Enhancing Algorithms with LLMs: A Case Study

Yashar Talebirad[1][0009−0002−5792−6464], Amirhossein Nadiri[2][0000−0003−4112−2138], Osmar R. Zaïane[1][0000−0002−0060−5988], and Christine Largeron[3][0000−0003−1059−4095]

[1] Alberta Machine Intelligence Institute, *University of Alberta*, Edmonton, Canada
[2] *York University*, Toronto, Canada
[3] *Hubert Curien Laboratory*, *Université Jean Monnet*, Saint-Etienne, France
talebira@ualberta.ca, anadiri@yorku.ca, zaiane@ualberta.ca, largeron@univ-st-etienne.fr

Abstract. This paper explores the potential of Large Language Models (LLMs) to enhance community detection algorithms, with a focus on the SIWO (Strong In, Weak Out) algorithm. By integrating LLMs into the algorithm development process, focusing on their multi-disciplinary knowledge as a potential advantage over human expertise, we demonstrate how LLMs (with the possible oversight of a human expert) can generate innovative algorithm modifications that lead to enhanced performance. Our study reveals substantial reductions in execution times by more than 50% for SIWO when utilizing these modifications. Motivated by these promising results within the domain of Social Networks Analysis, we briefly introduce the Algorithmic Enhancement Framework (*AEF*), designed to extend these methodologies for broader algorithm enhancement. AEF employs the collaborative use of LLMs to generate and refine solutions iteratively, offering a novel foundational approach for incorporating LLM capabilities for the refinement of algorithms across a broad range of computational domains.

Keywords: LLMs · Algorithm Optimization · Community Mining

1 Introduction

Large Language Models (LLMs) like GPT [23], Llama [24], and PaLM [26] have transformed interactions with data and language through their ability to interpret and generate human-like text. These models are applied in creative writing, technical problem-solving, robot task planning, and more, marking significant advancements in artificial intelligence [11,20,25]. AI's role in algorithm enhancement is expanding, as demonstrated by studies like AlphaDev [8], which show AI can develop more efficient search algorithms. The creative problem-solving abilities of LLMs [3,4,18] have opened new avenues for algorithm improvement, especially when combined with evolutionary strategies [5,7].

However, current applications of LLMs for algorithm optimization often focus on small, predefined solution spaces or structured code generation, limiting their

potential for cross-domain creativity. Complex algorithms, particularly in interdisciplinary fields, require solutions that integrate ideas from various domains. Social Networks Analysis (SNA) is a promising area for leveraging LLMs' interdisciplinary strengths, as algorithms for community detection and search algorithms can utilize domain-specific heuristics to adapt to different network types.

In this paper, we explore enhancing community detection algorithms using LLMs. Specifically, we demonstrate how ChatGPT can suggest innovative variations to the SIWO (**S**trong **I**n, **W**eak **O**ut) community mining algorithm [9]. Our results show that LLMs can improve algorithm performance by generating creative enhancements, leading to faster execution times. Building on this success, we propose the Algorithmic Enhancement Framework (*AEF*), a generalized approach for broader algorithm improvement. AEF leverages LLMs' extensive knowledge and interactive capabilities to iteratively develop and evaluate innovative solutions, supported by the Knowledge Storage Module (KSM), Idea Synthesis Module (ISM), and Code Synthesis Module (CSM). KSM serves as a dynamic knowledge base and version control system, enabling ISM to generate and rank improvement ideas while CSM translates these ideas into executable code for evaluation. AEF also utilizes multiple LLM agents in a collaborative setting to enhance idea synthesis, a method demonstrated to improve performance in areas such as code generation and problem-solving [1, 28, 29].

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 provides background on SIWO and how ChatGPT can enhance it. Section 4 introduces our framework and its potential implementations. Finally, Section 5 summarizes our work and discusses future directions.

2 Related Work

Large Language Models: Recent advancements in Large Language Models (LLMs) have showcased their versatility and interpretive capabilities across various domains. For instance, GPT-4 [23] excels in generating human-like text and code, while LLaMA 2 [24] offers an efficient and scalable training approach, which enables broader access within the research community. PaLM 2 [26] also showcases strong multilingual, reasoning, and coding skills.

Studies suggest LLMs contribute to early artificial general intelligence through creative problem-solving and adaptive learning [25, 27]. Chen and Ding [3] investigate LLMs' creative thinking through the divergent association task, and Franceschelli [4] analyzes their creativity in writing. Guo et al. [18] evaluate LLMs' optimization capabilities, introducing performance metrics and highlighting the benefits of specific prompting. Yang et al. [16] develop Optimization by Prompting (OPRO), demonstrating LLMs' adaptability and efficiency in solving various optimization tasks.

Automatic Algorithm Design: Automatic algorithm design aims to automate the creation of algorithms with minimal human intervention. This field primarily utilizes meta-heuristics, a high-level problem-independent algorithmic framework, to guide the search process for finding efficient and effective solutions

to complex optimization problems. Even though they do not guarantee an optimal solution, they are effective for a wide range of problems [6]. Burke et al. [14] explore hyper-heuristics for selecting and generating heuristics, while Meng and Qu [12] introduce AutoGCOP for automating local search algorithm design. Zhao et al. [13] conduct a survey on meta-heuristic algorithm creation, and Mei et al. [21] discuss genetic programming for enhancing model interpretability.

The integration of LLMs into algorithm design has led to innovative evolutionary methods. Lehman et al. [17] combine genetic programming with LLMs to evolve programs using natural language prompts. Meyerson et al. [19] introduce LMX, leveraging LLMs' in-context learning for evolutionary algorithms across different domains. Bradley et al. [22] present OpenELM, a Python library that uses LLMs for generating variations and evaluating fitness in evolutionary algorithms. Liu et al. [5, 15] propose the AEL framework, which outperforms hand-crafted algorithms for the traveling salesman problem. Romera et al. [7] introduce FunSearch, merging LLMs with evolutionary algorithms to find solutions for complex problems. Compared to the current approaches, our approach differs by focusing on cross-domain knowledge transfer, multi-agent collaboration, and explainability, as detailed in Section 4.1.

3 Case Study: Enhancing the SIWO Algorithm

Community detection identifies all communities in a network, while community search finds dense subgraphs around specific nodes. We use SIWO [9], a state-of-the-art algorithm for both tasks, as a case study to demonstrate improvements with ChatGPT[4]. Although other language models could be used, we chose GPT for its availability and proven collaboration capabilities [32].

SIWO (**S**trong **I**n, **W**eak **O**ut) was introduced by Gharaghooshi et al. [10] for global community detection and has since evolved into a capable local search method for weighted, unweighted, and uncertain graphs [9, 31]. Global community detection (SIWO+) is achieved by running the local SIWO multiple times iteratively, with different starting nodes in the unexplored regions. In this section, we explain SIWO and its enhancement using LLM assistance.

3.1 Preliminaries

We use the concept of "Edge Strength" from Gharaghooshi et al. [10], where each edge has a strength between -1 and 1, with higher values indicating stronger connections. The strength is based on the support value, which counts the number of shared neighbors (triangles) between two nodes:

Support: For a graph $G = (V, E)$, the support of edge $e_{u,v}$ is the number of mutual neighbors of u and v:

$$sup(u, v) = |\{w \in V \mid e_{u,w}, e_{v,w} \in E\}| \tag{1}$$

[4] https://chat.openai.com/ Version: GPT-4o, Apr 10, 2024

Here, $sup(u, v)$ is the **size** of the set of nodes that are adjacent to both u and v.

Link Strength: The strength $s_{u,v}$ of the link between u and v is:

$$s_{u,v} = sup(u, v) \left(\frac{1}{sup_{u,\max}} + \frac{1}{sup_{v,\max}} \right) - 1 \tag{2}$$

where $sup_{u,\max} = \max_{w \in V_N(u)} \{sup(u, w)\}$ and $V_N(u)$ are the neighbors of u. The SIWO algorithm [9] is based on these definitions, which we will now outline.

3.2 Algorithm Overview

SIWO uses a greedy approach to expand communities from a query node. It starts by calculating support and strength for connections between the community and its shell set (neighbors of C not in the community). The strongest connected node from the shell is then added, and the shell is updated with new neighbors of the added node (excluding current community members). The algorithm stops when no new nodes can be added (i.e. the best addition does not increase the community's overall strength). We will now explore how we enhance SIWO, by using ChatGPT to propose algorithmic improvements, aiming for better performance and faster execution.

3.3 Enhancements

The enhancement of SIWO was performed using a custom GPT instance[5], named `SIWO-GPT`, which we connected to a custom knowledge base containing the SIWO paper [9] and its implementation[6]. We instructed `SIWO-GPT` to identify optimization and innovation areas and propose novel enhancements by providing a detailed algorithm outline in its system prompt, in addition to descriptions of the added background material. This Retrieval-Augmented Generation (RAG) [30]-like setup enabled `SIWO-GPT` to generate informed, context-aware improvement suggestions without the costs and time associated with manual model fine-tuning. Subsequently, `SIWO-GPT` proposed several potential enhancements. The input prompts and summarized outputs are shown in Figure 1. The complete chat history, including all prompts and responses, can be found here[7].

Based on expert evaluation, we pursued the "Parallel Processing for Scalability" avenue, which after a discussion with `SIWO-GPT`, was turned into a tangible, refined idea ready for implementation, which focused on using a matrix multiplication technique to optimize SIWO's support and strength calculations (see Figure 2 for Iteration 1). Even though several innovative ideas inspired by cross-domain knowledge were suggested (e.g. quantum computing techniques and methods from statistical physics), these ideas were not explored further in our current work due to time constraints and the scope of this paper. However,

[5] https://openai.com/blog/introducing-gpts
[6] https://github.com/talebirad/SIWO
[7] https://gist.github.com/talebirad/700d80e6627389665a3f5aa59f040ea8

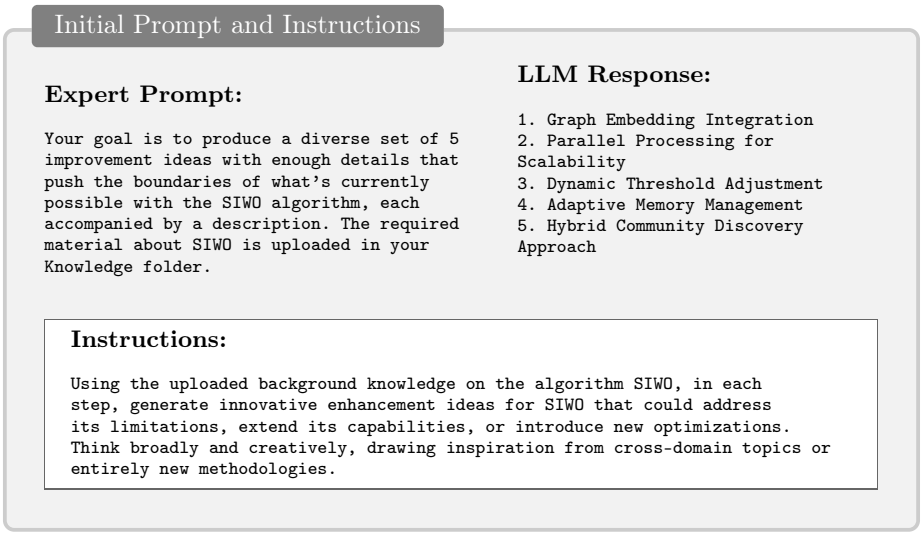


Fig. 1. Initial Prompts and Instructions for Idea Generation. The responses are summarized to include only the idea titles.

they represent promising avenues for future research (Refer to Section 5). Below, we present `SIWO-GPT`'s detailed explanation of optimizing SIWO's calculations using the matrix multiplication technique.

"Using matrix multiplication to compute support and strength values involves leveraging the adjacency matrix A. The process can be outlined as follows:

Adjacency Matrix: For a graph G, the adjacency matrix A is a square matrix where $A_{ij} = 1$ if there is an edge between nodes i and j, and 0 otherwise.

Matrix Multiplication for Support Calculation: The support $sup(u, v)$ for an edge (u, v) can be calculated by identifying the number of mutual neighbors between nodes u and v. This can be achieved by computing the square of the adjacency matrix A^2. The element (i, j) in the resultant matrix A^2 represents the number of paths of length 2 between nodes i and j, which corresponds to the number of mutual neighbors (or the support values).

Computing Strength Values: Once the support values are obtained, the strength $s_{u,v}$ can be computed using the provided formula. The maximum support value $sup_{u,max}$ and $sup_{v,max}$ for each node can also be determined by examining the support values involving node u and v, respectively."[8]

The matrix techniques for support and strength calculations were validated for mathematical accuracy and feasibility through consultations with another GPT instance and a human expert. After implementing the suggested technique with the help of a custom GPT with coding capabilities, we generated random

[8] This description is generated by `SIWO-GPT`.

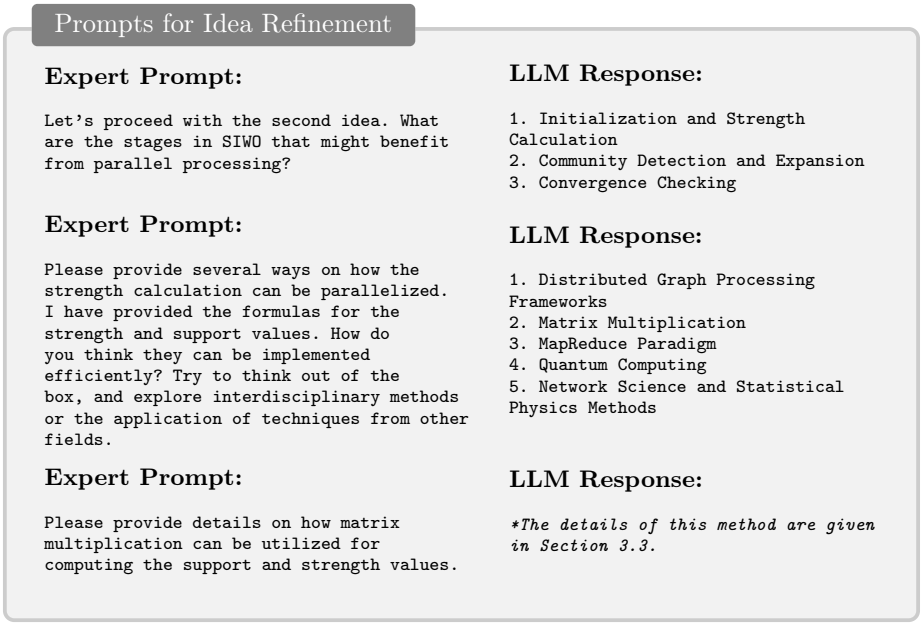


Fig. 2. Prompts for Idea Refinement and Parallelization. The responses are summarized to include only the idea titles.

Erdős-Rényi graphs to ensure that the calculated values and the identified communities matched those of the original SIWO implementation.

In the original paper [9], SIWO's community detection quality was thoroughly evaluated using real and synthetic datasets with ground-truth communities, which demonstrated superior memory management in larger graphs. In medium-sized graphs, memory usage was not a significant concern compared to execution time. Therefore, in this study, we focused on improving execution time and computational efficiency without re-evaluating metrics like modularity and Normalized Mutual Information (NMI). This led us to choose Erdős-Rényi graphs to evaluate runtime across different sizes and densities, because of their randomness and controllable density via the parameter p. Large dense graphs were excluded due to memory errors encountered during creation. For each configuration, we sampled 5000 edges, except for the small sparse graph, which had fewer edges, resulting in a sample size of 1000 edges. The graph specifications were as follows:

- **Small, sparse**: 1000 nodes, Erdős-Rényi $p = 0.01$.
- **Small, dense**: 1000 nodes, Erdős-Rényi $p = 0.5$.
- **Medium, sparse**: 10000 nodes, Erdős-Rényi $p = 0.01$.
- **Medium, dense**: 10000 nodes, Erdős-Rényi $p = 0.5$.
- **Large, sparse**: 25000 nodes, Erdős-Rényi $p = 0.01$.

We evaluated the generated code on these graph configurations to assess correctness and performance improvements over the original SIWO method. Evaluation and graph generation were conducted in Google Colaboratory[9] due to memory and time limitations of ChatGPT's Code Interpreter. Both implementations used a caching mechanism as suggested by Zafarmand et al. [9] to ensure a fair comparison. The evaluation results are shown in the second row (Iteration 1) of Table 1, with experiments exceeding 20 minutes or encountering memory issues being labeled as *TLE* and *MLE*, respectively. Results indicate that the matrix-based approach outperforms the original method on dense graphs but falls short in sparse graphs. Furthermore, the original SIWO algorithm hit a time limit in the medium-dense graph despite sufficient memory, as it needed to calculate support and strength values for many edges. Dense matrix multiplication likely caused excessive memory usage in the large-sparse graph, leading to termination.

Motivated by these findings, we included the evaluation results in the prompt for the second iteration to encourage the generation of ideas that further optimize the existing matrix approach or entirely new ideas. `SIWO-GPT` then generated several ideas, including graph simplification, incremental updates, and notably, sparse matrix optimization techniques to enhance computational performance in low-density graphs. The input prompt and summarized outputs are shown in Figure 3. We then generated error-free code for sparse matrix operations with LLM assistance, followed by validation and performance evaluation. The results are presented in the third row (Iteration 2) of Table 1.

Second Iteration Prompts for Idea Generation and Optimization

Expert Prompt:

```
Here are the results after evaluating the
performance of the matrix-based approach
on several graphs:
"Results are given here"

And these are the specifications of
the graphs:
"Graph specs are given here"

Drawing inspiration from these results
in each category, how do you think
we can improve the performance of the
implementation of the new matrix-based
approach?
```

LLM Response:

```
1. Optimization for Sparse Graphs
2. Hybrid Approach
3. Decomposition and Recombination
4. Graph Simplification
5. Incremental Updates
```

Fig. 3. Second Iteration Prompts for Idea Generation and Optimization. The responses are summarized to include only the idea titles.

[9] https://colab.research.google.com/

Approach	Small Sparse	Medium Sparse	Large Sparse	Small Dense	Medium Dense
Original	**0.031s**	10.36s	96.82s	5.79s	*TLE*
LLM Idea (Iteration 1)	0.085s	70.90s	*MLE*	**0.45s**	**180.58s**
LLM Idea (Iteration 2)	0.08s	**5.09s**	**55.91s**	5.09s	*TLE*
LLM Idea (Iteration 3)	0.08s	**5.09s**	**55.91s**	**0.45s**	**180.58s**

Table 1. Results for evaluation of LLM ideas until iteration 3

The results show that the modified approach is more effective for the sparse graph, running within time and memory limits and halving the runtime of the original method. However, it struggles with the dense graph by exceeding the 20-minute threshold and resulting in a TLE error, likely due to overhead in managing non-sparse matrices and lower efficiency compared to dense matrix operations in high-density graphs. Furthermore, in the small-sparse graph, the runtime differences between the original and optimized methods were negligible.

We then initiated a third iteration, prompting `SIWO-GPT` to generate ideas for further enhancing the algorithm by combining previous approaches. Among several suggestions, we selected the "Adaptive Algorithm Selection Based on Graph Density" and proceeded to integrate the codes from the previous two iterations into a new algorithm that first calculates the input graph density, and then selects the appropriate optimization technique (sparse or dense matrix multiplication). The code then underwent another cycle of validation and evaluation, with results presented in the last row of Table 1.

After this last iteration, we concluded this enhancement cycle for SIWO, resulting in a new variation that outperforms the original method in sparse and dense graphs that are not very large. Note that during idea generation, `SIWO-GPT` proposed several innovative enhancements inspired by cross-domain knowledge, such as quantum computing techniques and methods from network science and statistical physics. Although not explored further due to time constraints, these ideas represent promising avenues for future research and demonstrate the potential of LLMs to contribute creative solutions from diverse fields.

4 Towards a General Framework

Building on our initial exploration, we propose the Algorithmic Enhancement Framework (AEF), a structured and iterative approach to improve computer science algorithms using LLMs. We focus on a general implementation that leverages RAG and LLM capabilities without being tied to a specific model.

AEF consists of three main modules: the Knowledge Storage Module (KSM), the Idea Synthesis Module (ISM), and the Code Synthesis Module (CSM). KSM is a dynamic knowledge base used for storing, retrieving, and updating all data required in the enhancement process. Each cycle begins with ISM retrieving data from KSM to generate and rank ideas for improving the algorithm based on their potential impact on target metrics. These ideas, along with their scores,

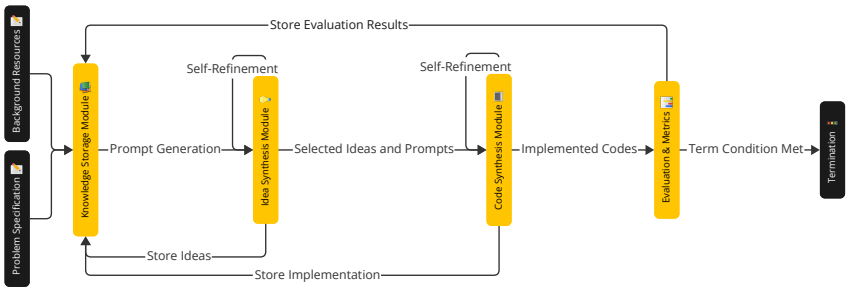


Fig. 4. Workflow of the Algorithmic Enhancement Framework (AEF). Starting with problem specifications and background resources, the framework iteratively synthesizes ideas, implements them as code, evaluates the results, and updates the knowledge base until the termination condition is met.

are refined and stored in KSM. CSM then retrieves the selected ideas from KSM, generates executable code, and stores the implementations back in KSM. The generated code is evaluated, and the results, including updated impact scores, are saved in KSM for use in the next iteration. An overview of the framework is demonstrated in Figure 4. The iterations in our framework mirror methodologies like the *scientific method*[10] and *Design Science Research* [35], with their emphasis on iteratively improving hypotheses based on empirical evidence. Using the empirical results to form more realistic ideas based on their impacts also draws inspiration from a concept called in-context or few-shot learning [2]. We will now go over each module in detail.

- **Knowledge Storage Module (KSM):** KSM is a dynamic knowledge base and version control system that provides all modules access to external knowledge not present in their training data. It functions as a RAG system, which involves embedding documents or data into a vector space using embedding models and storing them in a vector database for efficient retrieval. KSM is represented as a structured database $\mathcal{K} = \{\mathcal{K}_{\text{alg}}, \mathcal{K}_{\text{bg}}, \mathcal{K}_{\text{ideas}}\}$ and contains:

1) Algorithm Specifications ($\mathcal{K}_{\text{alg}}$): Includes detailed descriptions of the algorithm, including pseudo-code, and mathematical formulations.

2) Background Resources ($\mathcal{K}_{\text{bg}}$): Includes relevant literature, code repositories, datasets, and other external resources that provide context.

3) Ideas ($\mathcal{K}_{\text{ideas}}$): Includes a collection of ideas for each iteration t, notated as $I^{(t)} = \{I_1^{(t)}, I_2^{(t)}, \ldots, I_{k_t}^{(t)}\}$. Each idea $I_j^{(t)}$ is associated with a description, scoring vector $\mathbf{s}_j$, overall score S_j, status (e.g., implemented, evaluated, discarded), and a set of implementations ($\mathcal{C}_j = \{C_{j1}, C_{j2}, \ldots, C_{jm}\}$). Each implementation C_{jk} includes version information, the code, and the evaluation results vector (E_{jk}) based on the metrics. The framework is initialized by assembling KSM, which involved preparing $\mathcal{K}_{\text{alg}}$ and $\mathcal{K}_{\text{bg}}$.

[10] https://en.wikipedia.org/wiki/Scientific_method

• **Idea Synthesis Module (ISM):** ISM generates and ranks improvement ideas for the algorithm through an iterative process that combines generative and evolutionary strategies, supported by the knowledge base $\mathcal{K}$. Initially, ISM defines a set of criteria $C = \{c_1, c_2, \ldots, c_m\}$ with associated weights w_i indicating their importance, which is used in the scoring mechanism. The criteria and their weights remain consistent across iterations and may include feasibility, expected performance impact, and the novelty of each suggestion.

At iteration t, ISM utilizes chain-of-thought prompts and data from KSM to prompt LLMs to generate and refine a set of ideas $I^{(t)} = \{I_1^{(t)}, I_2^{(t)}, \ldots, I_{k_t}^{(t)}\}$. For each idea $I_j^{(t)}$, ISM may perform multiple internal refinement iterations l (up to a maximum $l_{\max}$) involving back-and-forth interactions between LLMs (or between human and AI), using techniques like chain-of-thought prompting and multi-agent collaboration, to ensure that the idea is tangible and ready to implement. This process may include generating detailed prompts that will be used by the LLMs in CSM. These ideas can be either:

1) Novel ideas that are unconventional and generated through the interdisciplinary knowledge of LLMs, with the potential to enhance the target metrics.

2) Refined ideas created by improving existing concepts from prior iterations based on their evaluations, similar to mutation in evolutionary algorithms. Vision-enabled LLMs can be used to incorporate visual information of the evaluation results from previous iterations.

3) Combined ideas that come from merging successful strategies from prior iterations into more robust solutions, akin to crossover in evolutionary algorithms. This may include developing adaptive algorithms that select optimization strategies based on input characteristics (e.g., graph density in the SIWO).

In ISM, the number of ideas k_t at iteration t is kept small to manage computational resources. ISM can also dynamically adjust the prompts based on internal variables to initially focus on generating new ideas and gradually shift to refining and combining existing ones as idea quality improves.

ISM then prompts the LLMs to rank the ideas $I^{(t)}$ for each criterion c_i and assigns heuristic scores based on those ranks. For instance, if there are k_t ideas, an idea ranked first can receive a score of k_t, second a score of $k_t - 1$, and so on. Let s_{ji} be the score for idea j on criterion i. The overall score S_j for idea $I_j^{(t)}$ is calculated as: $S_j = \sum_{i=1}^{m} w_i s_{ji}$. Finally, ideas are ranked by their overall scores and the top candidates, along with their score vectors and overall scores, are stored in $\mathcal{K}_{\text{ideas}}$ marked as *pending* under iteration t for implementation in the next stage.

ISM can contain multiple "LLM agents" with different roles that collaborate by validating mathematical concepts, providing multiple rankings for better heuristic scores, updating KSM with additional material (e.g., searching the web for recent developments), checking ideas on the web for novelty, and engaging in discussions to simulate a group of experts from various domains. For instance, one agent could act as a proposer, another as a validator for mathematical soundness, and a third as a critic assessing novelty against existing literature.

Frameworks like LangChain [34] can be used to connect modules, define agent roles, and manage the flow of information between the LLMs and KSM.

• **Code Synthesis Module (CSM):** CSM translates selected ideas from ISM into executable code using LLMs specialized in coding, with self-refinement loops for continuous improvement and debugging. It accesses $\mathcal{K}_{\text{ideas}}$ to retrieve the selected ideas in iteration t, and incorporates information from $\mathcal{K}_{\text{alg}}$ and $\mathcal{K}_{\text{bg}}$ to guide the coding process. For each selected idea $I_j^{(t)}$, CSM may use the detailed prompts generated by ISM to facilitate code generation. It then generates an initial code C_{j1}, and iteratively refines the code using error messages and test results to improve functionality [27], with limits on the number of iterations or runtime to prevent indefinite refinement. The code implementations from this iteration $\mathcal{C}_j = \{C_{j1}, C_{j2}, \dots\}$ are stored in $\mathcal{K}_{\text{ideas}}$ under the corresponding idea $I_j^{(t)}$, along with version information, timestamps, and implementation details. Note that in CSM, multiple LLMs each responsible for an idea can be run in parallel to generate the codes, speeding up the process.

• **Evaluation and Iteration:** The generated implementations $\mathcal{C}_j$ are evaluated locally using predefined test cases and benchmarks to ensure correctness and measure target metrics specified by the user, such as runtime, memory usage, and scalability. Evaluation results E_{jk} for each implementation C_{jk} are recorded in $\mathcal{K}_{\text{ideas}}$ under the corresponding idea $I_j^{(t)}$, including vectors of the evaluation metrics. Paths to plots or images generated during evaluation, along with their descriptions, are also stored in KSM.

These results inform the next iteration, where ISM uses the updated $\mathcal{K}$ to generate ideas using the three aforementioned avenues and rank them while focusing on the most effective ones. This feedback loop allows the framework to iteratively improve the algorithm based on empirical performance data and guarantees the generation of more informed ideas that build upon previous successes and address past shortcomings. This is based on the possibility that some suggestions may not yield universal improvements, as seen in our case study where different optimizations were needed for sparse versus dense graphs. The framework treats such setbacks as learning opportunities that refine the solution space.

• **Termination:** The iterations continue until a termination condition (e.g., reaching a maximum number of iterations $T_{\max}$ or achieving target evaluation metrics) is met. The framework then outputs the best ideas from $\mathcal{K}_{\text{ideas}}$, accompanied by their implementations, version histories, and evaluation results.

4.1 Discussion

Our proposed framework distinguishes itself from previous works in three areas:

1) Cross-domain Knowledge Transfer: AEF utilizes the extensive knowledge bases of LLMs coupled with a RAG to take different scientific domains into account to generate innovative and interdisciplinary ideas to improve algorithms.

2) Multi-agent Collaboration: AEF utilizes multi-agent collaboration within ISM. Multiple LLM agents with specialized roles can verify mathematical con-

cepts, aggregate heuristic scores for better ranking, update KSM with new information, and engage in discussions.

3) Explainability: AEF emphasizes explainability by ensuring that each proposed idea is accompanied by detailed, human-readable explanations. Unlike models that focus solely on code generation, AEF uses LLMs to provide clear and comprehensive descriptions of each enhancement. This transparency helps users understand the reasoning behind each idea and the evaluation results, making the framework's processes more transparent and understandable. This human-centric approach is also critical for validating the LLM's outputs, guarding against potential hallucinations, and ensuring that the final enhancements are both sound and practical.

5 Conclusion & Future Works

In this paper, we demonstrated the potential of Large Language Models (LLMs) to enhance algorithms through a case study on the community detection algorithm SIWO, achieving over a 50% reduction in execution time for specific graph configurations. We showcased how interdisciplinary knowledge can drive significant performance improvements by integrating LLM-generated modifications.

Furthermore, we introduced the Algorithmic Enhancement Framework (AEF), a structured approach designed to generalize the use of LLMs for algorithm optimization across various computational domains.

Despite these promising results, our study is limited by its focus on a single algorithm and specific graph types, which may restrict the generalizability of the findings. Additionally, the scope was confined by different constraints, preventing the exploration of all proposed enhancement ideas.

Future work will aim to apply AEF to a broader range of algorithms from diverse domains to validate its effectiveness and generalizability. We also plan to implement an automated version of the framework to incorporate additional evaluation metrics, enhance multi-agent collaboration, and integrate user feedback mechanisms to align algorithm improvements with specific application requirements.

References

1. Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X. & Wang, C. Autogen: Enabling next-gen LLM applications via multi-agent conversation framework. *ArXiv Preprint ArXiv:2308.08155*. (2023)
2. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A. & Others Language models are few-shot learners. *NIPS*.**33** (2020)
3. Chen, H. & Ding, N. Probing the "Creativity" of Large Language Models: Can models produce divergent semantic association?. *Findings Of The Association For Computational Linguistics: EMNLP 2023*. (2023,12)
4. Franceschelli, G. & Musolesi, M. On the Creativity of Large Language Models. (2023)

5. Liu, F., Tong, X., Yuan, M. & Zhang, Q. Algorithm Evolution Using Large Language Model. (2023)
6. Blum, C. & Roli, A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*. **35**, (2003,9)
7. Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M., Dupont, E., Ruiz, F., Ellenberg, J., Wang, P., Fawzi, O., Kohli, P. & Fawzi, A. Mathematical discoveries from program search with large language models. *Nature*. **625**, (2024,1)
8. Mankowitz, D. et al., Faster sorting algorithms discovered using deep reinforcement learning. *Nature*. **618**, 257-263 (2023,6)
9. Zafarmand, M., Talebirad, Y., Austin, E., Largeron, C. & Zaïane, O. Fast local community discovery relying on the strength of links. *SNAM*. **13** (2023,9)
10. Gharaghooshi, S., Zaïane, O., Largeron, C., Zafarmand, M. & Liu, C. Addressing the Resolution Limit and the Field of View Limit in Community Mining. *IDA*. (2020)
11. Bang, Y., Cahyawijaya, S., Lee, N., Dai, W., Su, D., Wilie, B., Lovenia, H., Ji, Z., Yu, T., Chung, W., Do, Q., Xu, Y. & Fung, P. A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity. (2023)
12. Meng, W. & Qu, R. Automated design of search algorithms: Learning on algorithmic components. *Expert Systems With Applications*. (2021)
13. Zhao, Q., Duan, Q., Yan, B., Cheng, S. & Shi, Y. Automated Design of Metaheuristic Algorithms: A Survey. (2023)
14. Burke, E., Hyde, M., Kendall, G., Ochoa, G., Özcan, E. & Woodward, J. A Classification of Hyper-Heuristic Approaches: Revisited. *Handbook Of Metaheuristics*. **272** pp. 453-477 (2019)
15. Liu, F., Tong, X., Yuan, M., Lin, X., Luo, F., Wang, Z., Lu, Z. & Zhang, Q. An Example of Evolutionary Computation + Large Language Model Beating Human: Design of Efficient Guided Local Search. (2024)
16. Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q., Zhou, D. & Chen, X. Large Language Models as Optimizers. (2023)
17. Lehman, J., Gordon, J., Jain, S., Ndousse, K., Yeh, C. & Stanley, K. Evolution through Large Models. (2022)
18. Guo, P., Chen, Y., Tsai, Y. & Lin, S. Towards Optimizing with Large Language Models. (2023)
19. Meyerson, E., Nelson, M., Bradley, H., Gaier, A., Moradi, A., Hoover, A. & Lehman, J. Language Model Crossover: Variation through Few-Shot Prompting. (2023)
20. Singh, I., Blukis, V., Mousavian, A., Goyal, A., Xu, D., Tremblay, J., Fox, D., Thomason, J. & Garg, A. ProgPrompt: Generating Situated Robot Task Plans using Large Language Models. *ICRA*. (2023)
21. Mei, Y., Chen, Q., Lensen, A., Xue, B. & Zhang, M. Explainable Artificial Intelligence by Genetic Programming: A Survey. *IEEE Transactions On Evolutionary Computation*. **27**, 621-641 (2023)
22. Bradley, H., Fan, H., Galanos, T., Zhou, R., Scott, D. & Lehman, J. The OpenELM Library: Leveraging Progress in Language Models for Novel Evolutionary Algorithms. (OpenReview Archive Direct Upload,2023,10)
23. OpenAI, Achiam, J. et al. GPT-4 Technical Report. (2023)
24. Touvron, H. et al. Llama 2: Open Foundation and Fine-Tuned Chat Models. (2023)
25. Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y., Li, Y., Lundberg, S., Nori, H., Palangi, H., Ribeiro, M. & Zhang, Y. Sparks of Artificial General Intelligence: Early experiments with GPT-4. (2023)

26. Anil, R. et al. PaLM 2 Technical Report. (2023)
27. Sobania, D., Briesch, M., Hanna, C. & Petke, J. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. *APR*. (2023,5)
28. Li, G., Hammoud, H., Itani, H., Khizbullin, D. & Ghanem, B. Camel: Communicative agents for "mind" exploration of large scale language model society. *ArXiv Preprint ArXiv:2303.17760*. (2023)
29. Hong, S., Zheng, X., Chen, J., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S., Lin, Z., Zhou, L. & Others Metagpt: Meta programming for multi-agent collaborative framework. *ArXiv Preprint ArXiv:2308.00352*. (2023)
30. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S. & Kiela, D. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *NIPS*. **33** (2020)
31. Talebirad, Y., Zafarmand, M., Zaiane, O. & Largeron, C. USIWO: A Local Community Search Algorithm for Uncertain Graphs. *ASONAM*. (2024)
32. Ahmad, A., Waseem, M., Liang, P., Fahmideh, M., Aktar, M. & Mikkonen, T. Towards Human-Bot Collaborative Software Architecting with ChatGPT. *International Conference On Evaluation And Assessment In Software Engineering*. (2023)
33. Yuan, W., Pang, R., Cho, K., Li, X., Sukhbaatar, S., Xu, J. & Weston, J. Self-Rewarding Language Models. (2024)
34. Chase, H. LangChain: Building Applications with LLMs through Composability. (2023)
35. Hevner, A.R., March, S.T., Park, J. & Ram, S. Design science in information systems research. *MIS Quarterly*. **28**, 75–105 (2004)