[16] S. Joel, J. J. Wu, and F. H. Fard. A survey on llm-based code generation for low-resource and domain-specific programming languages, 2024. URL https://arxiv.org/abs/2410.03981.

[17] S. Joel, J. J. W. Wu, and F. H. Fard. A survey on LLM-based code generation for low-resource and domain-specific programming languages. *arXiv [cs.SE]*, Oct. 2024.

[18] U. Kamath, K. Keenan, G. Somers, and S. Sorenson. *Large Language Models: A Deep Dive: Bridging Theory and Practice*. Springer Nature Switzerland, 2024. ISBN 9783031656477. URL https://books.google.es/books?id=kDobEQAAQBAJ.

[19] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample. Unsupervised translation of programming languages. *arXiv [cs.CL]*, June 2020.

[20] J. Li, G. Li, C. Tao, J. Li, H. Zhang, F. Liu, and Z. Jin. Large language model-aware in-context learning for code generation. *arXiv [cs.SE]*, Oct. 2023.

[21] F. Liu, R. Zhang, Z. Xie, R. Sun, K. Li, X. Lin, Z. Wang, Z. Lu, and Q. Zhang. Llm4ad: A platform for algorithm design with large language model, 2024. URL https:

//arxiv.org/abs/2412.17287.

[22] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016. doi: 10.1016/j.orp.2016.09.002.

[23] S. Mirchandani, F. Xia, P. Florence, B. Ichter, D. Driess, M. G. Arenas, K. Rao, D. Sadigh, and A. Zeng. Large language models as general pattern machines, 2023. URL https://arxiv.org/abs/2307.04721.

[24] OpenAI et al. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

[25] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. *arXiv [cs.SE]*, Aug. 2023.

[26] M. Pluhacek, A. Kazikova, T. Kadavy, A. Viktorin, and R. Senkerik. Leveraging large language models for the generation of novel metaheuristic optimization algorithms. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, GECCO '23 Companion, page 1812–1820, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701207. doi: 10.1145/3583133.3596401. URL https://doi.org/10.1145/3583133.3596401.

[27] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. R. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, P. Kohli, A. Fawzi, J. Grochow, A. Lodi, J.-B. Mouret, T. Ringer, and T. Yu. Mathematical discoveries from program search with large language models. *Nature*, 625:468 – 475, 2023. URL https://api.semanticscholar.org/CorpusID:266223700.

[28] C. C. Sartori, C. Blum, F. Bistaffa, and G. Rodríguez Corominas. Metaheuristics and large language models join forces: Toward an integrated optimization approach. *IEEE Access*, 13:2058–2079, 2025. doi: 10.1109/ACCESS.2024.3524176.

[29] K. Sim, Q. Renau, and E. Hart. Beyond the hype: Benchmarking llm-evolved heuristics for bin packing, 2025. URL https://arxiv.org/abs/2501.11411.

[30] N. v. Stein and T. Bäck. LLaMEA: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Transactions on Evolutionary Computation*, 29(2):331–345, 2025. doi: 10.1109/TEVC.2024.3497793.

[31] A. Team. Introducing Claude 3.5 Sonnet — anthropic.com. https://www.anthropic.com/news/claude-3-5-sonnet, 2024. [Accessed 02-11-2024].

[32] G. Team and et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024. URL https://arxiv.org/abs/2403.05530.

[33] M. Team. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

[34] H. Wang, T. Fu, Y. Du, W. Gao, K. Huang, Z. Liu, P. Chandak, S. Liu, P. Katwyk, A. Deac, A. Anandkumar, K. Bergen, C. P. Gomes, and Shir. Scientific discovery in the age of artificial intelligence. *Nature*, 620(7972):47–60, August 2023. doi: 10.1038/s41586-023-06221-. URL https://ideas.repec.org/a/nat/nature/v620y2023i7972d10.1038_s41586-023-06221-2.html.

[35] Y. Wen, P. Yin, K. Shi, H. Michalewski, S. Chaudhuri, and A. Polozov. Grounding data science code generation with input-output specifications, 2024. URL https://arxiv.org/abs/2402.08073.

[36] D. Wolpert and W. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997. doi: 10.1109/4235.585893.

[37] S. Xiao, Y. Chen, J. Li, L. Chen, L. Sun, and T. Zhou. Prototype2code: End-to-end front-end code generation from ui design prototypes, 2024. URL https://arxiv.org/abs/2405.04975.

[38] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, Y. Wang, and J.-G. Lou. Large language models meet NL2Code: A survey, 2023. URL https://arxiv.org/abs/2212.09420.

[39] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba. Large language models are human-level prompt engineers, 2023. URL https://arxiv.org/abs/2211.01910.