Evaluating Efficiency and Novelty of LLM-Generated Code for Graph Analysis

Atieh Barati Nia, Mohammad Dindoost, David A. Bader
Department of Data Science, New Jersey Institute of Technology, Newark, NJ, USA
{ab2763, md724, bader}@njit.edu

***Abstract*—Large Language Models (LLMs) are increasingly used to automate software development, yet most prior evaluations focus on functional correctness or high-level languages such as Python. As one of the first systematic explorations of LLM-assisted software performance engineering, we present a comprehensive study of LLMs' ability to generate efficient C implementations of graph-analysis routines—code that must satisfy stringent runtime and memory constraints. This emerging field of LLM-assisted algorithm engineering holds significant promise, as these models may possess the capability to design novel approaches that improve existing algorithms and their implementations. Eight state-of-the-art models (OpenAI ChatGPT o3 and o4-mini-high, Anthropic Claude 4 Sonnet and Sonnet Extended, Google Gemini 2.5 Flash and Pro, xAI Grok 3-Think, and DeepSeek DeepThink R1) are benchmarked using two distinct approaches. The first approach evaluates the ability of LLMs to generate algorithms that outperform existing benchmarks. The second approach assesses their capability to generate graph algorithms for integration into performance-critical systems. The results show that Claude Sonnet 4 Extended achieves superior performance in ready-to-use code generation and efficiency, outperforming human-written baselines in triangle counting. Although our findings demonstrate that contemporary LLMs excel in optimizing and integrating established algorithms, the potential for these models to eventually invent transformative algorithmic techniques represents a compelling frontier for future research. We provide prompts, generated code, and measurement scripts to promote reproducible research in this rapidly evolving domain. All of the source code is available on GitHub at https://github.com/Bader-Research/LLM-triangle-counting/.**

***Index Terms*—Software performance engineering; algorithm engineering; triangle counting.**

I. Introduction

The rapid advancement of Artificial Intelligence (AI), particularly large language models (LLMs), has significantly impacted both everyday life and scientific workflows, accelerating research and development across disciplines. LLMs have shown significant promise in improving learning, automating tasks, and boosting productivity in a variety of domains. A recent survey by Chang *et al.* [1] highlights a comprehensive review of evaluation efforts related to LLM applications in diverse areas, including reasoning, medical applications, ethics, and education. One particularly impactful application of LLMs is to help programmers and computer scientists by streamlining software development and minimizing human error. As a result, considerable research has focused on evaluating the correctness of code generated by these models [2]–[5] and some research's focus is on correctness and debugging [6]–[8]. Benchmarks such as HumanEval [9] and MBPP [10] have been widely used to evaluate the functional correctness of LLM-generated code through the completion of unit tests. Although these benchmarks assess whether the code works as intended, they do not account for performance, which is the focus of our study.

Although Python dominates recent research, computer scientists still depend on lower-level languages such as C and C++ to achieve peak performance. Assessing the efficiency of LLM generated C code is therefore vital to judging the ability of these models to produce optimized implementations. However, to our knowledge, no study has systematically examined the performance of LLM-generated code in C, a language central to high-performance and scientific computing.

Moreover, prompting an LLM to generate an algorithm without explicit contextual constraints may raise issues of construct validity because we cannot know whether the model is simply repeating code it saw during training. To address this concern, we adopt a more rigorous evaluation strategy, the first to our knowledge that places the models inside an existing codebase, by supplying concrete source files and asking the LLMs to produce algorithms that integrate with them. This targeted setup sharply reduces the chance of pre-training contamination and yields a clearer measure of each model's genuine coding ability.

This paper offers the first systematic evaluation of cutting-edge LLMs in producing high-performance C implementations for graph algorithms. We evaluated the generated code along four axes: compilability, functional correctness, execution time, and memory footprint. We anticipate that our findings will not only guide LLM developers in enhancing their models but also assist those computer and computational scientists, who implement application codes or graph analytics, in selecting the most suitable LLMs for their real-world applications.

A. Related work

Recent research has increasingly focused on the efficiency of code generated by large language models (LLMs), moving beyond the traditional emphasis on correctness. Huang *et al.* [11] introduce EffiBench, a benchmark that evaluates the runtime and algorithmic complexity of Python code, reveal-

This work was supported in part by NSF grants CCF-2109988 and OAC-2402560, and CCF-2453324.

ing that syntactically correct code is not always efficient. Similarly, Qiu *et al.* [12] propose a rigorous benchmark to jointly assess both the correctness and efficiency of the LLM-generated Python code. Licorish *et al.* [13] compares the human-generated and LLM code in correctness, efficiency, and maintainability, concluding that current models still lack human performance. Du *et al.* [14] presents Mercury, a broad efficiency benchmark for LLMs across multiple programming tasks, while Niu *et al.* [15] and Liu *et al.* [16] examine key challenges and proposed frameworks for measuring efficiency in generated Python code.

Although these studies yield valuable information for a popular high-level language, C remains the workhorse of high-performance computing (HPC) and system programming. Virtually all supercomputing kernels, MPI/OpenMP libraries, operating system components, and performance-critical embedded routines are still written in C (or C-centric dialects such as C++, CUDA, and OpenCL). Unlike Python - whose interpreter and dynamic typing add layers of abstraction - C exposes explicit memory management and low-level control over hardware, so a small inefficiency can translate into orders-of-magnitude slowdowns at HPC scale. Evaluating LLM output in C therefore probes a harder and more consequential problem: can these models generate code that meets the stringent runtime, memory, and power budgets demanded by real-world analytics, exascale simulations, and kernel-level software?

Other relevant efforts focus on improving the quality or efficiency of the code. Siddiq *et al.* [17] introduces FRANC, a framework designed to improve the correctness and readability of LLM-generated Java and Python code. Yetistiren *et al.* [18] empirically evaluate tools such as GitHub Copilot, Amazon CodeWhisperer, and ChatGPT, focusing on quality dimensions such as correctness and maintainability, but primarily using earlier LLMs. In contrast, our study evaluates the performance of code produced by state-of-the-art models.

Ye *et al.* [19] propose LLM4EFFI, a framework that guides LLMs to produce more efficient and correct Python code using prompt engineering and iterative feedback. Likewise, Waghjale *et al.* [20] introduces ECCO, a post-processing framework that optimizes the generated code for efficiency. Unlike these works, our study evaluates the inherent performance of the code generated by modern LLMs in C, without manual intervention or optimization, to reflect their true capabilities in producing performant code.

II. Experiment Design

In this section, we describe the data and source code of the program, the models we use for evaluation and comparison, and how we conduct experiments to investigate the performance of each LLM for assisting with software performance engineering tasks of optimizing the running time of graph algorithm code.

A. Data

Bader [21] introduced a comprehensive framework in C to evaluate the performance of the existing corpus of human-generated triangle counting algorithms. Within this framework, he collected and implemented all known CPU-based triangle-counting algorithms for a sparse graph given in compressed sparse row (CSR) format, measured their execution times, and established a benchmark for comparison. This framework serves as a good input to the LLMs to incorporate their LLM-generated code and to reduce the risk of using pretrained data.

B. Models

Since we want to study the ability of LLMs to assist programmers with software performance engineering, we selected the seven most frequently used frontier AI models available at the time of writing this paper and gained attraction because of their ability to generate source code.

OpenAI ChatGPT o3 and ChatGPT o4-mini-high: OpenAI's models are among the most well-known LLMs, mostly known as ChatGPT, having played a pivotal role in ushering in the modern era of generative AI. So, we included the evaluation of the latest available versions related to code implementation which were released on April 16, 2025. OpenAI o3 which uses advanced reasoning by spending more time thinking before responding, and OpenAI o4-mini-high which said to be great at coding and visual reasoning are selected to be the representative of OpenAI company.

Anthropic Claude 4 Sonnet and Claude 4 Sonnet Extended: Anthropic's Claude models are renowned for their advanced reasoning, language comprehension, and efficient code generation. The most recent model, Claude 4 Sonnet which was released on 22 May 2025, includes an extended variant with enhanced capabilities, both of which we incorporate in our analysis.

Google Gemini 2.5 Flash and Gemini 2.5 Pro: Google's strong track record in AI innovation makes its latest Gemini models essential for our benchmark. We therefore include Gemini 2.5 Flash, first released in public preview on April 17, 2025, alongside Gemini 2.5 Pro, whose public preview build launched on May 6, 2025, and is specifically promoted for advanced reasoning, mathematical skill, and code generation prowess, ensuring our evaluation is truly comprehensive.

xAI Grok 3 - Think: we include xAI's Grok 3, released on February 17, 2025, and evaluate it in its dedicated "Think" mode, which is optimized for deeper reasoning, to gauge how the latest entry performs under our efficiency benchmark.

DeepSeek DeepThink (R1): At its launch on 20 January 2025, DeepSeek R1 (marketed as DeepThink) attracted much attention for matching the performance of costly proprietary LLMs while remaining free and requiring far fewer training resources. Due to this exceptional efficiency and cost effectiveness, we include DeepSeek DeepThink (R1) in our evaluation to measure its capabilities against the established models in our benchmark.

C. Methodology

A critical challenge in evaluating LLMs is the risk that their outputs may reproduce memorized training data rather than demonstrate genuine reasoning. To address this, we employ a novel evaluation strategy: Instead of relying on textual prompts alone, we provide each LLM with the source files C from Bader's benchmark (described in Section II-A) and instruct them to generate a functional triangle counting algorithm that integrates into the existing framework. This approach allows us to assess the models' true algorithmic reasoning capabilities, minimizing the influence of memorized solutions.

Our study comprises two approaches: Optimization approach and Algorithm-Synthesis approach. The optimization approach supplies each LLM with the complete collection of C source files for existing sequential triangle-counting implementations and challenges it to generate the most efficient routine. By exposing the models to every known approach, we investigated whether they can synthesize genuinely novel strategies that outperform human-written code. The complete prompt template appears in Fig. 1. Note that we cold-start each LLM before the prompt.

Prompt: The attached code implements triangle counting in a graph given in CSR format. Write a C routine `tc_fast()` with the API below that is the fastest sequential triangle counting code you can generate.

```
UINT_t tc_fast(const GRAPH_TYPE
*graph)
```

Attached Files: (bfs.c, graph.c, main.c, queue.c, tc.c)

Fig. 1. The First Prompt Structure Template.

The Algorithm-Synthesis approach evaluates how well each model can generate efficient implementations of Triangle Counting, Diameter Finding, Vertex Connectivity, Edge Connectivity, and Clique Number, when no prior code for these algorithms is provided. In this approach the provided C source contains the project's core graph infrastructure; the algorithm implementations are deliberately absent. Each model must therefore write a new routine that integrates seamlessly with this codebase. The prompt template for this approach is shown in Fig. 2.

After collecting the generated code for both approaches, we incorporated them into the benchmark and executed all tests to compare performance. The benchmark reports the average runtime over 10 executions for each implementation, providing statistical validity and smoothing out measurement noise to ensure robust and reliable performance measurements.

Prompt: The attached code implements a graph in CSR format. Write a C routine `fast()` with the API below that is the fastest sequential implementation for finding the {ALGORITHM} of a graph that can be integrated into the attached files:

```
UINT_t fast(const GRAPH_TYPE *graph)
```

Attached Files: (bfs.c, graph.c, main.c, queue.c)

Fig. 2. The Second Prompt Structure Template. {ALGORITHM} is replaced by "number of triangles", "diameter", "vertex connectivity", "edge connectivity", and "clique number"

III. Experimental Results

The experiments were conducted on Wulver, NJIT's high-performance supercomputer, using a single core of an AMD EPYC 7753 CPU @ 2.45 GHz and 512 GB of RAM.

A. Optimization Approach

The evaluation demonstrates that while all models examined successfully generated code that seamlessly integrated into the benchmark framework without requiring modifications, DeepSeek R1, Gemini 2.5 Flash, and Grok 3-Think produced implementations that failed to correctly count the number of triangles. Consequently, due to these incorrect outputs, the algorithmic details of the code generated by these three models were excluded from further analysis.

Most models used efficient approaches already present in the benchmark, combining optimized techniques such as degree-based vertex sorting (using the benchmark's built-in function), hash-based intersection, the Forward Triangle Counting algorithm [22], [23] and the BFS traversal [24]. Although not groundbreaking, these strategies were effectively synthesized into workable solutions.

Although both Gemini 2.5 Pro and Claude Sonnet 4 Extended employed similar core methods, using the Forward Counting Algorithm combined with sorting and hashing, Claude Sonnet 4 Extended achieved faster performance. This improvement indicates a more effective utilization of memory management and cache locality by the code generated from the Claude model. The code of the other models achieves their fastest running times by using higher peak memory.

ChatGPT o4-mini-high implemented the Forward Triangle Counting Algorithm for its solution. However, its implementation did not incorporate advanced optimizations such as hashing or sorting of nodes to accelerate neighbor intersection computations, leading to suboptimal time performance, particularly in graphs with high-degree vertices where efficient lookup mechanisms could reduce redundant comparisons. As a result, its implementation was third among LLM implementations, trailing behind human-written code.

ChatGPT o3 incorporated two human-written helper functions, one for graphs with more than 16,384 edges and the

other for smaller instances. However, its choice of which function to apply was not optimal compared to other LLMs or the hand-written baseline, leading to a slower running time despite achieving the lowest peak memory usage compared to other LLMs.

Claude Sonnet 4 also adopted the Forward Triangle Counting Algorithm and incorporated hashing to optimize neighbor intersection operations, a strategy that theoretically should improve performance for common neighbors. However, its implementation exhibited significantly worse running time compared to other models, which shows its poor performance in implementing efficient code.

The source code for the implementations produced by each model is available on GitHub at https://github.com/Bader-Research/LLM-triangle-counting/. Table III then summarizes their comparative performance, both execution time and maximum memory usage, on the RMAT-18 dataset [25].

To comprehensively evaluate the performance of the generated code, we tested the corrected algorithms on graphs ranging from RMAT-6 to RMAT-18. This allowed us to observe runtime behavior across both small and large graphs. We also included an efficient human-generated implementation of Bader's algorithm that incorporates sorting vertices, BFS traversal, and hash arrays, as well as two different implementations of the forward triangle counting algorithm with neighbor intersections based on hash.

The results in Table IV show that Claude 4 Sonnet Extended - and, to a slightly lesser extent, Gemini 2.5 Pro - achieved the fastest running times by trading increased peak memory for lower execution time. Both models outperformed every other LLM and even the human-written baseline, despite relying on existing triangle-counting techniques rather than inventing new algorithms. This suggests that while large language models excel at optimizing and refining code, their current capabilities remain focused on improving known methods rather than developing fundamentally new algorithmic advances.

B. Algorithm-Synthesis Approach

In this approach, we evaluate the capacity of each model to produce efficient implementations of established graph algorithms without access to any pre-existing code. Performance is evaluated according to two principal criteria: Ready-To-Use (RTU) Capability and Efficiency Trade-Offs.

1) Ready-To-Use (RTU) Capability: This parameter is defined based on three important properties that generated code should have from a user's perspective:

- **Compilability:** The code must compile successfully without any manual modifications.
- **Correctness:** The code must pass all test cases in our comprehensive suite, which includes different types of graphs.
- **Timeliness:** The implementation must run within an acceptable runtime threshold, ensuring practical usability.

If the generated code does not meet any of these criteria or even fails in a single test case, it is not considered an RTU.

TABLE I
RTU PERCENTAGE FOR VARIOUS MODELS

model	RTU percentage
Google Gemini 2.5 Flash	17%
Google Gemini 2.5 Pro	50%
Deepseek DeepThink	33%
OpenAI ChatGPT o3	17%
OpenAI ChatGPT o4-mini-high	50%
xAI Grok 3 Think	50%
Anthropic Claude Sonnet 4	50%
Anthropic Claude Sonnet 4 Extended	**83%**

Table I reports the RTU rate for each model in the six target graph problems. The results indicate that Claude Sonnet 4 Extended achieves the highest RTU score, successfully generating fully compilable, correct, and timely code for 83% of the algorithmic tasks.

2) Efficiency Trade-Offs: Beyond the status of the RTU, we also evaluate the time-memory trade-off of each model when executing the generated routines. Specifically, we record two metrics for each model-algorithm pair:

1) **Relative runtime** (t)—normalized so that the fastest implementation for a given algorithm has $t = 1$.
2) **Relative peak memory usage** (m)—normalized so that the implementation that uses the least memory for the same algorithm has $m = 1$.

Table V presents the relative values for the code generated by each model and its RTU algorithm. Models that did not generate a valid RTU implementation are omitted from the analysis and are indicated with a dash.

To compare the models on a single efficiency metric, we introduce a rate that combines relative runtime and peak memory consumption:

$$\text{rate} = \begin{cases} 1/(tm), & \text{if the model produced RTU implementation} \\ 0, & \text{otherwise.} \end{cases}$$

Here, t is the code's running time expressed as a fraction of the fastest implementation, and m is its maximum memory usage expressed as a fraction of the most memory-efficient implementation. A higher rate therefore indicates greater overall efficiency. The overall score of each model is calculated by adding its rates in the six benchmark algorithms; The resulting totals appear in Table II.

The results indicate that Claude Sonnet 4 Extended is currently the most proficient LLM for software performance engineering of sparse graph algorithm code in terms of combined runtime and peak memory usage, achieving the highest rate (3.11). Grok 3 Think ranks second, followed by Claude Sonnet 4 in third place.

TABLE II
EFFICIENCY RATES OF DIFFERENT MODELS.

Model	Efficiency Rate
Google Gemini 2.5 Flash	0.76
Google Gemini 2.5 Pro	1.82
DeepSeek DeepThink	0.81
OpenAI ChatGPT o3	0.04
OpenAI ChatGPT o4-mini-high	1.05
xAI Grok3 Think	2.84
Anthropic Claude 4 Sonnet	2.02
Anthropic Claude 4 Sonnet Extended	**3.11**

IV. CONCLUSIONS

As one of the first systematic explorations of LLM-assisted software performance engineering, this empirical study establishes important foundational insights into the capabilities and potential of this rapidly evolving field. Our comprehensive evaluation of eight state-of-the-art models demonstrates that contemporary LLMs can successfully generate efficient C implementations for performance-critical graph algorithms, with all models seamlessly integrating their code into our benchmark framework during the optimization approach. The results reveal significant efficiency differences among models, with Claude 4 Sonnet Extended and Gemini 2.5 Pro achieving the most impressive runtime and memory profiles, even outperforming human-written baselines in triangle counting. In the algorithm-synthesis approach, Claude Sonnet 4 Extended achieved the highest Ready-To-Use (RTU) rate (83%) and generated the most efficient implementations, establishing it as the current leader for software performance engineering tasks in this domain.

Although our findings demonstrate that current LLMs excel in synthesizing, optimizing, and refining established algorithmic approaches, this represents a crucial first step in the emerging field of LLM-assisted algorithm engineering. The observed capabilities suggest that these models possess a sophisticated understanding of algorithmic principles, memory management, and performance optimization strategies. Most importantly, the potential for LLMs to eventually transcend the optimization of existing methods and invent transformative algorithmic techniques represents one of the most compelling frontiers in computational science.

This work provides essential benchmarks and methodologies for a field poised for rapid advancement, offering both practitioners and researchers the foundational tools needed to harness LLMs for high-performance computing applications while establishing the groundwork for future breakthroughs in algorithmic innovation.

V. FUTURE WORK

The pioneering nature of this study opens numerous avenues for advancing LLM-assisted software performance engineering, particularly as this field enters a phase of accelerated development. Several critical research directions warrant immediate exploration to unlock the transformative potential of these technologies.

Algorithmic Innovation and Creativity: Future research should investigate methods to enhance the capacity of LLMs for genuine algorithmic invention, moving beyond the optimization of known techniques toward the discovery of fundamentally novel computational strategies. This includes developing prompting techniques, training methodologies, and evaluation frameworks specifically designed to foster algorithmic creativity and breakthrough thinking.

Massively Parallel and GPU Computing: Extending evaluation to parallel graph algorithms and GPU architectures represents both a natural progression and a critical need, as LLMs may demonstrate superior capabilities in optimizing for complex parallel execution patterns, load balancing, synchronization, and memory hierarchy optimization in heterogeneous computing environments.

Theoretical Foundations and Guarantees: Establishing rigorous theoretical frameworks for LLM-generated algorithms, including formal analyses of time complexity, memory consumption, and correctness guarantees, will be essential for deploying these techniques in high-stakes and safety-critical computational applications.

Domain-Specific Algorithm Engineering: Expanding beyond graph algorithms to encompass broader algorithmic domains, including numerical methods, machine learning kernels, cryptographic implementations, and scientific computing routines, will reveal the full scope of the potential impact of LLMs on computational science.

Interactive and Iterative Algorithm Development: Investigating human-AI collaboration models where LLMs work interactively with domain experts to iteratively refine and evolve algorithmic solutions, which could lead to hybrid approaches that combine human intelligence with AI optimization capabilities.

Real-World Deployment and Integration: Developing frameworks to seamlessly integrate LLM-generated high-performance code into production systems, including automated testing, verification and continuous optimization pipelines that can adapt algorithms as hardware and requirements evolve.

As this field rapidly evolves, we anticipate that LLMs will increasingly demonstrate the capability to design novel approaches that fundamentally advance the state of algorithmic science, potentially revolutionizing how we approach computational problem solving across disciplines. The foundation established in this work provides a starting point for what promises to be one of the most transformative developments in algorithm engineering.

TABLE III

COMPARATIVE OVERVIEW OF THE LLMS

COMPILABLE: THE CODE COMPILED SUCCESSFULLY WITHOUT ANY MODIFICATIONS. CORRECTNESS: THE MODEL COUNTED THE TRIANGLES CORRECTLY. # TRIANGLES: NUMER OF TRIANGLES THAT THE MODEL COUNTED FOR RMAT18. RUNTIME RMAT 18: THE RUNTIME OF THE MODEL ON RMAT 18 IN SECONDS. MAX MEM. US.: THE RELATIVE MAXIMUM MEMORY USAGE OF THE MODEL ON RMAT 18 BY TAKING CHATGPT O3 AS THE BASELINE. SORT: THE MODEL SORTED THE VERTICES BY DEGREE AT THE START OF THE ALGORITHM. HASH: THE MODEL USED HASH TABLES FOR INTERSECTION OPERATIONS. FTC: THE MODEL USED THE FORWARD COUNTING ALGORITHM IN THE PROPOSED METHOD. BFS: THE MODEL EMPLOYED BREADTH-FIRST SEARCH (BFS) IN ITS IMPLEMENTATION.

Model	Compilable	Correctness	# triangles	Runtime	Max Mem. Us.	Sort	Hash	FTC	BFS
ChatGPT o3	✓	✓	101930789	16.814495	1	✓	✓	✓	✓
ChatGPT o4-mini-high	✓	✓	101930789	4.714634	1.01	×	×	✓	×
Claude 4 Sonnet	✓	✓	101930789	192.777913	1	×	✓	✓	×
Claude 4 Sonnet Extended	✓	✓	101930789	**0.624521**	1.45	✓	✓	✓	×
Gemini 2.5 Pro	✓	✓	101930789	0.665562	1.46	✓	✓	✓	×
Gemini 2.5 Flash	✓	×	305792367	NA	NA	NA	NA	NA	NA
DeepSeek DeepThink (R1)	✓	×	203861578	NA	NA	NA	NA	NA	NA
Grok 3-Think	✓	×	3812543	NA	NA	NA	NA	NA	NA

TABLE IV

EXECUTION TIME OF EACH MODEL (IN SECONDS).

VERTICES: NUMBER OF VERTICES IN THE GRAPH. # EDGES: NUMBER OF EDGES IN THE GRAPH. # TRIANGLES: NUMBER OF TRIANGLES IN THE GRAPH. O3: OPENAI CHATGPT O3. O4MH: OPENAI CHATGPT O4-MINI-HIGH. CL 4: ANTHROPIC CLAUDE SONNET 4. CL 4 EX.: ANTHROPIC CLAUDE SONNET 4 EXTENDED. GEMI.P: GEMINI 2.5 PRO. BADERBFS: BADER ALGORITHM BASED ON BFS. FORWARD 1: FIRST IMPLEMENTATION BASED ON FORWARD TRIANGLE COUNTING.FORWARD 2: SECOND IMPLEMENTATION BASED ON FORWARD TRIANGLE COUNTING.

Graph	# Vertices	# Edges	# triangles	o3	o4MH	Cl 4	Cl 4 Ex.	Gemi.P	BaderBFS	Forward 1	Forward 2
RMAT 6	64	1024	9100	**0.000022**	0.000028	0.000230	0.000033	0.000023	0.000029	**0.000022**	0.000023
RMAT 7	128	2048	18855	0.000059	0.000125	0.000303	**0.000050**	0.000060	0.000070	0.000053	0.000056
RMAT 8	256	4096	39602	0.000149	0.000371	0.001019	**0.000124**	0.000150	0.000204	0.000155	0.000154
RMAT 9	512	8192	86470	0.000419	0.000992	0.003197	**0.000320**	0.000360	0.000488	0.000374	0.000386
RMAT 10	1024	16384	187855	0.000990	0.002567	0.010182	**0.000727**	0.000831	0.001126	0.000891	0.000859
RMAT 11	2048	32768	408876	0.002312	0.006473	0.034632	**0.001707**	0.001918	0.002364	0.002048	0.001882
RMAT 12	4096	65536	896224	0.005334	0.017888	0.115028	**0.003860**	0.004316	0.005302	0.004758	0.004389
RMAT 13	8192	131072	1988410	0.012643	0.045412	0.408823	**0.009144**	0.010036	0.011355	0.011520	0.010052
RMAT 14	16384	262144	4355418	0.031400	0.115286	1.364177	**0.022311**	0.023743	0.026098	0.029162	0.023728
RMAT 15	32768	524288	9576800	1.094913	0.300161	4.770884	**0.053767**	0.056490	0.058261	0.076366	0.057285
RMAT 16	65536	1048576	21133772	2.727890	0.756705	16.158756	**0.128004**	0.134363	0.131487	0.189328	0.135958
RMAT 17	131072	2097152	46439638	6.737418	1.860513	56.074888	**0.283528**	0.298584	0.307493	0.465803	0.303541
RMAT 18	262144	4194304	101930789	16.814495	4.714634	192.777913	**0.624521**	0.665562	0.723090	1.259261	0.733882

TABLE V

RELATIVE RUNNING TIMES (TIME) AND RELATIVE PEAK MEMORY USAGE (MEM) FOR EACH GENERATED CODE; THE NUMERIC SUFFIX APPENDED TO AN ALGORITHM'S NAME DENOTES THE RMAT GRAPH SCALE ON WHICH IT WAS EVALUATED.

	Triangle Counting 15		**Diameter Finding 15**		**Vertex Connectivity 15**		**Edge Connectivity 12**		**Clique Number 12**		**Chromatic Number 15**	
	R. Time	**R. Mem**	**R. Time**	**R. Mem**	**R. Time**	**R. Mem**	**R. Time**	**R. Mem**	**R. Time**	**R. Mem**	**R. Time**	**R. Mem**
Google Gemini 2.5 Flash	-	-	1.32	1	-	-	-	-	-	-	-	-
Google Gemini 2.5 Pro	20.68	1	1.30	1	-	-	1	1	-	-	-	-
DeepSeek DeepThink	37.96	1	1.27	1	-	-	-	-	-	-	-	-
OpenAI ChatGPT o3	-	-	-	-	-	-	1.64	15.85	-	-	-	-
OpenAI ChatGPT o4-mini-high	4.78	1	1.28	1	-	-	1.05	15.85	-	-	-	-
xAI Grok3 Think	1	1	1.19	1	-	-	-	-	1	1	-	-
Anthropic Claude 4	57.93	1.04	1	1	-	-	-	-	-	-	1	1
Anthropic Claude 4 Extended	8.60	1.04	1	1	1	1	12.05	31.51	-	-	1	1

REFERENCES

[1] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang *et al.*, "A survey on evaluation of large language models," *ACM Transactions on Intelligent Systems and Technology*, vol. 15, no. 3, pp. 1–45, 2024.

[2] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "CodeT: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.

[3] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 558–21 572, 2023.

[4] W. Tong and T. Zhang, "CodeJudge: Evaluating code generation with large language models," *arXiv preprint arXiv:2410.02184*, 2024.

[5] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub copilot's code suggestions," ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–5.

[6] T. Dinh, J. Zhao, S. Tan, R. Negrinho, L. Lausen, S. Zha, and G. Karypis, "Large language models of code fail at completing code with potential bugs," *Advances in Neural Information Processing Systems*, vol. 36, pp. 41 386–41 412, 2023.

[7] F. Tambon, A. Moradi-Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, and G. Antoniol, "Bugs in large language models generated code: An empirical study," *Empirical Software Engineering*, vol. 30, no. 3, pp. 1–48, 2025.

[8] K. Jesse, T. Ahmed, P. T. Devanbu, and E. Morgan, "Large language models and simple, stupid bugs," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 563–575.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[10] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[11] D. Huang, Y. Qing, W. Shang, H. Cui, and J. Zhang, "EffiBench: Benchmarking the efficiency of automatically generated code," *Advances in Neural Information Processing Systems*, vol. 37, pp. 11 506–11 544, 2024.

[12] R. Qiu, W. W. Zeng, J. Ezick, C. Lott, and H. Tong, "How efficient is LLM-generated code? a rigorous & high-standard benchmark," *arXiv preprint arXiv:2406.06647*, 2024.

[13] S. A. Licorish, A. Bajpai, C. Arora, F. Wang, and K. Tantithamthavorn, "Comparing human and LLM generated code: The jury is still out!" *arXiv preprint arXiv:2501.16857*, 2025.

[14] M. Du, A. T. Luu, B. Ji, Q. Liu, and S.-K. Ng, "Mercury: A code efficiency benchmark for code large language models," *arXiv preprint arXiv:2402.07844*, 2024.

[15] C. Niu, T. Zhang, C. Li, B. Luo, and V. Ng, "On evaluating the efficiency of source code generated by LLMs," in *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, 2024, pp. 103–107.

[16] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. Zhang, "Evaluating language models for efficient code generation," *arXiv preprint arXiv:2408.06450*, 2024.

[17] M. L. Siddiq, B. Casey, and J. C. Santos, "FRANC: A lightweight framework for high-quality code generation," in *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2024, pp. 106–117.

[18] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT," *arXiv preprint arXiv:2304.10778*, 2023.

[19] T. Ye, W. Huang, X. Zhang, T. Ma, P. Liu, J. Yin, and W. Wang, "LLM4EFFI: Leveraging large language models to enhance code efficiency and correctness," *arXiv preprint arXiv:2502.18489*, 2025.

[20] S. Waghjale, V. Veerendranath, Z. Z. Wang, and D. Fried, "ECCO: Can we improve model-generated code efficiency without sacrificing functional correctness?" *arXiv preprint arXiv:2407.14044*, 2024.

[21] D. A. Bader, "Fast triangle counting," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2023, pp. 1–6.

[22] T. Schank and D. Wagner, "Finding, counting and listing all triangles in large graphs, an experimental study," in *International workshop on experimental and efficient algorithms*. Springer, 2005, pp. 606–609.

[23] T. Schank, "Algorithmic aspects of triangle-based network analysis," PhD dissertation, Karlsruhe Institute of Technology, 2007.

[24] D. A. Bader, F. Li, A. Ganeshan, A. Gundogdu, J. Lew, O. A. Rodriguez, and Z. Du, "Triangle counting through cover-edges," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2023, pp. 1–7.

[25] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–446.