Combinatorial Optimization for All: Using LLMs to Aid Non-Experts in Improving Optimization Algorithms

Camilo Chacón Sartori[*]
Artificial Intelligence Research Institute (IIIA-CSIC)
Bellaterra, Spain
cchacon@iiia.csic.es

Christian Blum
Artificial Intelligence Research Institute (IIIA-CSIC)
Bellaterra, Spain
christian.blum@iiia.csic.es

https://camilochs.github.io/comb-opt-for-all/

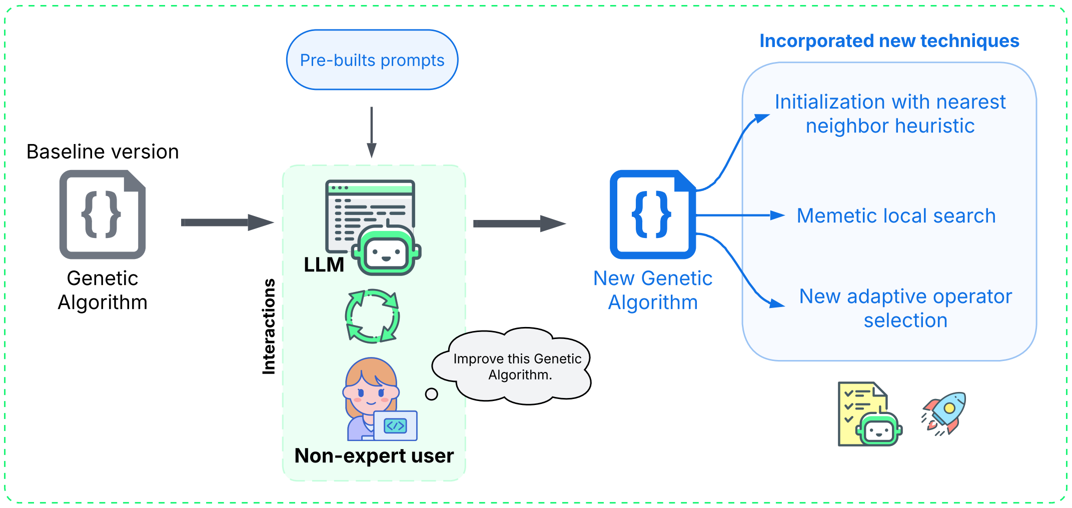

Pre-builts prompts
Incorporated new techniques
Baseline version
Genetic Algorithm
LLM
Interactions
Non-expert user
Improve this Genetic Algorithm.
New Genetic Algorithm
Initialization with nearest neighbor heuristic
Memetic local search
New adaptive operator selection

Figure 1: A non-expert user's interaction with an LLM can enhance an existing genetic algorithm by incorporating modern techniques.

Abstract

Large Language Models (LLMs) have shown notable potential in code generation for optimization algorithms, unlocking exciting new opportunities. This paper examines how LLMs, rather than creating algorithms from scratch, can improve existing ones without the need for specialized expertise. To explore this potential, we selected 10 baseline optimization algorithms from various domains (metaheuristics, reinforcement learning, deterministic, and exact methods) to solve the classic Travelling Salesman Problem. The results show that our simple methodology often results in LLM-generated algorithm variants that improve over the baseline algorithms in terms of solution quality, reduction in computational time, and simplification of code complexity, all without requiring specialized optimization knowledge or advanced algorithmic implementation skills.

Keywords Large Language Models · Hybrid Metaheuristics · Combinatorial Optimization · CMSA

[*]Corresponding author: cchacon@iiia.csic.es

1 Introduction

If we asked you how many optimization algorithms exist, would you be able to come up with an exact answer? Probably not, as there are simply too many algorithms (or algorithm variants) to count. A simple search for ‘optimization algorithm’ in databases like Scopus, IEEE Xplore, or GitHub returns thousands of results. And that is just the start. Classic algorithms, such as the Genetic Algorithm, have spawned so many variations that some barely resemble the original. On top of that, hybrid approaches combine heuristics with exact methods or machine learning techniques. New algorithms are introduced every day.

All these algorithms—whether publicly available or proprietary—could be enhanced with modern technologies. This would benefit not just optimization specialists but also non-expert users. By integrating advanced techniques into an algorithm’s implementation, we can improve its performance regardless of the programming language used.

The emergence of Large Language Models (LLMs)—AI systems trained on vast text corpora that can generate code and assist with complex tasks—showcased by innovations like OpenAI’s GPT-O1 [24], Anthropic’s Claude [1], Google’s Gemini [34], Meta’s Llama [35], and the newly unveiled DeepSeek R1 [33], has significantly broadened possibilities for scientific innovation. These models demonstrate remarkable code generation capabilities through tools like GitHub Copilot[2] and Cursor AI[3], enhancing developer efficiency. This raises an intriguing possibility: could individuals without optimization expertise leverage LLMs to enhance existing optimization algorithms?

Recent studies show LLMs can generate optimization algorithms from scratch for combinatorial problems [36, 41]. These approaches start with a simple prompt and improve through automated code generation. In contrast, using existing optimization algorithms as a foundation is a newer research direction, only recently explored by [29]. Implementing combinatorial optimization solutions requires significant expertise, particularly for achieving computational efficiency. LLMs can help in this process by identifying new algorithmic components such as heuristic information, applying methods from various fields, and writing more efficient code. This makes advanced algorithms accessible to non-specialists while helping experts implement innovative solutions more quickly.

In this paper, we present an in-depth study that showcases a methodology for using LLMs to enhance 10 classical optimization algorithms across various domains, including metaheuristics, reinforcement learning, deterministic heuristics, and exact methods, to solve the canonical Travelling Salesman Problem (TSP). Our findings demonstrate that LLMs can effectively suggest code improvements to existing algorithms (written in languages like Python) and incorporate modern techniques, all through a chatbot interface and without requiring prior expertise, by leveraging the extensive knowledge gained during their pre-training phase. For a schematic view of our framework on the example of a GA see Figure 1.

The paper unfolds as follows. In Section 2, we provide an overview of LLM advancements in combinatorial optimization, introduce the TSP problem, and briefly describe the 10 selected algorithms. Our methodology for enhancing existing optimization algorithms is detailed in Section 3. An exhaustive analysis of the 10 optimization algorithms improved through LLM application is presented in Section 4. Section 5 discusses implications and identifies directions for future research. We conclude by highlighting the key findings from our investigation.

2 Background

2.1 Large Language Models in Combinatorial Optimization

Large Language Models (LLMs) have recently shown promise in optimization tasks [40, 17, 12] by exploiting the vast knowledge gained during their pre-training phase. Beyond guiding the optimization process, LLMs excel at detecting patterns, identifying key features in problem instances, and refining search spaces. They have also shown to be able to generate new heuristics tailored to specific problems. Furthermore, LLMs offer valuable insights by explaining the results of optimization problems, making them versatile tools for both solving and interpreting complex tasks.

In this article, we focus on a specific type of optimization problem: combinatorial problems. These problems have unique properties that set them apart: many valid solutions (including the existence of equivalent or similar solutions), decomposability (some problems can be broken down into smaller, more manageable subproblems), constraint handling (a set of rules that define valid solutions), and search space structure (the presence of multiple local optima, requiring well-chosen search strategies to avoid getting trapped in suboptimal solutions), among others.

[2] `https://github.com/features/copilot`
[3] `https://www.cursor.com/`

Table 1: Overview of Selected Algorithms for Solving the Travelling Salesman Problem (TSP)

Algorithm	**Characteristics**	**Application to TSP**
Metaheuristic		
Ant Colony Optimization (ACO) [9]	Probabilistic, pheromone-based learning	Simulates ants' foraging behavior where solutions (routes) are constructed based on pheromone trails left by previous solutions.
Genetic Algorithm (GA) [26]	Population-based, crossover, mutation	Generates a population of routes and evolves them through selection, crossover, and mutation to find near-optimal solutions.
Adaptive Large Neighborhood Search (ALNS) [28]	Adaptive destruction and reconstruction	Iteratively destroys and reconstructs routes, dynamically adjusting strategies based on previous performance.
Tabu Search (TABU) [11]	Use of memory structures (tabu lists)	Iteratively modifies routes while keeping a list of features of previously visited solutions to prevent revisits.
Simulated Annealing (SA) [14]	Probabilistic, temperature-based search	Iteratively refines a route by accepting worse solutions with a decreasing probability to escape local optima.
Reinforcement Learning		
Q_Learning [37]	Value-based learning, exploration-exploitation trade-off	Learns an optimal routing policy by iteratively updating action-value functions based on rewards from different paths.
SARSA [37]	On-policy learning, continuous updates	Uses an on-policy approach to learning optimal routing strategies based on real-time interactions with the environment.
Deterministic Heuristic		
Christofides [7]	Guarantees 1.5-optimality, MST-based	Constructs a minimum spanning tree, finds perfect matching, and combines them to form a tour.
Convex Hull [10]	Geometric approach	Starts with a convex hull and incrementally inserts remaining points in a way that minimizes travel distance.
Exact		
Branch and Bound (BB) [23]	Systematic enumeration, pruning	Explores all possible solutions while pruning suboptimal paths to guarantee optimality.

As a result, researchers in this field must not only be knowledgeable about combinatorial optimization problems but also highly proficient in implementing optimization algorithms. Given the importance of computational efficiency, factors such as memory optimization, effective data structure management, minimizing unnecessary abstractions, and carefully selecting the right programming language play a crucial role in the design and development of these algorithms.

Recent research has leveraged LLMs as black-box collaborators of experts to develop novel strategies for tackling complex combinatorial optimization problems. These advancements can be categorized into three main areas:

- **Automatic generation of optimization algorithms** using LLMs. LLaMEA [36] is a groundbreaking framework that generates metaheuristic optimization algorithms in a bottom-up manner by combining LLMs with evolutionary computation techniques. Similarly, ReEvo [41] uses advanced reflective evolutionary approaches designed specifically for hyper-heuristic development, enabling the automatic discovery of new problem-solving strategies.
- **Analysis of optimization algorithms** through LLMs. Maddigan et al. [21] introduced GP4NLDR, a sophisticated web-based dashboard that combines genetic programming for non-linear dimensionality reduction with LLMs, significantly improving the interpretability of complex genetic programming trees. Additionally, [5] integrates LLMs into a web application that automates the analysis of search trajectory network (STN) visualizations, providing deeper insights into the behavioral patterns of various optimization algorithms.
- **LLMs as pattern recognition tools** for optimization problem structures. Going beyond code generation and analysis automation, [30] proposes an innovative approach where LLMs serve as powerful pattern recognition systems. The researchers used LLMs to detect critical structural patterns within combinatorial optimization problem instances, particularly related to graph metrics. These identified patterns were then transformed into heuristic information to be used by a metaheuristic, resulting in significant improvements in performance compared to state-of-the-art methods for solving complex social network optimization problems.

However, none of these approaches have explored how LLMs can make optimization algorithms more accessible to non-experts. Consider a scenario where a user with basic Python knowledge implements a Branch and Bound algorithm. By interacting with an LLM, they could receive tailored suggestions—like incorporating a problem-specific heuristic for initial bounds—potentially reducing computational time without requiring expertise in algorithmic optimization theory.

To introduce this novel research direction, the next two subsections present a classic combinatorial optimization problem along with ten traditional optimization algorithms, grouped into four distinct categories. Then, in the methodology section, Section 3, we demonstrate how LLMs can be leveraged to enhance these ten algorithms, improving both performance and efficiency while streamlining their implementations.

2.2 The Travelling Salesman Problem: A Brief Description

The Travelling Salesman Problem (TSP) is one of the most iconic problems in combinatorial optimization, serving as a foundational pillar in both Artificial Intelligence and Operations Research. Formally, let $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ be a set of n cities, and let $D : \mathcal{C} \times \mathcal{C} \to \mathbb{R}_{\geq 0}$ be a function that assigns a non-negative distance between each pair of cities. The objective of the TSP is to find a permutation π of the indices $\{1, 2, \ldots, n\}$ that minimizes the total travel distance of a closed tour, formally defined as:

$$\min_{\pi \in S_n} \left\{ \sum_{i=1}^{n-1} D(c_{\pi(i)}, c_{\pi(i+1)}) + D(c_{\pi(n)}, c_{\pi(1)}) \right\},$$

where S_n denotes the set of all permutations of $\{1, 2, \ldots, n\}$.

For this study, we selected the TSP because of the vast amount of implementations available online—public code repositories, books, and scientific articles—indicating that LLMs likely possess extensive knowledge of techniques for solving it [20].

2.3 Traditional Optimization Algorithms for the TSP

Algorithms for solving the TSP come in a broad variety of forms and approaches. We have chosen ten distinct optimization algorithms, summarized in Table 1. These algorithms are categorized into four groups: metaheuristic methods (stochastic, iterative optimization algorithms), reinforcement learning (policy-driven), deterministic heuristics, and an exact algorithm. These four algorithm categories can be briefly characterized as follows:

1. **Metaheuristics** [4] are versatile optimization methods that use heuristic and stochastic principles to search solution spaces. While not guaranteeing global optima, they efficiently find high-quality solutions.
2. **Reinforcement Learning** [39] is a machine learning paradigm where an agent learns optimal decision-making by interacting with an environment and maximizing cumulative rewards.
3. **Deterministic heuristics** [22] are among the most basic algorithms for combinatorial optimization. They generate generally one solution from scratch by making a myopic, deterministic decision at each step.
4. **Exact algorithms** [22] ensure optimality by exhaustively exploring the solution space.

The choice of algorithms from these four categories guarantees that our LLM-based improvement framework is tested across a diverse spectrum of methodologies, as the selected algorithms—though addressing the same problem (TSP)—vary fundamentally in their underlying principles.

2.4 Selected Implementations

To minimize the possibility of implementation errors in the ten algorithms for solving the TSP, and to ensure that these implementations are being utilized by the community, we employed `PyCombinatorial` [25].[4] This Python library, which includes a whole range of optimization algorithms for solving the TSP, has received over 100 stars on GitHub and was created by Valdecy Pereira. Each implementation is based on a standard algorithm variant, providing us with an excellent testing environment for attempting to improve them using LLMs. In the next section, we detail our methodology for improving optimization algorithms.

3 Methodology

3.1 Enhancing Traditional Optimization Algorithms with Large Language Models

We introduce a methodology based on LLM interactions to generate enhanced versions of the 10 before-mentioned algorithms, building upon and extending the approach proposed by [29] (see Figure 1). This process can be replicated using the chatbot available at the project URL.

Given the initial set $\{A_i \mid i = 1, \dots, 10\}$ of 10 original algorithm codes and an LLM, the algorithm improvement process can technically be stated as follows. First, a prompt P_i is generated based on our general prompt template T (shown in the second column of this page), the algorithm name N_i, the signature of the main function S_i, and the algorithm code A_i:

$$P_i = \text{Produce\_Prompt}(T, N_i, S_i, A_i), \quad i = 1, \dots, 10 \tag{1}$$

Then, the generated prompt P_i is executed by the LLM given a set of hyperparameters θ, resulting in a changed/updated implementation A'_i:

$$A'_i = \text{LLM}_{\text{execute}}(P_i, \theta), \quad i = 1, \dots, 10$$

To ensure correctness, each A'_i undergoes a validation process (see below). If an output fails validation, the refinement process is repeated iteratively until a valid version is obtained.

3.1.1 Code Validation

The validation of A'_i may fail for two reasons:

1. **Execution errors**: These lead to immediate failures during code runtime.

[4]`https://github.com/Valdecy/pyCombinatorial`

Table 2: Analysis of the Code Generation Process

Algorithm	**Claude-3.5-Sonnet** (temp = 1.0)		**Gemini-exp-1206** (temp = 2.0)		**Llama-3.3-70B** (temp = 1.0)		**GPT-O1** (temp = 1.0)		**DeepSeek-R1** (temp = 1.0)	
	Success		Success		Success		Success		Success	
	1st Try	# Attempts	1st Try	# Attempts	1st Try	# Attempts	1st Try	# Attempts	1st Try	# Attempts
ACO	✗	1	✓	-	✓	-	✓	-	✓	-
GA	✓	-	✗	1	✓	-	✗✗	3	✓	-
ALNS	✗	1	✓	-	✓	-	✓	-	✗✗	3
TABU	✓	-	✓	-	✓	-	✓	-	✓	-
SA	✓	-	✓	-	✓	-	✓	-	✓	-
Q_Learning	❎	-	✓	-	✓	-	✓	-	✓	-
SARSA	❎	-	✓	-	✓	-	✓	-	✓	-
Christofides	❎	-	✓	-	✓	-	✓	-	✓	-
Convex Hull	✗	1	✓	-	✓	-	✓	-	✓	-
Branch and Bound	✓	-	✗✗	4	✓	-	✓	-	✓	-

2. **Logical inconsistencies**: The algorithm executes without errors but produces invalid TSP solutions.

In the first case, an error message e is generated, and the LLM refines the code based on this feedback:

$$A'_i = \text{LLM}_{\text{execute}}(A'_i, \theta, e)$$

For the second case, where execution is error-free but the generated solutions are invalid, an explicit prompt requesting a correction is passed to the LLM, such as:

"The provided code generates invalid solutions; please verify and return a corrected version."

The refinement loop proceeds until a valid code A'_i is obtained. This process is carried out through an interactive conversation with an LLM-based chatbot. To increase the chances of generating a valid code with each retry, we begin with a high-temperature setting, which is then progressively lowered in each iteration of the process.

In LLMs, *temperature* controls the randomness of the model's output. A higher temperature (e.g., 1.0 or 2.0) makes the model's responses more diverse and less predictable, while a lower temperature (e.g., 0.2) makes the output more deterministic and stable.

Table 2 shows the results of this procedure for the five selected LLMs.[5] A green checkmark (✓) indicates that the first obtained A'_i was valid, while a red cross (✗) signifies that corrections were necessary due to either of the two code failures. A double red cross (✗✗) indicates that both failures occurred. Moreover, in the case of corrections, the number of necessary corrections is provided in a second column. Note that Table 2, below the LLM names, also indicates the initial temperature setting.

3.2 Prompt design

Since we consider 10 algorithm codes, 10 prompts must be generated. All these prompts are based on the same template (see below). The natural language formulation of the prompt focuses on two key aspects for updating the algorithm codes: (1) improving solution quality and (2) accelerating convergence. Moreover, we explicitly instruct the LLMs to explore state-of-the-art techniques to address these two objectives. Below, we provide the prompt template:

Prompt Template

```
You are an optimization algorithm expert.

I need to improve this {{algorithm name}} implementation for the travelling salesman problem
    (TSP) by incorporating state-of-the-art techniques. Focus on:

1. Finding better quality solutions
2. Faster convergence time
```

[5]The reasoning behind selecting these five LLMs (rather than others) will be explained in Section 4.

```
Requirements:
- Keep the main function signature: {{the signature of an the main function}}
- Include detailed docstrings explaining:
  * What improvement is implemented
  * How it enhances performance
  * Which state-of-the-art technique it is based on
- All explanations must be within docstrings, no additional text
- Check that there are no errors in the code

IMPORTANT:
- Return ONLY Python code
- Any explanation or discussion must be inside docstrings
- At the end, include a comment block listing unmodified functions from the original code

Current implementation:
{{algorithm code}}
```

The prompt template begins with *"You are an optimization algorithm expert,"* a technique known as "role prompting", which has been empirically shown to guide LLMs toward a specific behavior or specialization [15, 3]. By setting a clear context from the outset, it enhances both the relevance and quality of the model's response.

This approach aligns with "in-context learning" [8, 16, 31], combining an external context (the user-provided code in the prompt) with an internal one (the LLM's knowledge of various techniques to improve the TSP algorithm).

An important insight: supplying a complete algorithm code within the prompt works like a 'map,' guiding the model on how to update the code. The LLM must preserve the overall structure, making modifications only in the relevant sections without breaking the logic. Without this external context (the provided algorithm), the model's solution would likely be more constrained and less effective, as it would have to generate everything from scratch. In contrast, with an initial codebase, the model can focus on refining and improving specific areas rather than rebuilding the entire algorithm code from scratch. In other words, the provided code *influences* the update proposed by the LLM [8, 29].

As technically indicated already in Eq. 1, the prompt template receives three dynamic variables that are placed in the corresponding positions in the prompt template (enclosed within `{{ ... }}`):

- **Algorithm's name**: steers the LLM toward a specific context.
- **Main function's signature**: ensures the initial function's input arguments, output values, and name remain unchanged, preventing unintended modifications that could affect compatibility with the original code.
- **Algorithm code**: the optimization algorithm's original implementation in Python.

Additionally, we explicitly instruct the LLM to report the modifications it makes (*"Include detailed docstrings explaining: ..."*). This step is essential, as it enables us—as shown in Section 4—to understand why a particular LLM' code outperforms the original or another model's code.

4 Experimental evaluation

In this section, we present experiments with the 10 previously mentioned optimization algorithm codes taken from `PyCombinatorial`. We describe our setup for utilizing LLMs, the parameter tuning of the stochastic algorithms, the TSP datasets and evaluation metrics employed, and the comparative analysis of the results. We highlight key details from the generation process and conclude with an analysis concerning code complexity.

4.1 Setup

4.1.1 LLMs Environment

We selected five leading code-generation LLMs: Anthropic Claude-3.5-Sonnet [1], Google Gemini-exp-1206 [34], Meta Llama-3.3-70b [35], OpenAI GPT-O1 [24], and DeepSeek-R1 [33]. For simplicity, we will refer to the models

Table 3: Parameter values obtained by tuning with `irace`. Ranges show minimum/maximum values considered for tuning.

Algorithm	Parameter	Range	Original	Claude-3.5-Sonnet	Gemini-exp-1206	Llama-3.3-70B	GPT-O1	DeepSeek-R1
Metaheuristics								
ACO	m (ants)	(2, 20)	7	4	2	3	17	20
	α (alpha)	(1.0, 2.0)	1.34	1.72	1.67	1.46	1.72	1.22
	β (beta)	(1.0, 2.0)	1.59	1.24	1.98	1.97	1.93	1.55
	ρ (decay)	(0.01, 0.3)	0.24	0.12	0.24	0.29	0.06	0.05
GA	N (population size)	(5, 100)	97	14	97	84	55	58
	μ (mutation rate)	(0.01, 0.2)	0.02	0.04	0.16	0.16	0.01	0.13
	e (elite)	(1, 5)	4	2	5	2	3	5
ALNS	λ (removal fraction)	(0.05, 0.3)	0.27	0.05	0.26	0.29	0.22	0.29
	ρ (rho)	(0.01, 0.3)	0.27	0.25	0.04	0.2	0.27	0.02
TABU	T (tabu tenure)	(3, 30)	8	12	30	15	10	9
SA	T_0 (initial temperature)	(1, 50)	12	49	9	30	35	50
	T_f (final temperature)	(0.0001, 0.1)	0.0547	0.0464	0.074	0.056	0.0433	0.048
	α (cooling rate)	(0.8, 0.99)	0.9895	0.8732	0.8956	0.8131	0.9154	0.8777
Reinforcement Learning								
RL_QL	lr (learning rate)	(0.01, 0.5)	0.44	0.15	0.26	0.49	0.46	0.34
	df (decay factor)	(0.8, 0.99)	0.97	0.82	0.98	0.87	0.98	0.82
	ϵ (epsilon)	(0.01, 0.3)	0.09	0.28	0.03	0.24	0.21	0.13
	E (episodes)	(1000, 10000)	4266	1082	4906	2474	1294	1989
SARSA	lr (learning rate)	(0.01, 0.5)	0.04	0.36	0.49	0.19	0.41	0.29
	df (decay factor)	(0.8, 0.99)	0.86	0.91	0.80	0.88	0.83	0.87
	ϵ (epsilon)	(0.01, 0.3)	0.23	0.18	0.16	0.08	0.12	0.16
	E (episodes)	(100, 5000)	105	156	1850	137	124	1711

as Claude, Gemini, Llama, O1, and R1 throughout the remainder of this paper. Note that these LLMs rank among the top models in the LiveBench benchmark [38], which is immune to both test set contamination and the biases of LLM-based and human crowdsourced evaluations (**as of February 2025**). Using the OpenRouter API, we executed identical prompts across all models, enabling straightforward model switching for transparent experimentation. This produced 50 new algorithm codes which, combined with the 10 original algorithm codes from the `PyCombinatorial` framework, gave us 60 Python files ready for evaluation.

4.1.2 Parameter Tuning

Note that, while the heuristic methods and branch and bound are deterministic and parameter-less, the five selected metaheuristics and the two reinforcement learning approaches are probabilistic and require well-working values for their parameters. Thus, we tuned 42 algorithm codes: 35 new codes generated by the LLMs (5 per original algorithm code) and the 7 original algorithm codes. To ensure a fair comparison, we used `irace` [19], a well-established tool for parameter tuning. The best parameter configurations selected by `irace` are detailed in Table 3.

Parameter tuning in stochastic algorithms with parameters is essential, as suboptimal configurations can lead to poor performance regardless of the algorithm's inherent quality. For instance, in ACO, we tune key parameters—m (number of ants), α, β, and ρ (pheromone decay)—for all six code variants (five LLM-generated ones plus the original) to ensure they operate under optimal conditions for the TSP problem. This guarantees a fair evaluation, as each code variant is assessed using its best possible configuration.

4.2 Benchmark Datasets and Evaluation Metrics

To evaluate all algorithm codes, we use problem instances from the well-known TSPLib library [27]. We select 10 instances from the available ones, ranging from a small instance with 99 cities to a large one with 1084 cities.[6] This selection ensures a comparison across a diverse set of problem instance sizes.

As an evaluation metric, we used the objective function value of the best-found solution in all cases except for the Branch and Bound (BB) codes. This is because BB is an exact algorithm that, if given enough computation time, will always find an optimal solution. Therefore, we use runtime as the evaluation metric in the case of BB. Moreover, as the runtime of BB for the 10 selected problem instances is very high, we instead generate 10 random TSP instances with 10

[6]TSPLib names of the selected TSP instances: RAT99, BIER127, D198, A280, F417, ALI535, GR666, U724, PR1002, and VM1084.

to 15 cities for the evaluation of the BB codes. Interestingly, as we will see in the comparative analysis subsection, some LLM-generated versions of BB incorporate heuristic mechanisms during algorithm initialization, leading to significant improvements in runtime performance.

4.3 Experimental Design

The experiments were designed as follows:

- **Stochastic Algorithms**. Each of the metaheuristics and reinforcement learning codes is applied 30 times independently to each of the 10 problem instances. The output of each run is the best solution found. Performing 30 independent algorithm executions for each problem instance is a common practice in the optimization community to obtain a reliable estimate of the algorithm's performance. Moreover, the CPU time limit for each algorithm execution is set to the number of cities (in seconds) of the tackled problem instance. For example, the run-time limit for RAT99 is 99 seconds. This method, which aligns execution time with the instance size, is a common practice for comparing algorithms that solve the TSP.
- **Deterministic Heuristics**. Christofides and Convex Hull, since they are deterministic heuristics, always yield the same result for a given problem instance. Therefore, all corresponding codes are executed exactly once per instance.
- **Branch and Bound**. As previously mentioned, since Branch and Bound is an exact algorithm, the focus is on runtime rather than solution quality. All Branch and Bound codes are applied 30 times to each of the small problem instances specifically generated for the evaluation of the Branch and Bound codes.

All experiments, including parameter tuning, are conducted on a cluster equipped with Intel® Xeon® CPU 5670 processors (12 cores at 2.933 GHz) and 32 GB of RAM.

4.4 Comparative Analysis with Original Algorithm Codes

The comparative analysis between the LLM-updated algorithm codes and the original algorithm codes (referred to as 'original' from now on) is studied in the following in a separate way depending on the type of optimization algorithm.

4.4.1 Metaheuristics

The results of all metaheuristic codes are shown by means of boxplots in Figure 2. Note that the y-axes are shown in a logarithmic scale.

1. **ACO**: Apart from the first problem instance (RAT99) where the LLM-generated codes of Gemini, O1, and R1 perform similarly to the original code, in all other problem instances the LLM-generated codes of the mentioned three LLMs outperform the original code with statistical significance. It also appears that the LLM-generated codes of O1 and R1 are somewhat more robust than the original code, which can be seen in smaller boxes. In contrast, the code generated by Claude, apart from the first problem instance, performs always worse than the original code. Finally, the Llama-generated code generally performs similarly to the original code, with the exception of the first problem instance.
2. **GA**: The original code exhibits very low robustness for the first two problem instances, which can be seen by the large boxes. Generally, most codes (except for R1) are quickly trapped in local optima which they cannot escape. The R1-generated code clearly outperforms all others, including the original.
3. **ALNS**: Generally, the best-performing codes are those by Claude and Gemini, with a slight advantage for Gemini in the last two instances. Another noteworthy aspect is the low robustness of the O1-generated code in this case.
4. **TABU**: Like in the case of GA, also the TABU code generated by R1 significantly outperforms the remaining codes. Only the O1-generated code can compete for the smallest two problem instances. This suggests that R1 excels at generating efficient optimization algorithms.
5. **SA**: The Claude-generated codes clearly show the weakest performance here. In contrast, the R1-generated code again outperforms the remaining ones.

4.4.2 Reinforcement Learning (RL)

In Figure 3, the RL codes generated by the LLMs are compared with the original RL codes. The displayed results differ from the metaheuristics case presented before in two key aspects. First, some LLM-generated codes fail to produce a

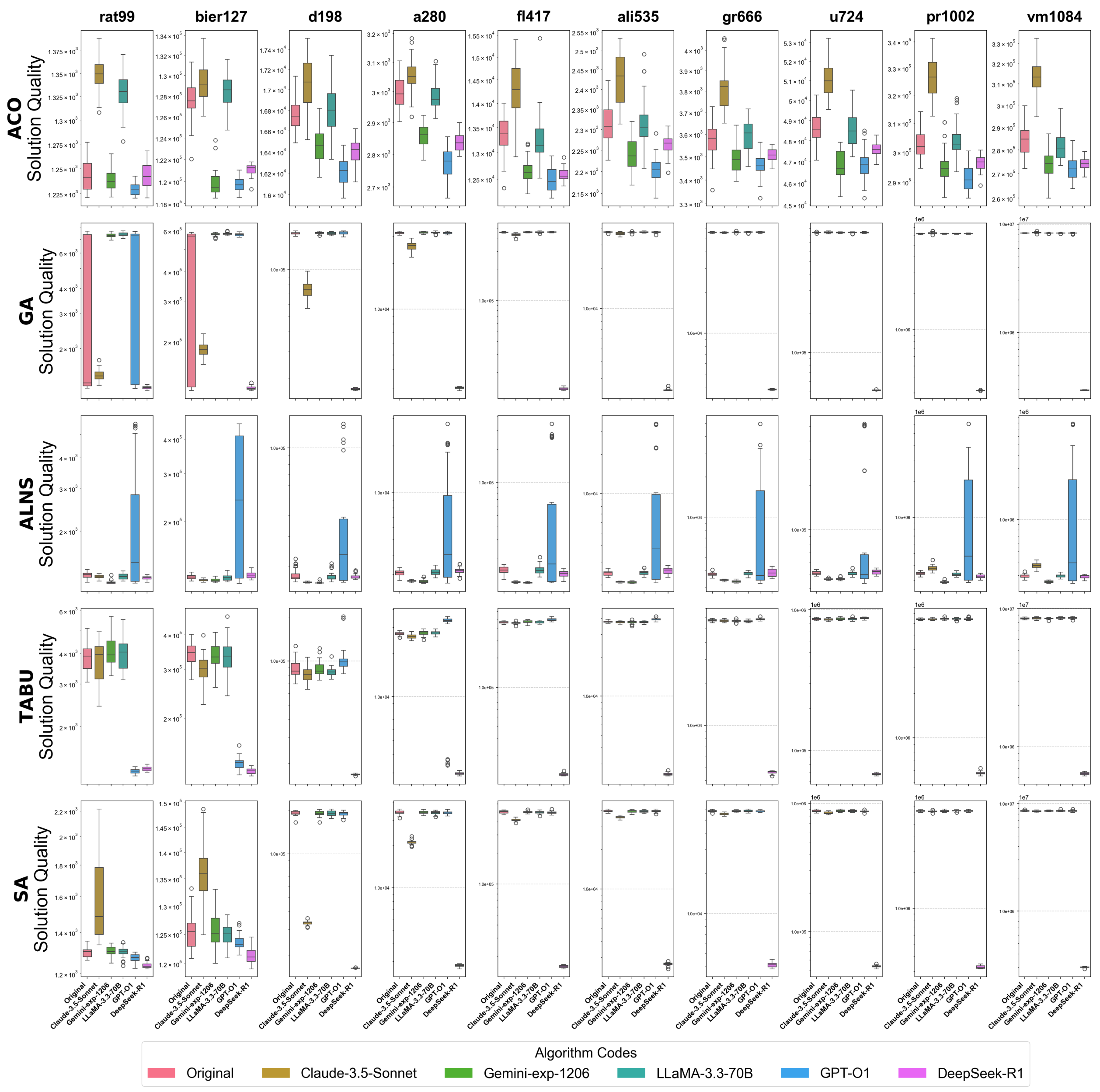


Figure 2: Comparison of the metaheuristic codes generated by the five LLMs with the original codes. Remember that the TSP is a minimization problem, that is, the lower the values, the better. The y-axes are shown in a logarithmic scale.

result for all problem instances within the time limit. These cases are marked as N/A. Second, especially in the case of Q_Learning the original code performs more competitively. We analyze these aspects below:

6. **Q_learning**: The original code is actually the best-performing one in this case. The Llama-generated code is the only one that achieves a nearly comparable performance—an unexpected result given Llama's poor performance in the case of the metaheuristics. In addition, the Claude-generated code outperforms the original one on the RAT99 and A280 instances.
7. **SARSA**: The general picture here aligns more with that observed in the case of the metaheuristics. The original code struggles (in comparison to some LLM-generated codes) as instance sizes grow larger. The R1-generated code fails to produce results for the largest three instances. The only code maintaining a stable and strong performance across all 10 instances is the O1-generated one.

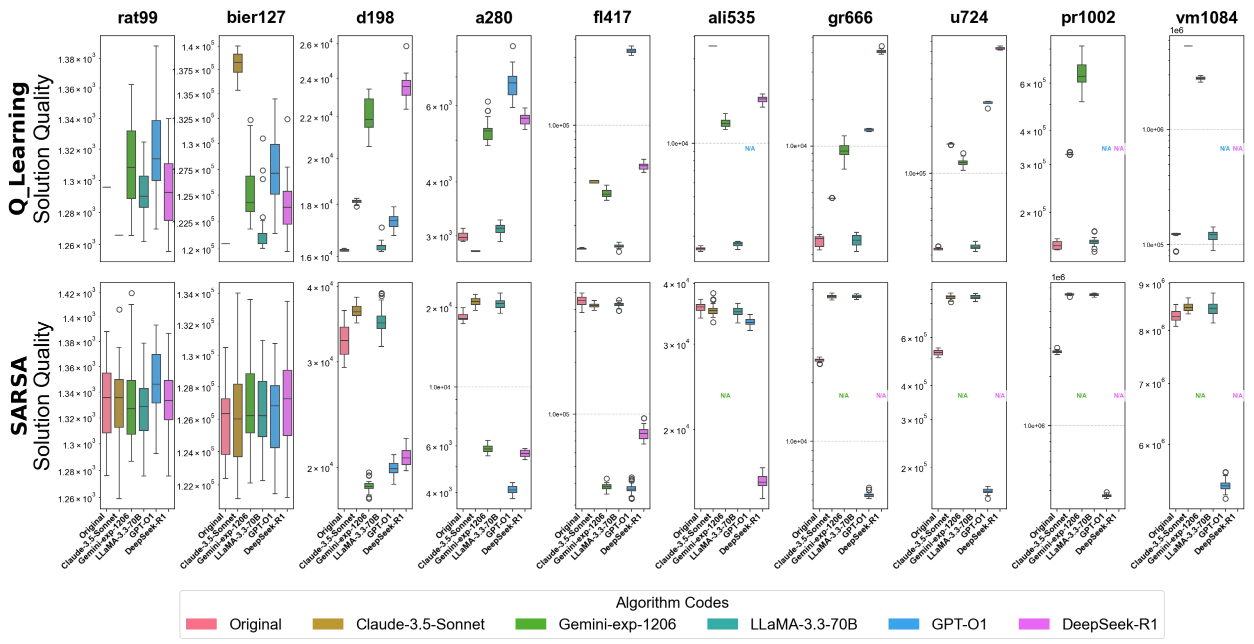


Figure 3: Comparison of the reinforcement learning (RL) codes generated by the five LLMs with the original codes. Remember that the TSP is a minimization problem, that is, the lower the values, the better. The y-axes is shown in logarithmic scale.

4.4.3 Deterministic/Heuristic

Remember that, as the chosen heuristics are deterministic, there is no need to analyze the distribution of their results over multiple runs. Therefore, in Figure 4, we simply compare the GAP of the results produced by the LLM-generated codes (in percent) relative to the results of the original codes. A positive value indicates that the respective LLM-generated code outperforms the original, while a negative value suggests the opposite.

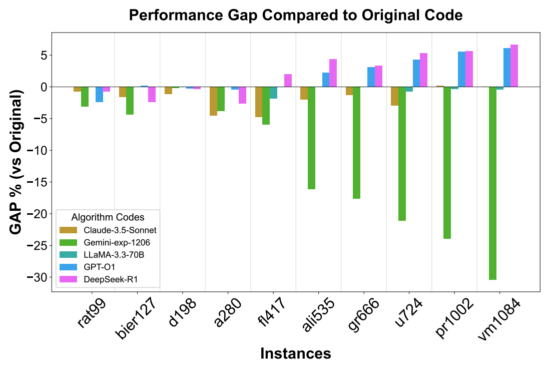
8. **Christofides** (see Figure 4 (a)): While the Llama-generated code produces very similar results to the original code over the whole instance range, the Gemini-generated code is (apart from instance D198) always inferior. Moreover, its relative performance decreases as instance size grows. Concerning O1 and R1, it can be stated that the performance of their codes is slightly inferior to the one of the original code for rather small problem instances. However, with growing instance size, they clearly outperform the original code.

9. **Convex Hull** (see Figure 4 (b)): In contrast to Christofides, the Convex Hull codes generated by O1 and R1 perform rather poorly. In fact, the best code for Convex Hull is the one generated by Gemini. This code has slight disadvantages for smaller instances but increasingly outperforms the original code with growing instance size. The code generated by Claude shows the opposite pattern. While it outperforms the original code for smaller problem instances, its performance strongly decreases with growing instance size.

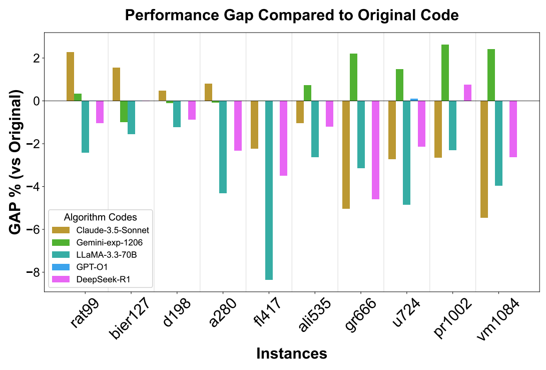
4.4.4 Exact Approach

As already mentioned in Section 4.3 (Experimental Design), in the context of the exact BB method, the comparison is based on computation time.

10. **Branch and Bound**: Figure 5 shows that the codes generated by O1 and R1 outperform both the original code. Notably, R1—an open-weight LLM—achieves the best performance, surpassing all proprietary models. In contrast to O1 and R1, the other LLM-generated codes perform worse than the original code.



(a) Christofides



(b) ConvexHull

Figure 4: Comparison of the deterministic heuristic codes generated by the five LLMs with the original codes. The bar plots show the performance gaps (in percent) relative to the original codes. Note that a positive value indicates that the LLM-generated code produces a better solution.

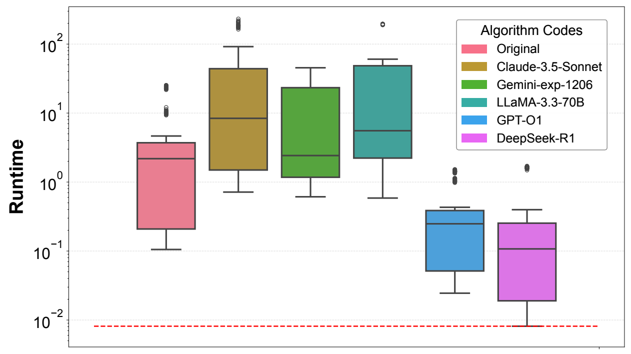


Figure 5: Comparison of the BB codes generated by the five chosen LLMs with the original BB code (in terms of computation time). Each code was applied 30 times, and the y-axis is shown in a logarithmic scale.

Summary: A notable conclusion is that LLMs can produce improved versions of baseline algorithms, resulting in performance improvements without necessitating specialized expertise in each algorithm. In the following subsection, we showcase examples of code improvements achieved, for example, by integrating more sophisticated algorithmic components.

4.5 Key Insights in Code Generation

Next, we explore why certain LLM-generated (or LLM-updated) codes outperform the original ones. Our focus was to understand if this was due to optimized data structures, for example, or due to adding different algorithmic components. In particular, we analyze four cases to address these questions on the basis of the LLM-generated codes.

4.5.1 Case 1: GA (R1-generated code)

The R1-generated version of GA features the following improvements, as stated by the model itself by means of a docstring in the code, as requested in the prompt.

```
Improvements:
1. Hybrid initialization with nearest neighbor heuristic
2. Rank-based fitness + tournament selection
3. Adaptive operator selection (OX, ER, BCR)
4. Memetic local search with stochastic 2-opt
5. Diversity preservation mechanisms
```

In particular, R1-generated GA is the only LLM-generated code that introduces a modification to the population initialization by incorporating the nearest neighbor heuristic. In contrast, both the original code and all other LLM-generated variants use the following initialization function:

```python
# Function: Initial Population
def initial_population(population_size, distance_matrix):
    population = []
    for i in range(0, population_size):
        seed = seed_function(distance_matrix)
        population.append(seed)
    return population
```

Instead, R1-generated GA features the following initialization that leads to an improved performance.[7]

```python
def initial_population(population_size, distance_matrix):
    """Initialize population with mix of random and heuristic solutions. Combines diversity (random) with quality (NN) for
     better exploration. Implements hybrid population initialization from modern metaheuristics."""
    population = []
    if population_size >= 5:  # Include 20% NN seeds
        for _ in range(max(1, population_size//5)):
            population.append(nearest_neighbor_seed(distance_matrix))

    ...
    return population
def nearest_neighbor_seed(distance_matrix):
    """Generate initial solution using Nearest Neighbor heuristic. Provides high-quality initial seeds to accelerate
     convergence. Based on constructive heuristic methods commonly used in TSP."""
    ...
```

In particular, the GA is initialized with 20% nearest neighbor solutions for population sizes of at least five individuals. This well-known TSP heuristic significantly speeds up convergence. In this way, R1 shows its ability to 'dig' into its knowledge base to choose an alternative population initialization method and implement it effectively.

4.5.2 Case 2: SA (R1-generated code)

Also in the case of SA, R1 identifies and utilizes two well-known mechanisms recognized for their efficiency in solving the TSP: (1) the Lundy-Mees adaptive cooling schedule for improved temperature control, introduced years after the original SA [18], and (2) the nearest neighbor heuristic for TSP. In the latter case, R1 integrates the nearest neighbor heuristic for the TSP in a way similar to what it did in the case of the GA, demonstrating a consistent pattern in leveraging effective initialization strategies.

4.5.3 Case 3: SARSA (O1-generated code)

The O1-generated SARSA code achieved the best results among the competitors. This is due to being the only code to make use of *Boltzmann Exploration* (see code below). Unfortunately, LLMs do not have the capacity to identify the exact source (book, scientific article, etc) from which the information about Boltzmann Exploration was extracted. However, after reviewing the code, it is likely that it was sourced from a 2017 paper (see [2]), which suggests a Boltzmann operator for SARSA applied to the TSP.

In fact, the code below shows that, unlike the original code, O1 not only applies a random operator to select the next unvisited city but also assigns a probability—derived from the `q_table` data structure—to this choice (line 9), making the selection more dynamic. Moreover, it avoids unnecessary abstractions (e.g., extra data structures) that could slow down the Python code.

```python
 1 ...
 2 while len(visited) < num_cities:
 3             unvisited = [city for city in range(num_cities)
 4                             if city not in visited]
 5             # Boltzmann exploration
 6             q_values = q_table[current_city, unvisited]
 7             exp_q = np.exp(q_values / temperature)
 8             probabilities = exp_q / np.sum(exp_q)
 9             next_city = np.random.choice(unvisited, p=probabilities)
10
11             reward = -distance_matrix_normalized[current_city, next_city]
12             visited.add(next_city)
13             route.append(next_city)
14             ...
15 ...
```

[7]Note that, in all Python code snippets shown in this paper, '...' indicates omitted parts that are not relevant.

4.5.4 Case 4: BB (R1-generated code)

When studying why the BB code of R1 was faster than the original code, first we noticed that, like in cases 4.5.1 and 4.5.2, R1 made use of the nearest neighbor heuristic for initialization. Moreover, R1 modified the `explore_path` function of BB by dynamically sorting the next candidates by edge weight to prioritize the cheapest/nearest extensions first. Both updates are not trivial. R1 notes the following in the code comments: *"Enhancements reduce unnecessary branching and accelerate convergence through early solution bias."* [8]

In addition, the code snippet below is not present in the original code. O1 introduces `current_node` and `candidates` efficiently, using slicing (lines 4 and 5) and sorting with a lambda function (line 6) to enhance path exploration in BB. This new array-based data structure is both efficient and implemented in a Pythonic style to improve performance.

```python
1 def explore_path(route, distance, distance_matrix, bound, weight, level, path, visited, min1_list, min2_list):
2     ...
3     current_node = path[level - 1]
4     candidates = [i for i in range(distance_matrix.shape[0])
5                   if distance_matrix[current_node, i] > 0 and not visited[i]]
6     candidates = sorted(candidates, key=lambda x: distance_matrix[current_node, x])
7     ...
```

4.6 Code complexity

In the previous subsection, we analyzed the LLM-generated codes in terms of their performance. But do these codes also offer better readability and reduced complexity in comparison to the original codes? To address this question, we evaluate their *cyclomatic complexity*—a metric that quantifies the number of independent paths through a program's source code. Through empirical research, Chen [6] demonstrated that a high cyclomatic complexity correlates with an increased bug prevalence. For our measurements, we employ the RADON library for Python.[9]

As shown in Table 4, the Claude-generated codes have the lowest average cyclomatic complexity (5.60 points), which improves code readability but, as shown before, comes at the cost of performance. The other models' codes and the original code have complexity scores between 6.84 (O1) and 7.51 (R1), which is still considered low and well-structured according to standard software engineering metrics. The values in the Risk Category column are taken from the documentation of the RADON library.[10]

Finally, Figure 6 reveals that there are cases—such as the R1-generates codes in the case of GA and Christofides, or the O1-generated code for SARSA—in which the LLM-generated codes not only outperform the original codes, but also decrease the cyclomatic complexity.

Table 4: Average Cyclomatic Complexity of the codes

	Algorithm Codes	**Average Complexity**	**Risk Category**
	Original	6.95	B (Low - Well structured)
LLMs	Claude-3.5-Sonnet	5.60	A (Low - Simple)
	Gemini-exp-1206	7.34	B (Low - Well structured)
	Llama-3.3-70b	7.38	B (Low - Well structured)
	GPT-O1	6.84	B (Low - Well structured)
	DeepSeek-R1	7.51	B (Low - Well structured)

[8] This can be seen in line 48 of file `bb_deepseek_r1.py` of our online repository URL.
[9] `https://pypi.org/project/radon/`.
[10] https://radon.readthedocs.io/en/latest/commandline.html#the-cc-command

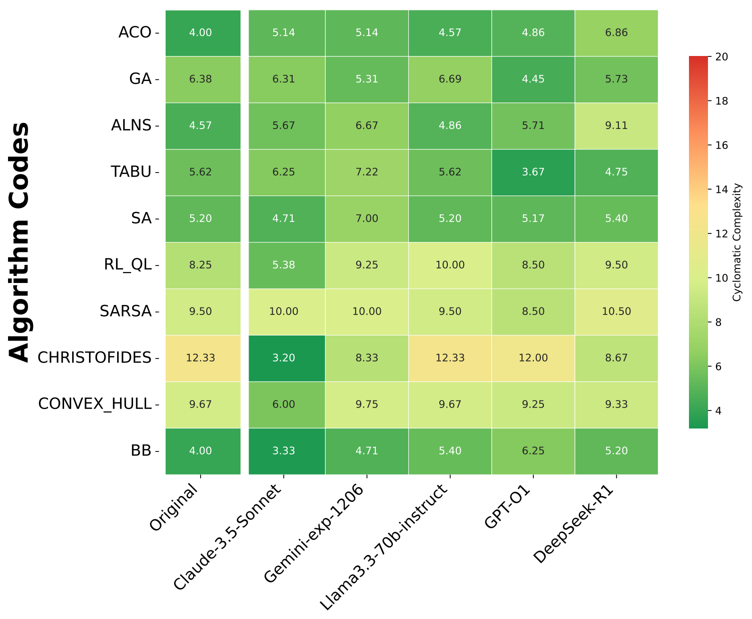


Figure 6: Cyclomatic Complexity

Summary: Based on all evaluations presented in this paper, we can state that among the five LLMs tested, R1 generally produces the best results, followed by O1. Gemini performed well in certain cases, such as ACO and ALNS; however, it underperformed in others, such as, for example, Christofides. Among all tested models, Claude showed the lowest performance.
In summary, *LLM-enhanced code versions clearly outperformed the original implementations in nine out of ten cases/algorithms*. Only for Q_Learning none of the models was able to improve the original code. In this case, Llama matched the performance of the original code.

5 Discussion

In [29], it was demonstrated that LLMs can be used to update and improve a specific, high-performance optimization algorithm. In this work, we successfully expanded this initial study to a classic combinatorial optimization problem, the TSP, using a wide range of optimization algorithms of different classes and natures.

5.1 Limitations and Challenges of LLM-Improved Optimization

- Since **LLMs do not have the capacity of precisely identifying the sources of code updates they make**, pinpointing which specific modifications contributed to performance improvements remains challenging [13]. While this requires manual code analysis and verification, exploring the question of *"where specifically does the generated code come from?"* presents a fascinating avenue for future research.
- **Not all LLMs produce correct implementations in the first attempt** (see Table 2), which requires the implementation of interactive feedback mechanisms—such as providing error codes or flagging invalid TSP solutions. While we addressed this through chatbot interactions, more automated validation and correction approaches warrant further investigation.

5.2 Directions for Future Research

- **How do LLMs perform for less known optimization problems?** We chose the TSP based on the intuition that numerous implementations are readily available online due to its widespread popularity. However, we do not yet know what would happen if we had used less-known combinatorial optimization problems. Such problems may present greater challenges for LLMs with limited training examples. One potential avenue of research to remedy this problem would be to develop fine-tuned LLMs specialized in optimization.

- The creation of **specialized benchmarks for baseline implementations of optimization algorithms** would be ideal. With this, we could analyze multiple optimization problems and algorithms; as new versions of LLMs emerge within short time frames, it becomes necessary to keep testing their capabilities in a structured way. As an example consider "reasoning models" that attempt to provide slower (test-time compute) but higher-quality responses [32].

6 Conclusion

Our research demonstrates that Large Language Models (LLMs) can significantly enhance the performance of 10 baseline optimization algorithms for a classic combinatorial problem: the Travelling Salesman Problem. These improvements not only resulted in higher solution quality but sometimes also in reduced computation times. By utilizing in-context prompting techniques, we were able to optimize existing code through better data structures, the incorporation of modern heuristics, and a reduction in code complexity. This approach is fully reproducible via a chatbot that is available on our project website.

Building on this foundation, future work will extend these advancements to less common optimization problems. Moreover, we plan to explore additional enhancements, such as leveraging LLMs to migrate to more efficient programming languages or integrating LLM-based agents to automate and continuously improve the enhancement process for existing algorithms.

Acknowledgements

The research presented in this paper was 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 Figure 1. Finally, thanks to Valdecy Pereira, creator of `PyCombinatorial`, for kindly answering our inquiries.

References

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

[2] Kavosh Asadi and Michael L. Littman. An Alternative Softmax Operator for Reinforcement Learning, 2017. URL `https://arxiv.org/abs/1612.05628`.

[3] Liam Barkley and Brink van der Merwe. Investigating the Role of Prompting and External Tools in Hallucination Rates of Large Language Models, 2024. URL `https://arxiv.org/abs/2410.19385`.

[4] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[5] Camilo Chacón Sartori, Christian Blum, and Gabriela Ochoa. Large Language Models for the Automated Analysis of Optimization Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '24, page 160–168, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704949. doi: 10.1145/3638529.3654086. URL `https://doi.org/10.1145/3638529.3654086`.

[6] Changqi Chen. An Empirical Investigation of Correlation between Code Complexity and Bugs, 2019. URL `https://arxiv.org/abs/1912.01142`.

[7] Nicos Christofides. Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem. *Operations Research Forum*, 3(1):20, Mar 2022. ISSN 2662-2556. doi: 10.1007/s43069-021-00101-z. URL `https://doi.org/10.1007/s43069-021-00101-z`.

[8] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui. A Survey on In-context Learning, 2024. URL `https://arxiv.org/abs/2301.00234`.

[9] M. Dorigo and L.M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997. doi: 10.1109/4235.585892.

[10] Samuel Eilon, C. D. T. Watson-Gandy, Nicos Christofides, and Richard de Neufville. Distribution Management-Mathematical Modelling and Practical Analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-4 (6):589–589, 1974. doi: 10.1109/TSMC.1974.4309370.

[11] Fred W. Glover. Tabu Search - Part I. *INFORMS J. Comput.*, 1:190–206, 1989. URL `https://api.semanticscholar.org/CorpusID:5617719`.

[12] Sen Huang, Kaixiang Yang, Sheng Qi, and Rui Wang. When Large Language Model Meets Optimization, 2024. URL https://arxiv.org/abs/2405.10098.

[13] Muhammad Khalifa, David Wadden, Emma Strubell, Honglak Lee, Lu Wang, Iz Beltagy, and Hao Peng. Source-Aware Training Enables Knowledge Attribution in Language Models, 2024. URL https://arxiv.org/abs/2404.01019.

[14] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598): 671–680, 2025/02/28/ 1983. URL http://www.jstor.org/stable/1690046. Full publication date: May 13, 1983.

[15] Aobo Kong, Shiwan Zhao, Hao Chen, Qicheng Li, Yong Qin, Ruiqi Sun, Xin Zhou, Enzhi Wang, and Xiaohang Dong. Better Zero-Shot Reasoning with Role-Play Prompting, 2024. URL https://arxiv.org/abs/2308.07702.

[16] Jia Li, Ge Li, Chongyang Tao, Jia Li, Huangzhao Zhang, Fang Liu, and Zhi Jin. Large language model-aware in-context learning for code generation. *arXiv [cs.SE]*, October 2023.

[17] Shengcai Liu, Caishun Chen, Xinghua Qu, Ke Tang, and Yew-Soon Ong. Large Language Models as Evolutionary Optimizers, 2024. URL https://arxiv.org/abs/2310.19046.

[18] M. Lundy and A. Mees. Convergence of an annealing algorithm. *Mathematical Programming*, 34(1):111–124, Jan 1986. ISSN 1436-4646. doi: 10.1007/BF01582166. URL https://doi.org/10.1007/BF01582166.

[19] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Thomas Stützle, and Mauro 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.

[20] Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. At Which Training Stage Does Code Data Help LLMs Reasoning?, 2023. URL https://arxiv.org/abs/2309.16298.

[21] Paula Maddigan, Andrew Lensen, and Bing Xue. Explaining Genetic Programming Trees using Large Language Models, 2024. URL https://arxiv.org/abs/2403.03397.

[22] R. Martí and G. Reinelt. *The Linear Ordering Problem: Exact and Heuristic Methods in Combinatorial Optimization*. Applied Mathematical Sciences. Springer Berlin Heidelberg, 2011.

[23] David R. Morrison, Sheldon H. Jacobson, Jason J. Sauppe, and Edward C. Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79–102, 2016. ISSN 1572-5286. doi: https://doi.org/10.1016/j.disopt.2016.01.005. URL https://www.sciencedirect.com/science/article/pii/S1572528616000062.

[24] OpenAI et al. GPT-4 Technical Report, 2024. URL https://arxiv.org/abs/2303.08774.

[25] V. Pereira. pyCombinatorial - A library to solve TSP (Travelling Salesman Problem) using Exact Algorithms, Heuristics and Metaheuristics. https://github.com/Valdecy/pyCombinatorial, 2022. Accessed: 2025-03-10.

[26] Jean-Yves Potvin. Genetic algorithms for the traveling salesman problem. *Annals of Operations Research*, 63 (3):337–370, Jun 1996. ISSN 1572-9338. doi: 10.1007/BF02125403. URL https://doi.org/10.1007/BF02125403.

[27] Gerhard Reinelt. TSPLIB - A Traveling Salesman Problem Library. *INFORMS J. Comput.*, 3(4):376–384, 1991. URL http://dblp.uni-trier.de/db/journals/informs/informs3.html#Reinelt91.

[28] Stefan Ropke and David Pisinger. An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows. *Transportation Science*, 40(4):455–472, 2025/02/28/ 2006. URL http://www.jstor.org/stable/25769321. Full publication date: November 2006.

[29] Camilo Chacón Sartori and Christian Blum. Improving Existing Optimization Algorithms with LLMs, 2025. URL https://arxiv.org/abs/2502.08298.

[30] Camilo Chacón Sartori, Christian Blum, Filippo Bistaffa, and Guillem 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.

[31] Sander Schulhoff, Michael Ilie, and et al. The Prompt Report: A Systematic Survey of Prompt Engineering Techniques, 2025. URL https://arxiv.org/abs/2406.06608.

[32] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters, 2024. URL https://arxiv.org/abs/2408.03314.

[33] DeepSeek-AI Team. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning, 2025. URL `https://arxiv.org/abs/2501.12948`.

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

[35] Meta Team. The Llama 3 Herd of Models, 2024. URL `https://arxiv.org/abs/2407.21783`.

[36] Niki van Stein and Thomas Bäck. LLaMEA: A Large Language Model Evolutionary Algorithm for Automatically Generating Metaheuristics. *IEEE Transactions on Evolutionary Computation*, pages 1–1, 2024. doi: 10.1109/TEVC.2024.3497793.

[37] Jiaying Wang, Chenglong Xiao, Shanshan Wang, and Yaqi Ruan. Reinforcement learning for the traveling salesman problem: Performance comparison of three algorithms. *The Journal of Engineering*, 2023. URL `https://api.semanticscholar.org/CorpusID:261504600`.

[38] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Siddartha Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. LiveBench: A Challenging, Contamination-Free LLM Benchmark, 2024. URL `https://arxiv.org/abs/2406.19314`.

[39] Marco Wiering and Martijn van Otterlo. *Reinforcement Learning: State-of-the-Art*. Springer Publishing Company, Incorporated, 2014.

[40] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large Language Models as Optimizers, 2024. URL `https://arxiv.org/abs/2309.03409`.

[41] Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large Language Models as Hyper-Heuristics with Reflective Evolution, 2024. URL `https://arxiv.org/abs/2402.01145`.