NP Complete Problem Solver Agents

Mohammad Taha Majlesi
tahamajlesi@ut.ac.ir

Mahdi Naeni
mhd.naeeni@gmail.com

Babak Hosseini Mohtasham
babak.hosseini.m@ut.ac.ir

Abstract

Large Language Models (LLMs) have demonstrated remarkable capabilities in pattern recognition and reasoning across diverse domains. This project investigates the potential of LLMs to address NP-Complete (NPC) problems, a class of computationally intractable challenges central to operations research and computer science. We analyze the inherent ability of LLMs to solve canonical NPC problems like Hamiltonian Path (HAMPATH) and the Boolean Satisfiability Problem (3-SAT). Furthermore, we explore methods to enhance their performance, including equipping LLM agents with external tools and fine-tuning smaller, specialized models. Our empirical results indicate that while off-the-shelf LLMs struggle with NP complete problems, by utilizing the combination of fine-tuned small LLMs with large LLMs in an agentic system the performance can greatly increase offering a new way for tackling complex computational tasks. This work contributes to understanding the computational boundaries of LLMs and outlines a path toward building more powerful SMT solvers. The code is available on Github at this link [1].

1 Introduction

Many of the most critical computational challenges in fields such as logistics, scheduling, hardware verification, and resource allocation are classified as Non-deterministic Polynomial-time Complete (NP-Complete) problems, which are notoriously difficult to solve efficiently as their input size grows. Traditional exact algorithms become computationally infeasible for large instances, while heuristic approaches often fail to generalize or lack adaptability in dynamic environments.

[1] https://github.com/babakhm83/LLM_UT_FINAL_PROJECT

The practical importance of NPC problems has driven the development of highly sophisticated solver tools. Satisfiability Modulo Theories (SMT) solvers, such as Z3, represent the state-of-the-art in solving these problems exactly. Integrated into modern programming environments like Python, they make formulating and attacking complex instances more approachable. However, despite their efficiency and low constant-factor overhead, their underlying runtime complexity remains exponential.

This project aims to first analyze the power of LLMS in solving NPC problems and then try to improve them using different methods, such as providing LLM agents with useful tools or just fine-tuning small models. We hypothesize that LLMs, by virtue of their powerful pattern recognition and polynomial-time inference, can learn effective heuristics and strategies to either solve problems directly or guide traditional solvers more efficiently, potentially offering a new paradigm for tackling these foundational computational challenges.

The core of this investigation is to answer the following specific research questions:

- How proficient are state-of-the-art LLMs at directly solving instances of NP-Complete problems?
- What is the relationship between key parameters of an NP-Complete problem instance and the likelihood of an LLM correctly solving that instance?
- For specific classes of problem instances, are LLMs able to arrive at a correct solution faster than a traditional, highly-optimized SMT solver like Z3?
- Does fine-tuning a general-purpose LLM on a dataset of NP-Complete problem instances

improve its generalizability on unseen instances of the same problem, potentially creating a more efficient specialized solver?

2 Related work

The challenge of solving NP-class problems has spurred decades of research across computational complexity theory, heuristic algorithms, and artificial intelligence. While no known polynomial-time algorithms exist for NP-complete problems, numerous approximate and heuristic-based methods have been developed to tackle them with varying levels of success.

Traditional approaches such as backtracking, branch-and-bound, and dynamic programming offer exact solutions but suffer from exponential time complexity in worst-case scenarios. Heuristic and metaheuristic strategies, including genetic algorithms (GAs), simulated annealing, and ant colony optimization, have shown strong performance in approximating solutions for NP-complete problems. For instance, Goldberg and Lingle (1985) and Grefenstette and Gucht (1985) have compared different representations in Genetic Algorithms for the Traveling Salesman Problem. While effective, these approaches often rely on handcrafted representations and problem-specific genetic operators, limiting their generalizability and adaptability.

With the growth usage of neural networks in learning patterns and heuristics, many including Hanjun Dai and Song (2018) and Marcelo Prates and Vardi (2018) have combined graph neural networks with reinforcement algorithms for bridging differential programming and combinatorial domains. They were able to solve many problems such as The Traveling Salesman Problem with little error.

Recently, many works have focused on LLMs and their reasoning capabilities. By focusing on prompt engineering and using RL algorithms it has been shown that with methods like forcing the model to produce their chain of thoughts (Jason Wei and Zhou, 2023), LLMs can solve many complex tasks requiring step by step reasoning.

Therefore, recent works have tried to utilize the reasoning capabilities of LLMs in finding suboptimal solutions to NPC problems. Chengrun Yang and Chen (2024) from Google DeepMind have proposed OPRO to leverage LLMs as optimizers, where the optimization task is described in natural language and showcase OPRO on linear regression and traveling salesman problems. In their method in each optimization step, the LLM generates new solutions from the prompt that contains previously generated solutions with their values, then the new solutions are evaluated and added to the prompt for the next optimization step. Haoran Ye and Song (2024) presented ReEvo an integration of evolutionary search for efficiently exploring the heuristic space, and LLM reflections to provide verbal gradients within the space. Chang Gong and Zheng (2025) introduced PIE a framework LLMs are tasked with understanding the problem and extracting relevant information to generate correct code and be able to tackle graph computational tasks. These works have shown that just using LLMs for generating solutions based on long graph inputs would not yield proper solutions also asking the LLMs to generate solutions by generating an algorithm causes the LLM to generate brute force algorithms with exponential running time.

Recently, researchers at DeepSeek AI have introduced an RL method, called GRPO (Zhihong Shao, 2024), and by combining the algorithmic reward functions with very large LLM models trained on huge datasets, they were able to train a reasoning model (DeepSeek-AI, 2025) and gain significant improvements in math, reasoning and coding tasks. As far as we know no one has ever analyzed the impact of GRPO on solving NPC problems and our work will be the first to fine-tune a model using GRPO.

3 Preliminaries

In computational complexity theory, the class NP consists of decision problems for which a proposed solution (or "certificate") can be *verified* in polynomial time by a deterministic Turing machine. A problem is classified as **NP-Complete** (NPC) if it is in NP and every other problem in NP can be reduced to it in polynomial time. This makes NP-Complete problems the most computationally difficult problems within NP; a polynomial-time algorithm for any single NP-Complete problem would imply a polynomial-time algorithm for all problems in NP.

We now explain the six NP complete problems we chose for our investigation:

3.1 3-SAT

The Boolean Satisfiability Problem (SAT) is the first problem proven to be NP-Complete by the Cook-Levin theorem, establishing it as a cornerstone of computational complexity theory. The problem is defined as follows:

Given a Boolean formula ϕ composed of variables, logical AND ($\wedge$), OR ($\vee$), and NOT ($\neg$), does there exist an assignment of the variables $\{X_1, X_2, ..., X_n\}$ to the values $\{\text{TRUE}, \text{FALSE}\}$ such that the entire formula ϕ evaluates to TRUE? If such an assignment exists, the formula is *satisfiable*; otherwise, it is *unsatisfiable*.

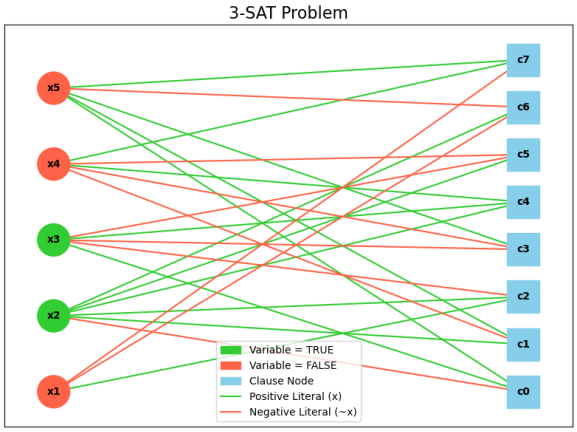


Figure 1: Visualization for a 3-SAT problem instance

A highly common and useful restriction is to formulas in *conjunctive normal form* (CNF). A formula in CNF is a conjunction (AND) of one or more *clauses*, where a clause is a disjunction (OR) of one or more *literals*. A literal is either a variable or its negation.

3-SAT is the special case of SAT where the formula is in CNF and each clause is limited to *exactly three literals*. This restriction remains NP-Complete and is often used in theoretical and empirical studies due to its structured nature.

3.2 Subset Sum Problem

The subset sum problem (SSP) is another NP-Complete decision problem. It is formally defined as follows:

Given a multiset (or set) of integers $S = \{a_1, a_2, ..., a_n\}$ and a target integer T, does there exist a subset $S' \subseteq S$ such that the sum of the elements in S' is exactly equal to T?

$$\sum_{a_i \in S'} a_i = T$$

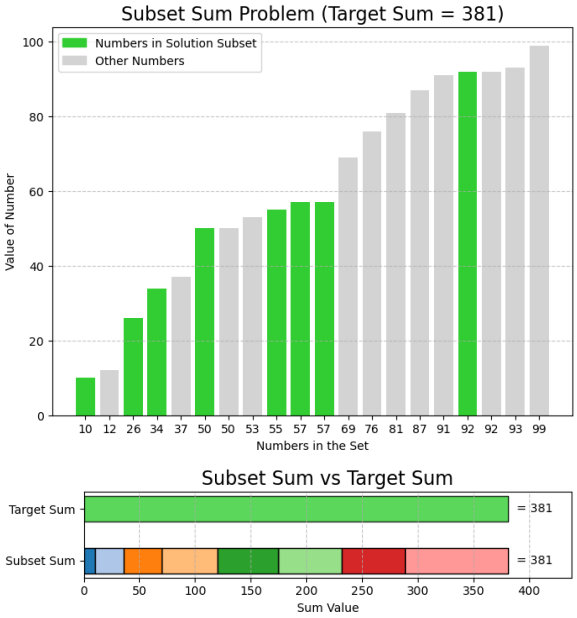


Figure 2: Visualization for a SSP problem instance

A crucial nuance of SSP is that it is considered *weakly* NP-Complete. This means it can be solved in *pseudo-polynomial time* using a dynamic programming algorithm whose runtime depends on the magnitude of the target T and the number of elements n. The standard dynamic programming solution has a time complexity of $O(n \cdot T)$. This makes the problem tractable for instances where T is relatively small, but intractable for instances with very large target values, as the runtime becomes exponential in the number of bits needed to represent T.

3.3 Minimum Vertex Cover Problem

In graph theory, a **vertex cover** of an undirected graph $G = (V, E)$ is a set of vertices $C \subseteq V$ such that for every edge $(u, v) \in E$, at least one of its endpoints (u or v) is contained in C. Informally, the set C "covers" all the edges of the graph.

The Vertex Cover problem is one of Karp's 21 NP-complete problems and is often used in complexity proofs. It is formally defined as follows:

Given an undirected graph $G = (V, E)$ and a positive integer $k \leq |V|$, does there exist a vertex cover C of size at most k?

Here we chose the optimization variation of the problem that the only input is the graph and the task is to find the vertex cover with the minimum number of vertices.

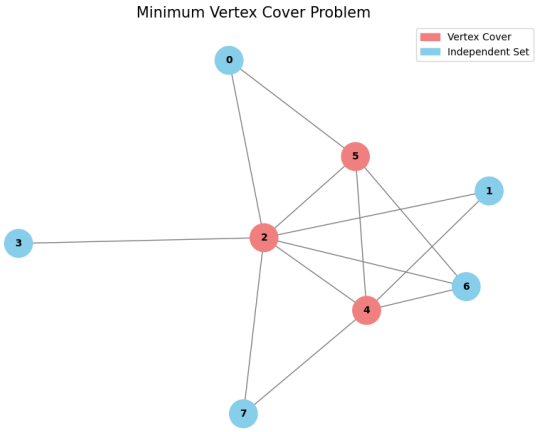


Figure 3: Visualization for a VC problem instance

3.4 Maximum Clique Problem

In graph theory, a **clique** is a subset of vertices within an undirected graph $G = (V, E)$ such that every two distinct vertices in the subset are adjacent; that is, the induced subgraph is complete. A clique of size k is often called a *k-clique*. The problem is formally defined as follows:

Given an undirected graph $G = (V, E)$ and a positive integer $k \leq |V|$, does there exist a set of vertices $C \subseteq V$ of size $|C| \geq k$ that forms a clique?

Same as Vertex Cover for the Clique problem we only give the graph as input and ask the LLM to find the Clique with maximum number of vertices in the graph.

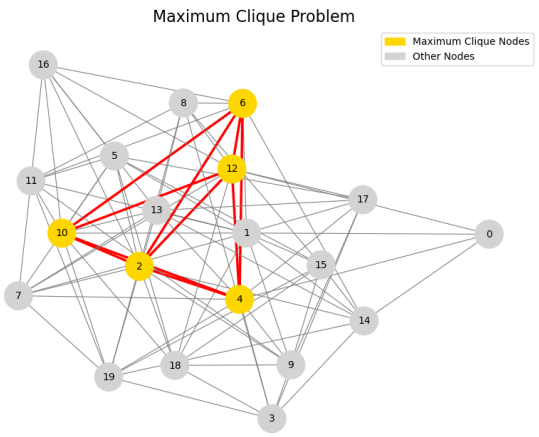


Figure 4: A solution to a CLIQUE problem instance

3.5 Hamiltonian Path Problem

In graph theory, a **Hamiltonian path** is a path in an undirected or directed graph $G = (V, E)$ that visits each vertex exactly once. Determining whether such a path exists is a classic and computationally difficult problem. The problem is formally defined as follows:

Given a graph $G = (V, E)$ (which may be directed or undirected), does there exist a sequence of vertices $\langle v_1, v_2, ..., v_{|V|} \rangle$ such that every vertex $v \in V$ appears exactly once in the sequence and each consecutive pair of vertices (v_i, v_{i+1}) is an edge in E?

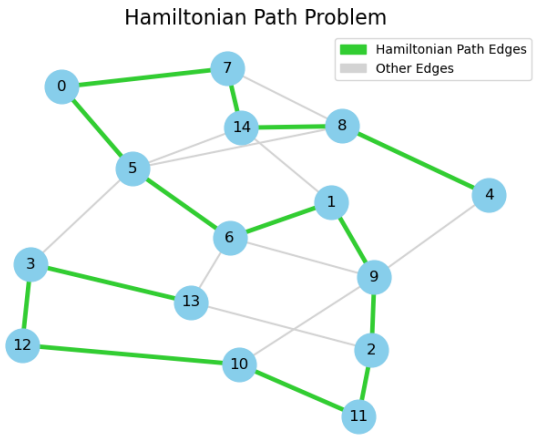


Figure 5: Visualization for a HAMPATH problem instance

3.6 Hamiltonian Cycle Problem

The Hamiltonian Cycle Problem is a close relative of the Hamiltonian Path Problem. In graph theory, a **Hamiltonian cycle** (or Hamiltonian circuit) is a cycle in an undirected or directed graph $G = (V, E)$ that visits each vertex exactly once and returns to the starting vertex. The problem is formally defined as follows:

Given a graph $G = (V, E)$, does there exist a cycle that visits every vertex in V exactly once, excluding the repetition of the starting and ending vertex? More formally, does there exist a sequence of vertices $\langle v_1, v_2, ..., v_{|V|}, v_1 \rangle$ such that every vertex $v \in V$ appears exactly once in the internal sequence $\langle v_1, v_2, ..., v_{|V|} \rangle$, and each consecutive pair (v_i, v_{i+1}) for $1 \leq i < |V|$ and the pair $(v_{|V|}, v_1)$ are edges in E?

4 Dataset

A significant challenge in evaluating Large Language Models on NP-Complete problems is the scarcity of large, and diverse benchmarks. To ensure a controlled and comprehensive evaluation across all six selected problems, we procedurally generated our own datasets. For each problem, we

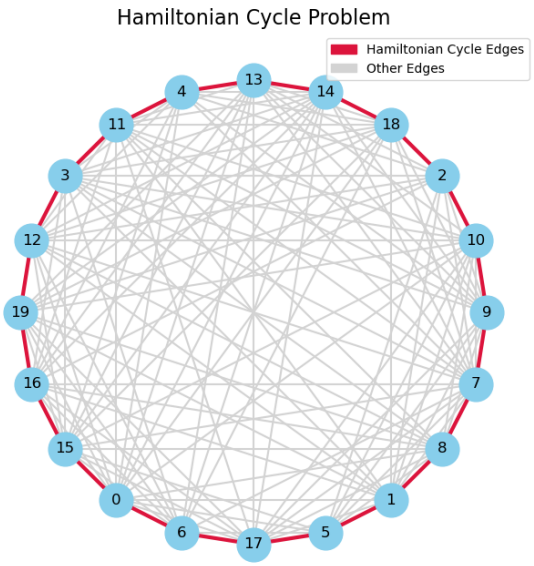


Figure 6: Visualization for a HAMCYCLE problem instance

created a set of instances along with a corresponding solution. We should note that the generated solution is not necessarily the only solution to the problem instance.

Generating random problem instances is trivial; however, generating instances that are *guaranteed to be satisfiable* requires a more nuanced approach. Simply generating random instances often results in unsatisfiable instances. Our algorithms, ensure satisfiability by first generating a random solution and then constructing an instance that is satisfied by it.

Moreover along with datasets, for each problem we wrote a function to verify a given solution, a function to visualize a given solution and finally a function to solve the problem instance with Z3. This section details the algorithm used to generate satisfiable instances for each problem.

4.1 3-SAT Instance Generation

The algorithm proceeds in three main steps:

1. **Solution Generation:** A random truth assignment is generated for all variables. This assignment serves as the target solution that will satisfy the final formula.

2. **Clause Construction:** For each clause, three distinct variables are randomly selected. Each variable is negated or left positive with equal probability, forming a initial clause.

3. **Satisfiability Guarantee:** The initial clause is checked against the pre-defined solution. If the clause is not satisfied, the polarity of one randomly chosen literal within the clause is flipped. This corrective action ensures the clause is satisfied by the target solution.

This method provides fine-grained control over the distribution of the dataset through parameters such as the number of variables (n), the number of clauses (m), and the bias towards `TRUE` or `FALSE` in the solution.

4.2 Subset Sum Problem Instance Generation

The algorithm proceeds in four main steps:

1. **Solution Subset Generation:** The size of the solution subset is determined as a fraction of the total set size. A collection of random integers within a specified range $[min\_val, max\_val]$ is generated to form this subset.

2. **Target Sum Calculation:** The target sum T is simply calculated as the sum of all integers in the pre-defined solution subset.

3. **Population of Remaining Set:** The remaining integers needed to complete the full set are generated randomly within the same value range.

4. **Randomization:** The solution elements and the remaining elements are combined and shuffled to form the final set S.

This method provides control over the problem's difficulty through parameters such as the total set size (set_size), the solution subset ratio ($subset\_ratio$), and the value range of the integers (min_val, $range\_val$), which influences the magnitude of the target sum T.

4.3 Minimum Vertex Cover Instance Generation

The algorithm proceeds in four main phases:

1. **Vertex Partitioning:** The set of vertices is partitioned into two subsets: a predefined minimum vertex cover C of size $k = \lfloor n \times vc\_ratio \rfloor$, and the remaining vertices, which form an independent set I. By definition, every edge must have at least one endpoint in C.

2. **Forcing Cover Necessity:** To ensure the set C is necessary for a cover, an edge is added from every vertex in C to a randomly chosen vertex in the independent set I. This guarantees that no vertex in C can be removed without leaving an edge uncovered.

3. **Graph Population:** Additional edges are added probabilistically according to a specified $edge\_density$ parameter. Crucially, no edges are added between vertices within the independent set I to preserve its independence property and the minimality of C.

4. **Result:** The algorithm returns the generated graph G and the known minimum vertex cover C, which serves as a solution for evaluation.

This method provides fine-grained control over the problem instance through parameters such as the number of vertices ($num\_vertices$), the size of the vertex cover as a ratio of the graph size (vc_ratio), and the general connectivity of the graph ($edge\_density$).

4.4 Maximum Clique Problem Instance Generation

The algorithm proceeds in five main phases:

1. **Initialization and Partitioning:** The vertex set is partitioned into two groups: a predefined clique C of size $k = \lfloor n \times clique\_ratio \rfloor$, and the remaining vertices O.

2. **Clique Construction:** A complete graph (a clique) is explicitly constructed on the vertex set C by adding all possible edges between its members.

3. **Independent Set Simulation:** To prevent the formation of a clique larger than k, the remaining vertices O are distributed into k different partitions. The algorithm then ensures that no edge is added *within* each partition, effectively making each partition an independent set. This guarantees that any clique can contain at most one vertex from each partition.

4. **Inter-Partition Edge Addition:** Edges are added probabilistically between vertices in O that belong to *different* partitions, according to the $edge\_density$ parameter. This creates connectivity without violating the partition constraints that cap the clique size.

5. **Connecting Clique to Other Vertices:** Each vertex u in the known clique C is connected to vertices in O that are *not* in its corresponding "forbidden" partition. This ensures that u cannot form a larger clique with any vertex from its own forbidden partition, preserving C as the maximum clique.

This method provides control over the problem's nature through the size of the planted clique ($clique\_ratio$) and the general connectivity of the graph ($edge\_density$).

4.5 Hamiltonian Path Instance Generation

The algorithm proceeds in three main phases:

1. **Backbone Construction:** A random permutation of all vertices is generated. This permutation defines the sequence of the Hamiltonian path. Edges are then explicitly added between consecutive vertices in this sequence, forming the guaranteed Hamiltonian path 'backbone' of the graph.

2. **Path Edge Tracking:** The edges that make up this initial path are stored in a set. This is crucial to avoid duplicate edges in the next phase and to ensure the graph remains simple.

3. **Graph Augmentation:** Additional edges are added probabilistically between all non-adjacent vertex pairs according to a specified $edge\_density$ parameter.

This method provides control over the problem instance through the number of vertices ($num\_vertices$) and the general connectivity ($edge\_density$), which influences how well-hidden the Hamiltonian path is among other potential paths.

4.6 Hamiltonian Cycle Instance Generation

The algorithm proceeds in three main phases:

1. **Cycle Backbone Construction:** A random permutation of all vertices is generated. This permutation defines the sequence of the Hamiltonian cycle. Edges are explicitly added between consecutive vertices in this sequence. Crucially, an additional edge

is added between the first and last vertex in the sequence, transforming the path into a cycle and ensuring the graph is Hamiltonian.

2. **Cycle Edge Tracking:** The edges that make up this initial cycle are stored in a set. This prevents the addition of duplicate edges in the subsequent phase and ensures that that the graph remains simple.

3. **Graph Augmentation:** Additional edges are added probabilistically between all non-adjacent vertex pairs according to a specified $edge\_density$ parameter.

This method provides control over the problem instance through the number of vertices ($num\_vertices$) and the general connectivity ($edge\_density$), which influences how well-hidden the Hamiltonian cycle is within the graph's structure.

5 Baselines

5.1 Z3 Theorem Prover

The Z3 Theorem Prover(Leonardo de Moura, 2008) is a high-performance satisfiability modulo theories (SMT) solver developed by Microsoft Research. It is designed to solve complex constraints and is widely used in applications such as software verification, symbolic execution, and advanced program analysis. As an *exact solver*, Z3 is guaranteed to find a solution if one exists, or definitively prove that no solution exists, for the problems it supports.

For each of the six NP-Complete problems in our study, we developed a formal model using the Z3 Python API. These models precisely encode the constraints of each problem instance. It is important to note that while Z3 provides exact answers, its worst-case time complexity remains exponential for NP-Complete problems. The central hypothesis of our work is that LLMs, with their polynomial-time inference, could potentially serve as efficient heuristic solvers, so by increasing their accuracy in solving problems our goal is to reach a model with high accuracy and polynomial run time.

5.2 Base LLM

Our second baseline is a state-of-the-art large language model, chosen for its optimal balance of performance, cost, and inference speed. We selected Gemini 2.5 Flas. While more powerful reasoning models exist (e.g., Gemini 2.5 Pro, GPT-4o), their significantly higher computational cost and latency make them unsuitable for this task as Z3 can solve the problems much faster. Gemini 2.5 Flash provides a strong representative of the current capabilities of fast, commercially available LLMs.

Our evaluation of this baseline consisted of a zero-shot prompting strategy. For each problem instance, we designed a natural language prompt that described the problem and explicitly instructed the model to provide its final answer in a structured, parseable format (e.g., `"[a, b, c, ...]"` for Subset Sum). The performance of this baseline, as detailed in Section 7.1, establishes the inherent ability of a powerful general-purpose LLM to solve complex computational problems without any specialized training or tool augmentation.

6 Methodology

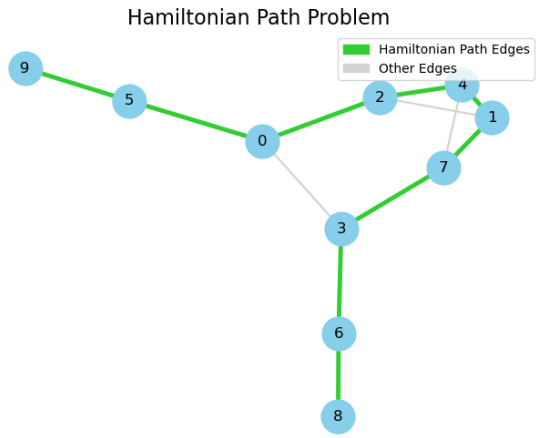
Following the creation of the synthetic dataset and the evaluation of the baseline Gemini model, we selected the Hamiltonian Path Problem as the primary focus for enhancement. as it along with Hamiltonian Cycle were the only problems that Z3 was not able to solve faster than LLM. This section details our approach to improving LLM performance.

6.1 Training

Our initial challenge was the high token cost of representing graph structures textually. To maximize the number of vertices we could process within a finite context window, we focused on graphs with up to 100 vertices and pursued two optimizations: prompt compression and tokenizer selection.

We designed a short prompt that provided essential task instructions alongside a densely encoded graph representation. The graph was represented by listing each vertex's label followed by a colon and a concatenated string of its neighbors' labels, without any spaces or other characters. This design minimized the number of tokens.

We evaluated several state-of-the-art model families based on their tokenizers' efficiency in encoding numerical sequences. Models like Gemma and Qwen tokenized each digit individu-




```
You are an expert in graph theory
    specializing in Hamiltonian
Path Problem. You receive an undirected
    graph where
each line contains adjacent vertices for
     vertex. Your must
return a path that visits every vertex
    exactly once.
You must provide the sequence of
    vertices and no additional text.
Graph:
000:002003005
001:002004007
002:000001004
003:000006007
004:001002007
005:000009
006:003008
007:001003004
008:006
009:005
Hamiltonian Path:
008006003007001004002000005009
```

Figure 7: (a) An example graph for the Hamiltonian Path problem. (b) The corresponding minimal prompt given to the model. Vertex labels are represented by three digits to ensure they are parsed as single tokens by the Llama tokenizer.

ally, leading to impractically long sequences (e.g., 20,000 tokens for a 100-vertex graph). In contrast, the Llama tokenizer often groups sequences of digits into single tokens. We selected the **Llama 3.2 3B Instruct** model for its favorable tokenization, which reduced the worst-case token count to 10,290 tokens for the input and 105 tokens for the output path. Due to hardware constraints on Google Colab (T4 GPU), we used the 4-bit quantized version of the model provided by the `unsloth` library to enable efficient training.

We generated a dataset of 1,000 graphs. The number of vertices for each graph was sampled from a uniform distribution between 5 and 100. The edge density for each graph was independently sampled from a uniform distribution between 0 and 1. This dataset was split, with 40% allocated for supervised fine-tuning (SFT) and the remaining 60% reserved for subsequent reinforcement learning (GRPO).

In the first stage, we performed supervised fine-tuning on the 400-instance SFT subset. The objective was to teach the model the desired output format and to prime its parameters on the Hamiltonian Path task. We applied LoRA (Low-Rank Adaptation) with a rank of 64 to all linear layers in the model. The model was trained for one epoch on this dataset.

Building on recent successes in using GRPO to improve reasoning capabilities, we also applied GRPO to the remaining 600 instances. GRPO aligns the model's output with a reward function that evaluates the quality of the proposed Hamiltonian path.

We defined two reward functions:

- **Simple Reward (R_1):** The reward is calculated as the number of correctly chosen vertices in the sequence divided by the total number of vertices. A correct solution yields a reward of 1.0. For example, an output path `5-0-7-3-6` for the graph in figure would receive a reward of $5/10 = 0.5$, as there is no edge connecting 0 to 7.

- **Strict Reward (R_2):** This more realistic function counts the number of consecutive correct vertices from the start of the path until the first error is encountered. The reward is this count divided by the total number of vertices. For the same erroneous path, the strict reward would be $2/10 = 0.2$, as the path `5-0` is valid, but the next step (`0` to `7`) is invalid as they are not connected.

An initial GRPO run was performed on a dataset sorted by number of vertices, which yielded high rewards and provided stable initial learning. Sorting the dataset resulted in higher reward values throughout training. A final GRPO training phase was conducted on a separate dataset of 570 graphs using the strict reward function (R_2) on an unsorted dataset to simulate a more realistic and challenging training environment.

6.2 Agent

We used the free API of Gemini 2.5 flash model to create our main agent node. This agent operates using a ReAct (Reasoning + Acting) paradigm (Shunyu Yao, 2023), utilizing a set of tools to critique and improve upon an initial candidate solution generated by the fine-tuned model.

The agent's primary tool is a path validation function. This tool is also made available to the agent itself, enabling it to perform self-critique and iterative improvement. The agent's workflow, is detailed in its prompt.

The core repair heuristic provided to the agent is a localized search algorithm based on the 2-OPT strategy, commonly used for routing problems. Given an input path, the tool iteratively explores the solution space by swapping pairs of vertices. Its objective is to minimize the number of "gaps"-i.e., consecutive vertices in the path that are not connected by an edge in the graph.

The algorithm operates with a time complexity of $O(C \cdot n^2)$, where n is the number of vertices and C is a constant defining the number of iterations (set to $C = 3$ in our experiments). This provides a efficient, polynomial-time heuristic for the agent to suggest improvements to flawed paths.

```
\label{lst:agent_prompt}
[caption={Prompt given to the agent node
    for HAMPATH problem}]
"You are a diligent assistant that finds
    a Hamiltonian Path using powerful
    tools.\n"
f"The graph is: {json.dumps(self.
    input_graph)}\n"
f"Your goal is to find a valid
    Hamiltonian Path, visiting all {self
    .num_vertices} vertices exactly once
    .\n\n"
"**Your Workflow:**\n"
"1. **Verify Initial Path**: You'll
    receive a proposed path. Immediately
    use the `verify_path` tool on it.\n
    "
"2. **If Correct**: Great! Call `
    get_final_answer` with the verified
    path to finish.\n"
"3. **If Incorrect**: Analyze the reason
    from `verify_path`.\n"
"   - If the path has **non-existent
    edges** but includes all vertices,
    use the `repair_path_with_2_opt`
    tool.\n"
"   - After getting a `repaired_path`
    from the tool, you MUST verify it
    again with `verify_path`.\n"
"4. **Iterate**: Always verify any new
    path. A solution is guaranteed to
    exist."
```

7 Experiments

This section details our presents the results of our empirical evaluation. We first established a performance baseline for a state-of-the-art LLM across all six NP-Complete problems. This initial analysis, detailed in Section 7.1, aimed to identify problems where the LLM showed potential for high speed but left room for accuracy improvement, making them suitable candidates for our enhancement techniques. Based on these results, we selected the Hamiltonian Path problem for further investigation and applied the fine-tuning and tool-augmentation approaches outlined in Section 6. The results of these interventions are presented in Section 7.1.

7.1 Base Model Evaluation

We generated a dataset of 100 unique instances for each of the six NP-Complete problems, varying key parameters (e.g., graph size, clause-to-variable ratio) to ensure a diverse and challenging benchmark. For each instance, we designed a structured natural language prompt that clearly defined the problem and requested a solution in a parseable format.

The prompts were batched and submitted to the Gemini 2.5 Flash API. The model's responses were automatically parsed and verified. To understand the factors influencing LLM performance, we conducted a multifaceted analysis:

- **Feature Correlation:** We generated 2D scatter plots for every pair of problem parameters, colored by success/failure, to visually identify potential relationships and decision boundaries.
- **Conditional Probability:** We discretized each parameter into 7 bins and generated bar plots to show the proportion of correctly solved instances within each bin, revealing how difficulty scales with specific parameter values.
- **Feature Importance:** We trained a Random Forest classifier (100 estimators, max depth of 5) on an 80% stratified sample of the data. The task was to predict, based on the problem parameters, whether the LLM would solve the instance correctly. We then analyzed the importance scores of each feature to quantify its influence on the model's success rate.

- **Running time analysis:** By measuring the time taken by the LLM and Z3 to solve the same set of problem instances we were able to decide which problems had the potential for further fine-tuning.

The primary goals of our baseline analysis were: first, to determine whether an LLM's success for an instance is predictable, and second, to compare its computational efficiency against the exact solver Z3 to identify problems worthy of further investment.

The performance of this Random Forest model is presented in the following table. The high accuracy and F1 scores across all problems demonstrate a strong predictive relationship between the problem parameters and the LLM's likelihood of success. This indicates that the LLM's performance is not random but is systematically influenced by the structural features of the problem instance.

Problem	Accuracy	F1 Score
3-SAT	0.8	0.8
Subset Sum	0.95	0.8
Vertex Cover	0.95	0.89
Clique	0.75	0.71
Hamiltonian Path	1.0	1.0
Hamiltonian Cycle	0.95	0.67

Table 1: Performance of the random forest model in predicting whether the base LLM can solve a specific instance of each problem.

Problem	Avg. LLM	Avg. Z3	Std. Z3
3-SAT	18.090	0.025	0.012
SSP	14.012	0.055	0.039
VC	22.781	5.930	54.387
CLIQUE	28.519	0.394	1.419
HPATH	22.306	-	-
HCYCLE	22.785	-	-

Table 2: Average time taken by the base LLM and the Z3 solver across 100 instances of each problem. Due to giving inputs to the LLM in batches the std of its time can't be computed. Moreover, because it took Z3 hours to solve 10 instances of HPATH and HCYCLE, we didn't compute solving time of Z3.

From Table 2 it is clear that Z3 is not efficient in solving the Hamiltonian Path and Cycle problems. Also interestingly, by analyzing the results of the base model, we found out these are much more difficult for the LLM to solve than the other problems. Therefore, we focused on improving the LLM's accuracy on the Hamiltonian Path problem, aiming to extend its reliable performance to larger and more complex graphs where its speed advantage would be most critical.

In the following subsections, we talk about the results for each problem in more detail.

7.1.1 3-SAT Problem

The analysis of the base LLM's performance on the 3-SAT problem revealed a clear and consistent relationship between problem structure and solvability.

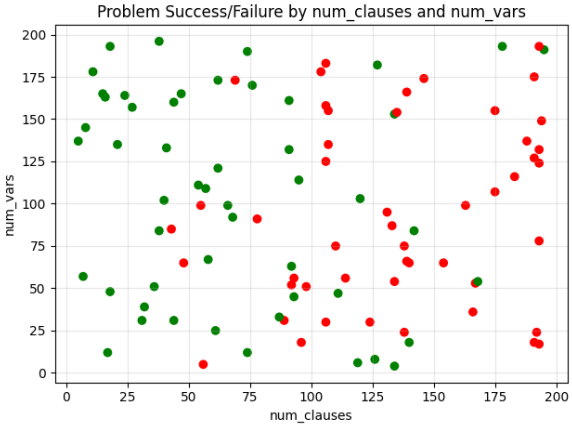


Figure 8: LLM success rate for 3-SAT instances, plotted by the number of clauses and variables.

Figure 8 illustrates that the LLM's success rate is highly dependent on the number of clauses in a formula, while the number of variables appears to have a comparatively minor effect. A potential explanation for this phenomenon is the correlation between the number of clauses and the number of input tokens presented to the LLM.

This relationship is further quantified in Figure 9, which shows a clear decreasing trend in the model's accuracy as the number of input tokens increases.

The conclusion that clause count is the primary determinant of difficulty is robustly supported by the feature importance analysis from our Random Forest classifier, shown in Figure 10. The number of clauses was found to be at least twice as important as the number of variables for predicting the LLM's success.

It is important to note that the number of input tokens was intentionally excluded as a feature for

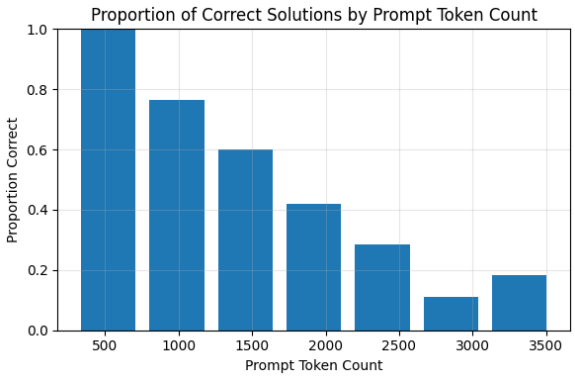


Figure 9: Proportion of correctly solved 3-SAT instances, binned by number of input tokens.

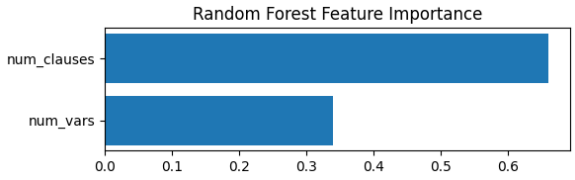


Figure 10: Random forest feature importance for 3-SAT problem

the classifier of all problems. Due to its high correlation with other features, its inclusion led to multicollinearity, which significantly decreased the interpretability and performance of the models.

It's very interesting to see that the parameter number of vertices which is the main parameter in finding the answer in brute force method and makes the search space exponential has less importance that the number of clauses.

7.1.2 Subset Sum Problem

The Subset Sum Problem (SSP) is classified as weakly NP-complete. This means its intractability is fundamentally tied to the *magnitude* of the target value which is influenced by minimum and range of the numbers and also size of the solution subset rather than solely the *count* of elements. Consequently, we hypothesized that the set size parameter had little correlation with the success rate.

Contrary to our expectation, the results revealed a different reality for the LLM. Figure 11 shows that the model success rate is predominantly influenced by the size of the set (that is, the number of elements, n). This mirrors our finding for 3-SAT (Section 7.1.1), indicating a broader pattern: the LLM's performance on NP-Complete problems is more sensitive to parameters that directly increase number of input tokens than to those that increase the underlying mathematical search space.

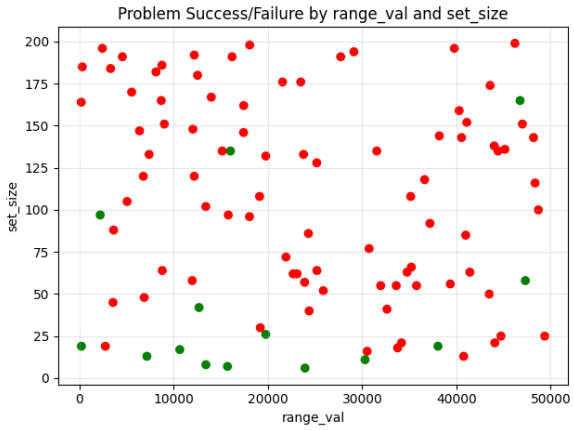


Figure 11: LLM success rate for SSP instances, plotted by the number of clauses and variables.

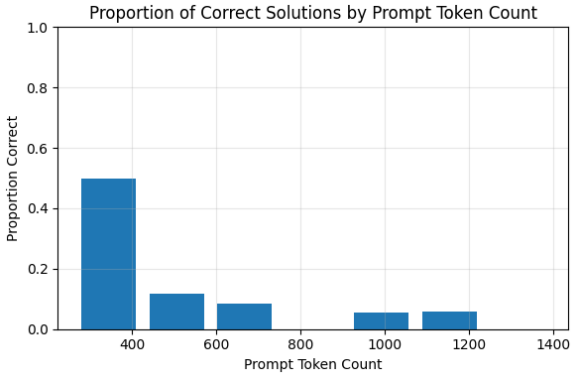


Figure 12: Proportion of correctly solved SSP instances, binned by number of input tokens.

This sensitivity to input scale is further emphasized in Figure 12. Although the number of input tokens for SSP are generally shorter than those for 3-SAT, the rate of accuracy degradation as number of tokens increases is more severe. Furthermore, even for short prompts the LLM fails to solve most of the instances while it is much more successful for 3-SAT instances.

The conclusions drawn from the visualizations are also confirmed by the feature importance analysis from our Random Forest classifier, shown in Figure 13. The results quantify the hierarchy of influence: the size of the full set (n) is the most important feature, followed by the size of the solution subset (k). The parameters governing the magnitude of the numbers (minimum value and range) were found to be least important, with an importance score roughly half that of the subset size.

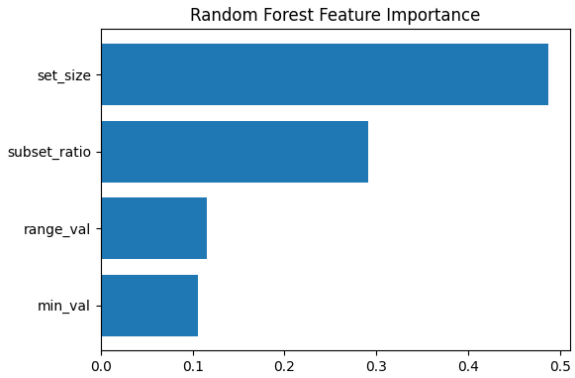


Figure 13: Random forest feature importance for SSP problem

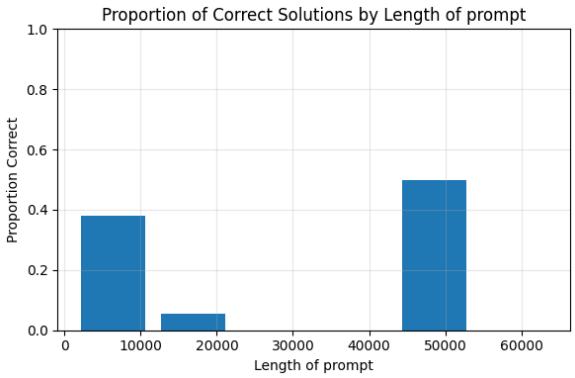


Figure 15: Proportion of correctly solved VC instances, binned by number of input tokens.

7.1.3 Minimum Vertex Cover Problem

The Minimum Vertex Cover (VC) problem presents a more complex relationship between its parameters and the LLM's performance. Unlike 3-SAT and Subset Sum, where a single parameter dominated, in VC all key parameters—number of vertices ($|V|$), the size of the minimum vertex cover (k), and edge density—influence the structure of the problem and the length of its textual description.

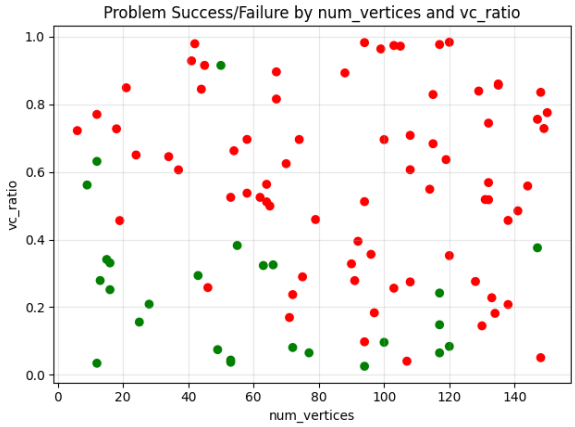


Figure 14: LLM success rate for VC instances, plotted by the number of vertices and ratio of vertices in the minimum vertex cover.

Figure 14 indicates that the LLM's success rate is more sensitive to the relative size of the vertex cover ($k/|V|$) than to the absolute number of vertices.

The relationship between prompt token count and accuracy as illustrated in Figure 15is **non-monotonic** and does not follow the simple negative correlation observed in 3-SAT and SSP. Instead, we observe a sudden increase in accuracy for prompts of length around 5000. This may suggest that for Minimum Vertex Cover problem, the LLM's performance is not solely determined by number of input tokens but is highly dependent on the specific *structural properties* of the graph.

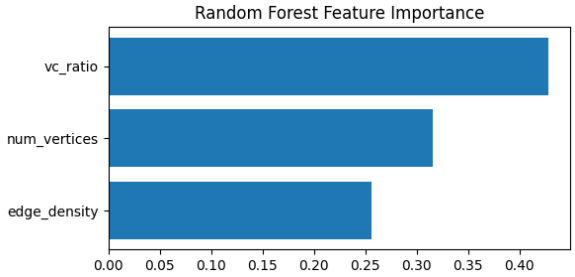


Figure 16: Random forest feature importance for VC problem

The feature importance analysis, shown in Figure 16, provides a quantitative hierarchy for the parameters influencing the LLM's performance. The results indicate that the size of the minimum vertex cover (k) is the most important predictor of difficulty, followed by the number of vertices ($|V|$), with edge density being the least important of the three. This suggests that the core challenge for the LLM is finding a solution of a specific size, with the overall graph scale being a secondary factor, and the graph's connectivity playing a less critical role.

Same as the previous two questions here the number of vertices which is the parameter making the search space exponential is less important than the size of the minimum vertex cover.

7.1.4 Maximum Clique Problem

The Maximum Clique (CLIQUE) problem presents a unique case where the primary parameter governing computational complexity—the

number of vertices ($|V|$)—is also the main driver of input prompt length. This confluence makes it challenging to disentangle whether the LLM's difficulty stems from the inherent exponential growth of the search space or from the increasing complexity of processing a longer textual description.

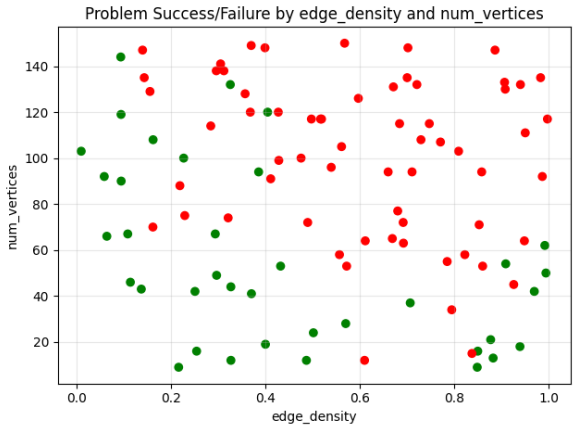


Figure 17: LLM success rate for CLIQUE instances, plotted by the number of vertices and edge density.

Figure 17 indicates that the LLM's success rate decreases with both an increasing number of vertices and higher edge density. This suggests that the model struggles with larger, more connected graphs.

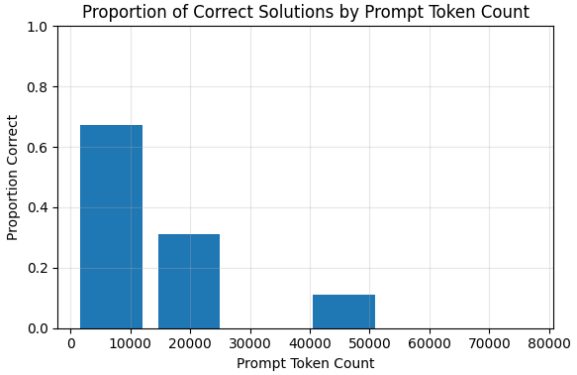


Figure 18: Proportion of correctly solved CLIQUE instances, binned by number of input tokens.

Mirroring the trends observed in 3-SAT and SSP, Figure 18 shows a clear negative correlation between the count of input tokens and the accuracy of the LLM. This consistent pattern across multiple problem domains reinforces the hypothesis that the number of input tokens is a significant factor influencing the performance of LLM in combinatorial problems.

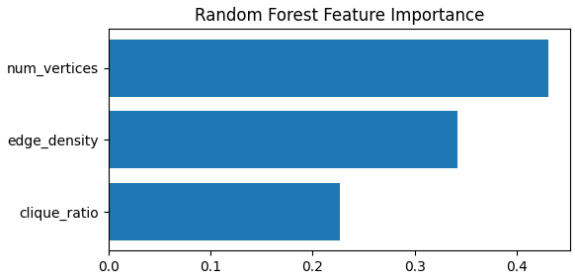


Figure 19: Random forest feature importance for CLIQUE problem

Unlike the previous problems here the most important feature (number of vertices) is also the feature that makes the search space exponential. However, it is also the main feature for increasing prompt length. The next important feature is edge density followed by size of the maximum clique.

The feature importance analysis, shown in Figure 19, reveals the number of vertices ($|V|$) is the most important predictor of difficulty, followed by edge density, with the size of the maximum clique being the least important. Unlike the previous problems, the most important feature here is also the one that dictates the exponential growth of the classical search space. However, since this parameter is also the primary contributor to prompt length, it remains an open question whether the LLM's performance degradation is due to computational intractability or context window limitations.

7.1.5 Hamiltonian Path Problem

The Hamiltonian Path (HAMPATH) problem proved to be the most challenging for the base LLM, with performance worse than on the other NP-Complete problems studied. In HAMPATH, both the number of vertices ($|V|$) and the edge density heavily influence the count of the input tokens.

Figure 20 reveals that the LLM's success rate exhibits a sharp decline as the number of vertices increases, effectively falling to zero for instances with more than 30 vertices.

The feature importance analysis, shown in Figure 21, indicates that the number of vertices has significantly higher importance than edge density.

7.1.6 Hamiltonian Cycle Problem

The results for the Hamiltonian Cycle Problem is very similar to the Hamiltonian Path problem so here we just plot the corresponding plots which

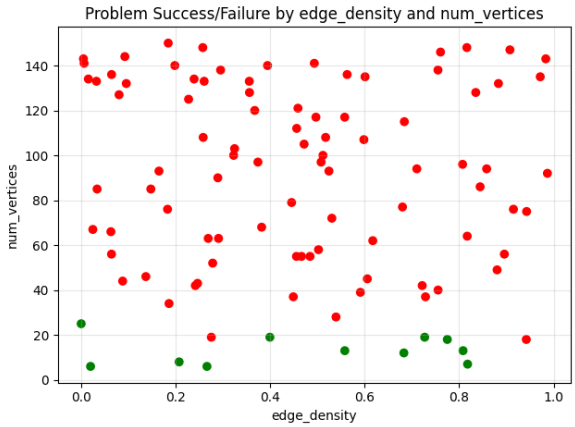


Figure 20: LLM success rate for HAMPATH instances, plotted by the number of vertices and edge density.

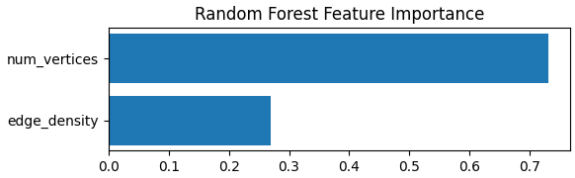


Figure 21: Random forest feature importance for HAMPATH problem

the same explanations for the Hamiltonian Path applies here as well.

The results for the Hamiltonian Cycle (HAMCYCLE) problem closely mirror those of the Hamiltonian Path problem, reinforcing the conclusion that global graph connectivity poses a significant challenge for the base LLM. The same parameters—number of vertices ($|V|$) and edge density—govern both number of input tokens.

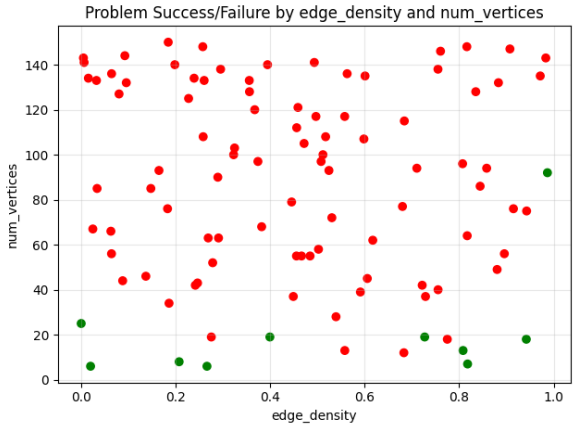


Figure 22: LLM success rate for HAMCYCLE instances, plotted by the number of vertices and edge density.

As shown in Figure 22, the LLM's performance on HAMCYCLE exhibits the same strong inverse relationship with graph size that was observed for HAMPATH. The success rate drops precipitously for instances with more than approximately 30 vertices, confirming that the model's reasoning capacity is overwhelmed by the complexity of identifying a single cycle that traverses all nodes in a larger graph.

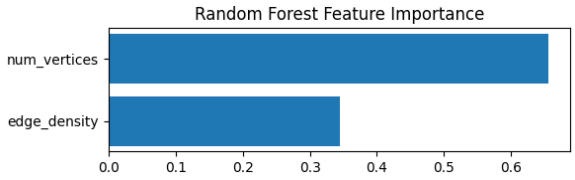


Figure 23: Random forest feature importance for HAMCYCLE problem

The feature importance analysis for HAMCYCLE, presented in Figure 23, further validates this parallel. The number of vertices ($|V|$) is more important predictor of LLM failure, compared to edge density.

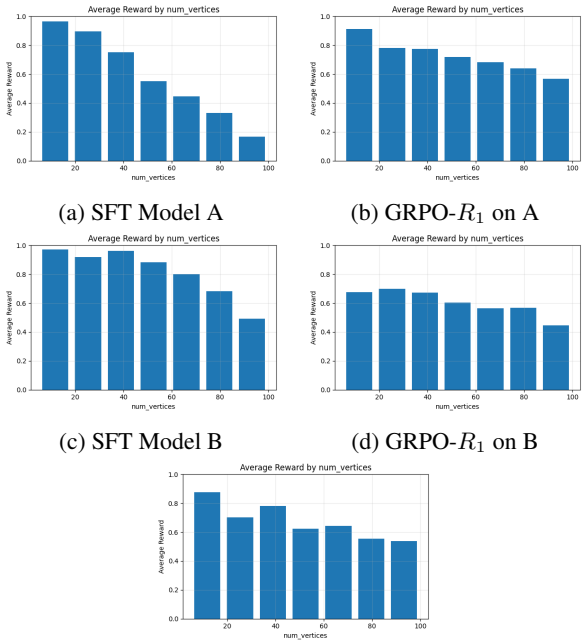
7.2 Training

The performance of our fine-tuned models, as evaluated on a separate test set of 100 HAMPATH instances, fell significantly short of the base Gemini model. The results for our various training configurations are summarized in Figure 24.

The overall accuracy was low across all models. Model 24a (SFT-only) solved only 7 out of the 100 test instances. Subsequent models, including those trained with GRPO, performed even worse, solving fewer than 5 instances. We hypothesize that the primary reason for this underwhelming performance is the limited size of our training dataset, which may have been insufficient for the model to learn the complex reasoning required for the Hamiltonian Path problem. The further performance degradation observed with GRPO training may be due to the same data limitation, hindering effective policy exploration in the reinforcement learning phase.

Model 24c, trained on a larger dataset consisted of graphs with 40-60 vertices, achieved higher average reward values than other configurations. This offers support for our hypothesis that investing in a larger dataset could lead to improved performance.

Given their limited standalone capability, we repurposed these fine-tuned models as *initial path*



(a) SFT Model A

(b) GRPO-R_1 on A

(c) SFT Model B

(d) GRPO-R_1 on B

(e) GRPO-R_2 on Model B

Figure 24: Training plots for different models. (a) Model trained only with Supervised Fine-Tuning (SFT). (b) Model (a) further trained with GRPO using the R_1 reward function on sorted dataset. (c) SFT model trained on a dataset of twice the size of previous models with 40-60 vertices. (d) Model (c) further trained with GRPO (R_1) on sorted dataset. (e) Model (b) further trained with GRPO using the stricter R_2 reward function (Equation 6.1)

generators for our agent-based refinement system, which is described in Section **??**.

7.3 Agent

This section presents the results of applying the agentic methodology outlined in Section 6.2. We evaluated the performance of an LLM agent equipped with a solution repair tool, first using the base Gemini model and subsequently using our fine-tuned models as the initial solution generator.

Initially, we investigated the performance of the Gemini-powered agent without our custom-trained model. This agentic setup could solve most problem instances with fewer than 50 vertices, representing a significant improvement over the base Gemini model operating without tools. However, the agent failed to generate any output for the majority of graphs with more than 50 vertices.

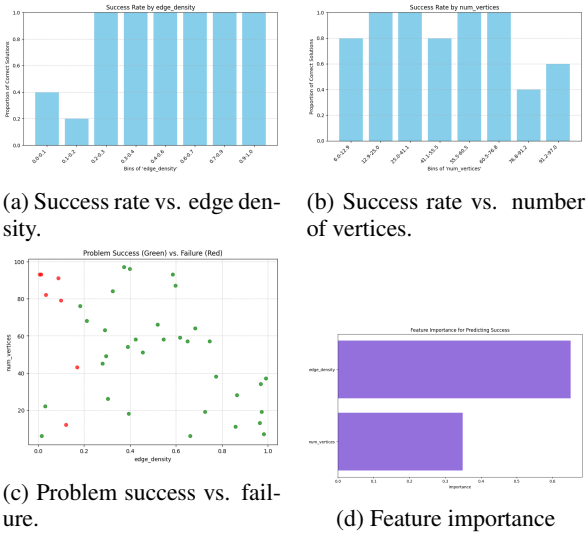
To address this limitation, we integrated our fine-tuned models as the initial solution generators within the agent framework.

Using the base SFT model (Model A, see Section 24a) as the generator, the agent's performance improved markedly. It successfully solved most instances with up to 100 nodes, achieving an overall accuracy of **81.4%**.

Replacing the generator with the GRPO-trained model (Model E, see Section **??**) yielded a further improvement, increasing the overall accuracy to **82.5%**. This result confirms that the reinforcement learning phase had a positive, albeit modest, effect on the quality of the initial solutions provided to the repair tool.

The agent with Model E average solve time was 129.57 seconds with a standard deviation of 160.13 seconds. While this variance indicates some instances were challenging, the performance is orders of magnitude faster than the hours-long runtime experienced by the Z3 solver, representing a significant practical improvement.

A pivotal observation from this experiment concerns the factors influencing problem difficulty. The feature importance analysis (Figure 40d, Subfigure D) reveals a shift compared to the base model's behavior. For the agent, **edge density** emerged as a more important predictor of failure than the number of vertices. This suggests that the inherent difficulty of a problem instance is not absolute but is instead dependent on the solving methodology.



(a) Success rate vs. edge density.

(b) Success rate vs. number of vertices.

(c) Problem success vs. failure.

(d) Feature importance

Figure 25: Performance analysis of the LLM agent using the GRPO-trained model (Model E) as an initial solution generator.

8 Contributions of group members

We collaborated closely and supported each other on various tasks, regularly providing feedback. However, the general responsibilities of each member were as follows:

- Mohammad Taha Majlesi: Contributed to analyzing the performance of Z3, building the agents, and training the models.
- Mahdi Naeni: Created the datasets and verifiers, analyzed Z3's performance, built the agents, and assisted in evaluating the trained models.
- Babak Hosseini Mohtasham: Analyzed the performance of the base LLM model, trained models using SFT and GRPO, and evaluated them.

9 Conclusion

We aimed to pursue a unique project that had not been done before while also making a contribution to the computer science community. We believe the project accomplished most goals.

Our investigation yielded several key findings. We confirmed that state-of-the-art LLMs can, in fact, solve instances of certain NP-Complete problems—notably the Hamiltonian Path problem—faster than the highly optimized Z3 solver on larger instances where Z3's exponential worst-case complexity manifests. This demonstrates that LLMs can serve as effective heuristic solvers, trading off marginal decreases in guaranteed accuracy for monumental gains in speed on appropriate problem classes.

The project also surfaced significant challenges, primarily the substantial token overhead required to represent graph problems textually. This necessitated extensive prompt engineering and a careful selection of model architecture based on tokenizer efficiency.

Looking forward, this work opens up several promising avenues for future research:

- **Scaled Training:** With access to greater computational resources, training on larger and more diverse datasets could yield a more accurate NP complete solver.
- **Inference Optimization:** Applying advanced inference techniques like speculative decoding or layer skipping (Mostafa Elhoushi, 2024) could further reduce the LLM's runtime, solidifying its speed advantage over traditional solvers.
- **Problem Characterization:** A deeper analysis is needed to identify the common properties of problems where LLMs excel. This could lead to a hybrid architecture where a classifier directs problems to either a fast LLM heuristic or a precise but slower solver like Z3, creating a system that is both fast and exact.

In conclusion, while LLMs are not replacements for exact solvers, they represent a powerful new tool in the computational toolbox. This project provides a detailed analysis on accuracy and speed of LLMs in solving NP complete problems along with a framework to improve the performance of the LLMs on these problems.

References

Chang Gong, Wanrui Bian, Z. Z. and Zheng, W. (2025). Pseudocode-injection magic: Enabling llms to tackle graph computational tasks.

Chengrun Yang, Xuezhi Wang, Y. L. H. L. Q. V. L. D. Z. and Chen, X. (2024). Large language models as optimizers.

DeepSeek-AI (2025). Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning.

Goldberg and Lingle (1985). Comparative study of different representations in genetic algorithms for job shop scheduling problem.

Grefenstette, R. G. and Gucht, D. V. (1985). Comparative study of different representations in genetic algorithms for job shop scheduling problem.

Hanjun Dai, Elias B. Khalil, Y. Z. B. D. and Song, L. (2018). Learning combinatorial optimization algorithms over graphs.

Haoran Ye, Jiarui Wang, Z. C. F. B. C. H. H. K. J. P. and Song, G. (2024). Reevo: Large language models as hyper-heuristics with reflective evolution.

Jason Wei, Xuezhi Wang, D. S. M. B. B. I. F. X. E. H. C. Q. V. L. and Zhou, D. (2023). Chain-of-thought prompting elicits reasoning in large language models.

Leonardo de Moura, N. B. (2008). Z3: an efficient smt solver.

Marcelo Prates, Pedro Avelar, H. L. L. C. L. and Vardi, M. Y. (2018). Learning to solve np-complete problems: A graph neural network for decision tsp.

Mostafa Elhoushi, Akshat Shrivastava, D. L. B. H. B. W. L. L. A. M. B. A. S. A. A. R. A. A. A. B. C. C. J.-W. (2024). Layerskip: Enabling early exit inference and self-speculative decoding.

Shunyu Yao, Jeffrey Zhao, D. Y. N. D. I. S. K. N. Y. C. (2023). React: Synergizing reasoning and acting in language models.

Zhihong Shao, Peiyi Wang, Q. Z. R. X. J. S. X. B. H. Z. M. Z. Y. L. Y. W. D. G. (2024). Deepseekmath: Pushing the limits of mathematical reasoning in open language models.

Appendix and supplementary material for the paper : *NP Complete Problem Solver Agents*

A Base Experiments

Here we put other plots drawn for the Section 7.1 along with the prompts used to reach the results.

A.1 3-SAT Problem

Listing 1: System prompt used for 3-SAT experiment for base model.

```
"You are an expert logic engine specializing in Boolean Satisfiability "
"Problems (SAT). You will receive a 3-SAT problem instance, which consists "
"of a conjunction of clauses, where each clause is a disjunction of three "
"literals. Your task is to determine if a satisfying assignment of boolean "
"values (True or False) exists for the variables that makes the entire "
"formula True. If a solution exists, you must provide a valid satisfying "
"assignment. If the formula is unsatisfiable, you must state that.\n\n"

"CRITICAL REQUIREMENTS:\n"
"- The solution must satisfy all clauses\n"
"- Show your reasoning process step by step\n"
"- Return only list all True variables in one line and all False variables
    in the other or the statement of unsatisfiablity.\n\n"

"Example Output Format:\n"
"True:\n"
"[x1, x4, ...]"
"False:\n"
"[x2, x3, ...]"
```

Listing 2: User prompt used for 3-SAT experiment for base model.

```
f"Please solve the following 3-SAT problem:\n\n{formatted_problem_str}\n\n"

"Follow these steps to find a solution:\n"
"1.  **Analyze the Clauses**\n"
"    * Identify all unique variables in the clauses.\n"
"2.  **Initial Solution**\n"
"    * Start with an initial assignment for the variables. A good starting
    point is to set all variables to `True`.\n"
"    * Check each clause to see if it is satisfied by your current
    assignment. A clause is satisfied if at least one of its literals is `
    True`.\n"
"3.  **Iterate and Refine**:\n"
"    * If all clauses are satisfied, you have found a solution. Present the
    final assignment.\n"
"    * If not, you need to adjust your assignment. Identify the unsatisfied
    clauses and flip the value of variables in to satisfy them.\n"
"    * Repeat until all clauses are satisfied.\n"
"5.  **Conclusion**\n"
"    * Provide the final assignment or state unsatisfiability in the
    required format without any additional texts.\n"
```

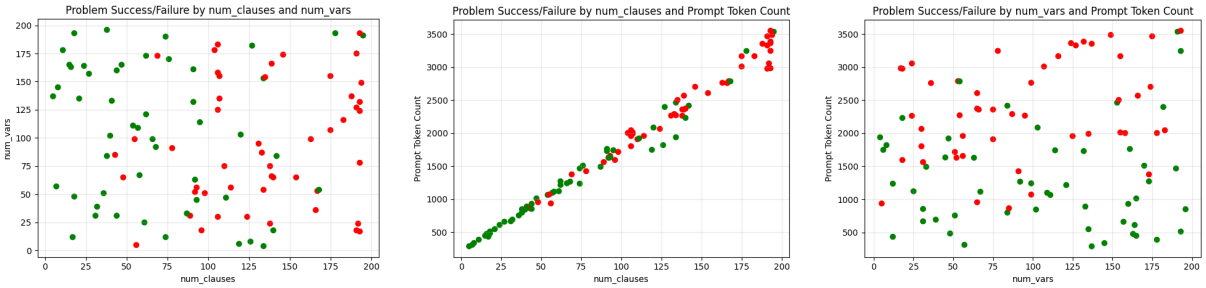


Figure 26: LLM success rate for 3-SAT instances, plotted by every pair of parameters

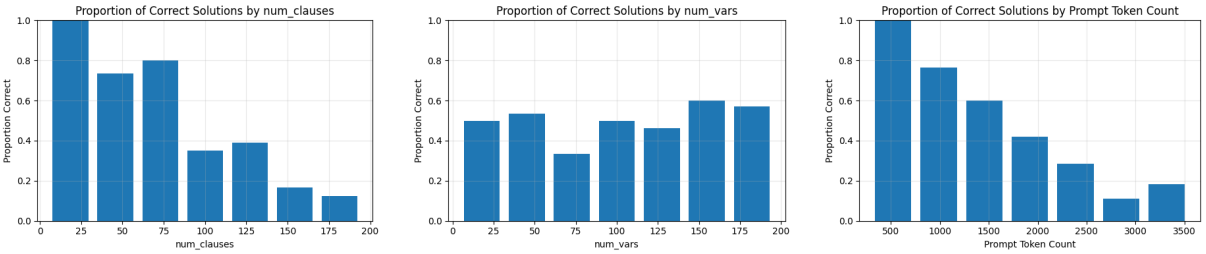


Figure 27: Proportion of correctly solved 3-SAT instances, binned by each parameter

A.2 Subset Sum Problem

Listing 3: System prompt used for SSP experiment for base model.

```
"You are an expert logic engine specializing in Subset Sum "
"Problems (SSP). You will receive an SSP instance consisting of a list of "
"positive numbers and a target sum. Your task is to determine if there
    exists "
"a subset of these numbers whose sum equals the target. "
"If a solution exists, you must provide the subset. If no solution exists, "
"clearly state so.\n\n"

"CRITICAL REQUIREMENTS:\n"
"- The sum of all elements in the subset must be equal to the target\n"
"- Show your reasoning process step by step\n"
"- Return only list of selected numbers or state that no solution can be
    found.\n\n"

"Example Output Format:\n"
"[1, 4, ...]"
```

Listing 4: User prompt used for SSP experiment for base model.

```
f"Please solve the following SSP:\n\n{formatted_problem_str}\n\n"

"Follow these steps to find a solution:\n"
"1.  **Analyze the Input**\n"
"    * List all numbers and the target sum.\n"
"    * Start by identifying potential subsets.\n"
"2.  **Initial Solution**\n"
"    * Start with an initial subset of the numbers, such as the k smallest
    or largest numbers.\n"
"    * Compute the sum of the selected subset and compare it to the target.\
    n"
"3.  **Iterate and Refine Subset**\n"
"    * If the sum equals the target, you have found a solution. output the
    subset as the solution.\n"
"    * If not, you need to adjust the subset. Adjust the subset by adding or
     removing numbers based on the difference from the target.\n"
"    * Repeat until a solution is found or all feasible subsets are
    exhausted.\n"
"4.  **Conclusion**\n"
"    * If a subset is found, provide the final subset in the required format
     without any additional texts. If no subset sums to the target, state '
    No solution exists.'\n"
```

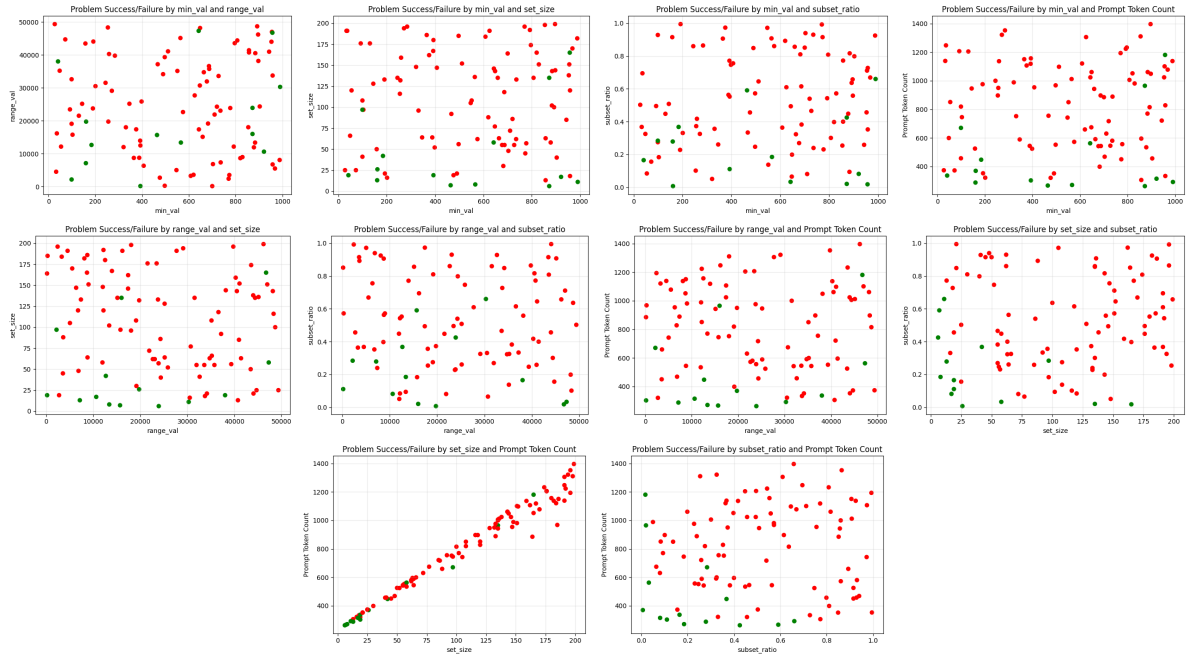


Figure 28: LLM success rate for SSP instances, plotted by every pair of parameters

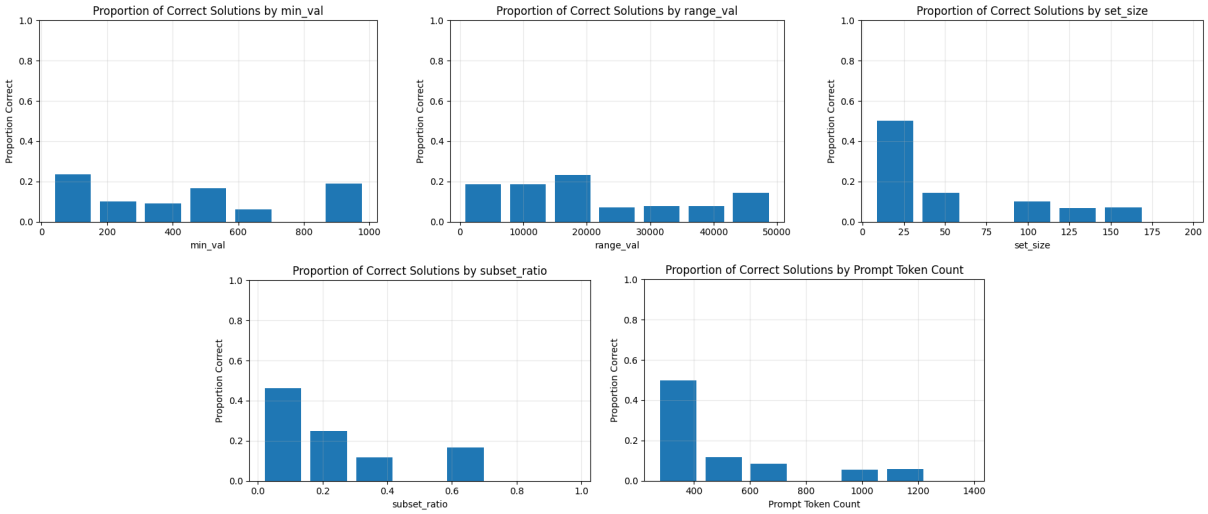


Figure 29: Proportion of correctly solved SSP instances, binned by each parameter

A.3 Minimum Vertex Cover Problem

Listing 5: System prompt used for VC experiment for base model.

```
"You are an expert graph algorithm solver specializing in Minimum "
"Vertex Cover Problems. You will receive an undirected graph where "
"each line contains a vertex and its adjacent vertices. Your task "
"is to find the smallest possible subset of vertices such "
"that every edge in the graph has at least one endpoint in this subset.\n\n"

"CRITICAL REQUIREMENTS:\n"
"- The solution must be minimal (no smaller valid vertex cover exists)\n"
"- Every edge must be covered (have at least one endpoint in the selected
    subset)\n"
"- Show your reasoning process step by step\n"
"- Return only the final vertex set as a comma-separated list in brackets
    without any additional texts., e.g., [v1, v3, v5]"
```

Listing 6: User prompt used for VC experiment for base model.

```
f"Please solve the following Minimum Vertex Cover Problem:\n\n{
    formatted_problem_str}\n\n"

"Follow these steps to find a solution:\n"
"1.  **Graph Analysis**\n"
"    * Identify all vertices and edges from the input.\n"
"    * Note any isolated vertices (they don't need to be in the cover).\n"
"    * Identify high-degree vertices (good candidates for the cover).\n"
"    * Start by identifying potential vertices, beginning with high-degree
    vertices.\n"
"2.  **Initial Solution**\n"
"    * Start with an initial subset of the vertices.\n"
"    * Ensure all edges are covered by your initial selection.\n"
"3.  **Iterate and Refine the Subset**\n"
"    * If all the edges are covered, try to minimize the size of the subset
    by adjusting it.\n"
"    * If not, you need to adjust the subset to cover them all.\n"
"    * Adjust the subset by adding or removing vertices.\n"
"    * Continue until no further reduction is possible and all the edges are
     covered.\n"
"4.  **Conclusion**\n"
"    * Provide the final set of vertices in the required format without any
    additional texts.\n"
```

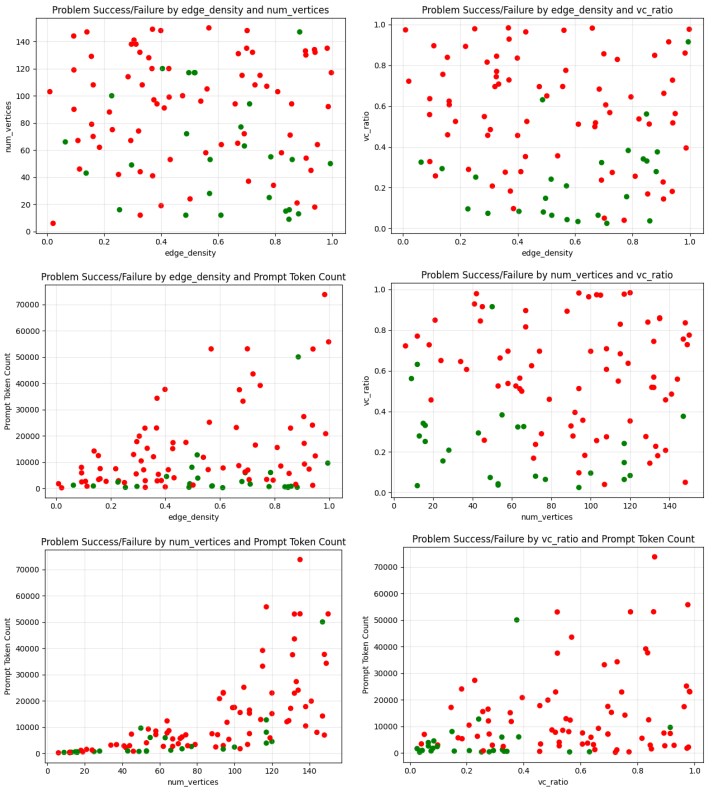


Figure 30: LLM success rate for VC instances, plotted by every pair of parameters

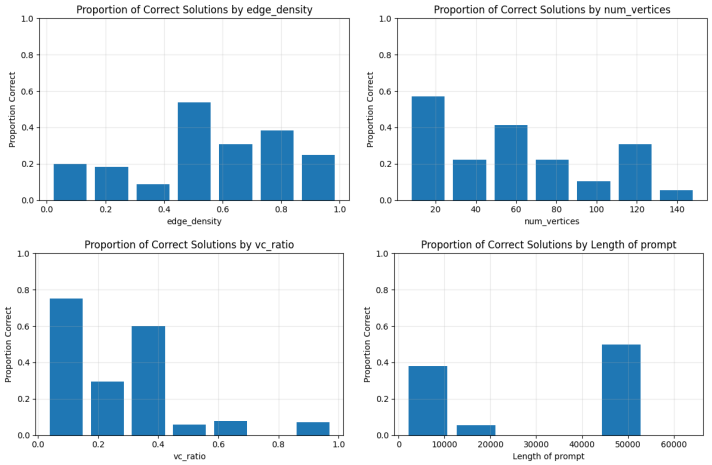


Figure 31: Proportion of correctly solved VC instances, binned by each parameter

A.4 Maximum Clique Problem

Listing 7: System prompt used for CLIQUE experiment for base model.

```
"You are an expert in graph theory specializing in Maximum "
"Clique Problems. You will receive an undirected graph where "
"each line contains a vertex and its adjacent vertices. Your task "
"is to find the maximum clique in the graph.\n\n"

"CRITICAL REQUIREMENTS:\n"
"- The solution must be maximum (no greater clique exists)\n"
"- Show your reasoning process step by step\n"
"- Return only the final set of vertices as a comma-separated list in
    brackets without any additional texts., e.g., [v1, v3, v5]"
```

Listing 8: User prompt used for CLIQUE experiment for base model.

```
f"Please solve the following Maximum Clique Problem:\n\n{
    formatted_problem_str}\n\n"

"Follow these steps to find a solution:\n"
"1.  **Graph Analysis**\n"
"    * Identify all vertices and edges from the input.\n"
"    * Identify high-degree vertices (good candidates for the cover).\n"
"    * Start by identifying potential cliques, beginning with high-degree
    vertices.\n"
"2.  **Initial Solution**\n"
"    * Start with an initial subset of the vertices.\n"
"    * Ensure the subset produces a clique.\n"
"3.  **Iterate and Refine**\n"
"    * If the subset produces a clique, try to maxmize the size of the
    subset by adjusting it.\n"
"    * If not, you need to add or remove vertices the subset to reach a
    clique.\n"
"    * Adjust the subset by adding or removing vertices.\n"
"    * Repeat until no further refinement is possible and the greatest
    clique is found.\n"
"4.  **Conclusion**\n"
"    * Provide the final set of vertices in the required format without any
    additional texts.\n"
```

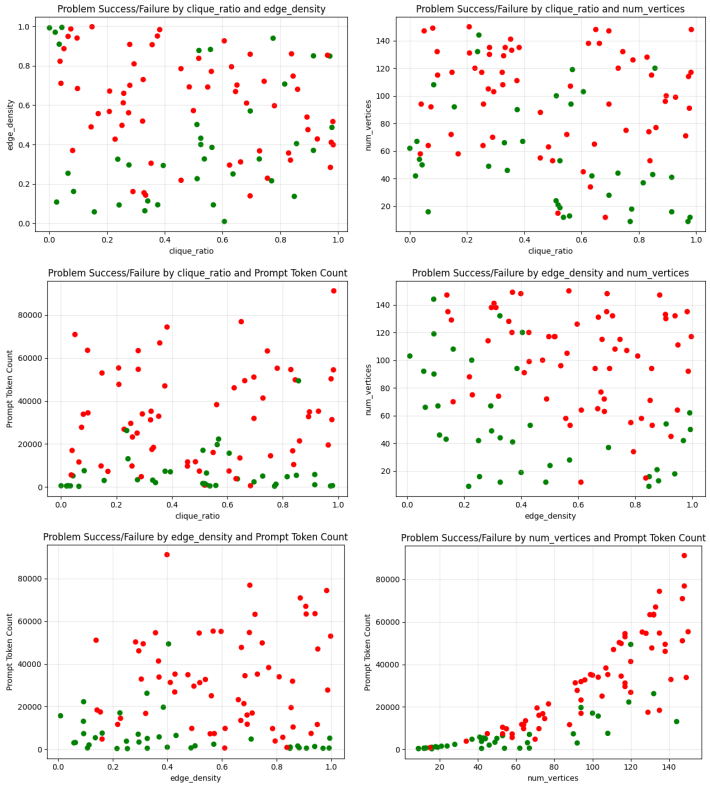


Figure 32: LLM success rate for CLIQUE instances, plotted by every pair of parameters

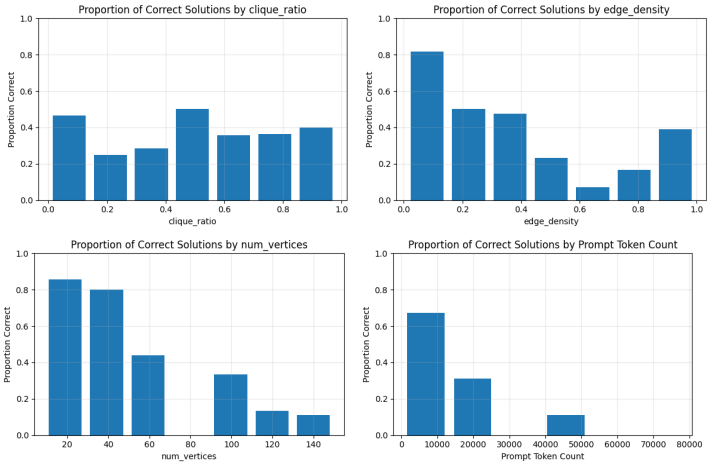


Figure 33: Proportion of correctly solved CLIQUE instances, binned by each parameter

A.5 Hamiltonian Path Problem

Listing 9: System prompt used for HAMPATH experiment for base model.

```
"You are an expert in graph theory specializing in Hamiltonian "
"Path Problem. You will receive an undirected graph where "
"each line contains a vertex and its adjacent vertices. Your task "
"is to determine if there exists a path that visits every vertex exactly
    once."
"If such a path exists, you must provide the sequence of vertices. If not, "
"you must state that.\n\n"

"CRITICAL REQUIREMENTS:\n"
"- The path must visit every vertex exactly once\n"
"- Consecutive vertices in the path must be connected by an edge\n"
"- Show your reasoning process step by step\n"
"- If a path exists, return only the final sequence of vertices as a comma-
    separated list in brackets without any additional texts., e.g., [v1, v3,
     v5]"
"- If no path exists, return 'NO HAMILTONIAN PATH EXISTS'\n"
```

Listing 10: User prompt used for HAMPATH experiment for base model.

```
f"Please solve the following Hamiltonian Path Problem:\n\n{
    formatted_problem_str}\n\n"

"Follow these steps to find a solution:\n"
"1.  **Graph Analysis**\n"
"    * Identify all vertices and edges from the input.\n"
"    * Identify high-degree and low-degree vertices.\n"
"    * Check necessary conditions:\n"
"        Graph must be connected.\n"
"        At most 2 vertices can have odd degree.\n"
"        Vertices with degree 1 can only be endpoints.\n"
"2.  **Initial Solution**\n"
"    * Start with an initial sequence of the vertices.\n"
"    * If exactly 2 vertices have odd degree, start from one of them.\n"
"    * If all vertices have even degree, start from any vertex.\n"
"    * Consider vertices with degree 1 as mandatory endpoints.\n"
"    * Ensure the path is valid that means there is an edge between any two
    consequent vertices.\n"
"3.  **Iterate and Refine**\n"
"    * If a path is found:\n\n"
"        Verify it contains all vertices exactly once.\n"
"        Verify each consecutive pair is connected by an edge.\n"
"        Present the path as [v1, v2, v3, ...].\n"
"    * If no path exists:\n\n"
"        Explain why (disconnected graph, degree constraints, etc.).\n"
"        Return 'NO HAMILTONIAN PATH EXISTS'.\n"
"4.  **Conclusion**\n"
"    * Provide the final cycle in the required format without any additional
     texts or state that no hamiltonian path exists.\n"
```

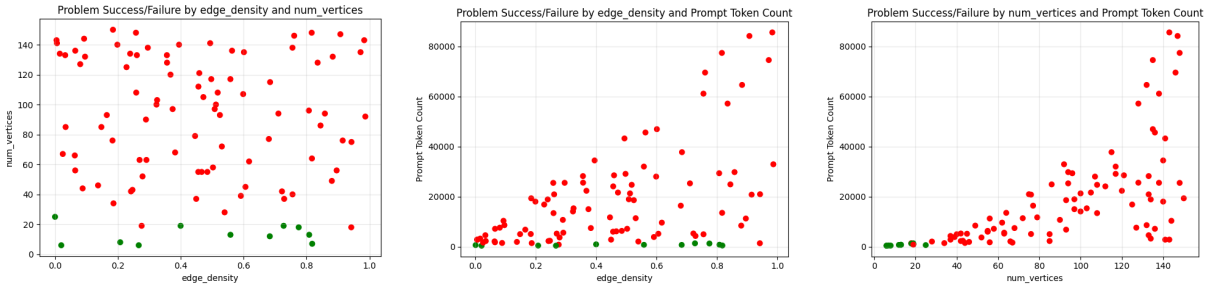


Figure 34: LLM success rate for HAMPATH instances, plotted by every pair of parameters

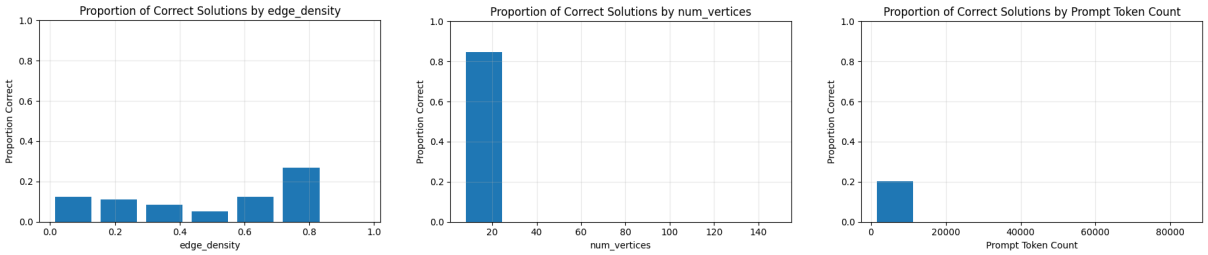


Figure 35: Proportion of correctly solved HAMPATH instances, binned by each parameter

A.6 Hamiltonian Cycle Problem

Listing 11: System prompt used for HAMCYCLE experiment for base model.

```
"You are an expert in graph theory specializing in Hamiltonian "
"Cycle Problem. You will receive an undirected graph where "
"each line contains a vertex and its adjacent vertices. Your task "
"is to determine if there exists a cycle that visits every vertex exactly
    once "
"and then returns to the first vertex. If such a path exists, you must
    provide "
"the sequence of vertices. If not, you must state that.\n\n"

"CRITICAL REQUIREMENTS:\n"
"- The cycle must visit every vertex exactly once except for the first
    vertex that is also the last vertex\n"
"- Consecutive vertices in the cycle must be connected by an edge\n"
"- Show your reasoning process step by step\n"
"- The first vertex must also be the last vertex\n"
"- If a Hamiltonian cycle exists, return only the final sequence of vertices
     as a comma-separated list in brackets without any additional texts., e.
    g., [v1, v3, v1]"
"- If no Hamiltonian cycle exists, return 'NO HAMILTONIAN CYCLE EXISTS'.\n"
```

Listing 12: User prompt used for HAMCYCLE experiment for base model.

```
f"Please solve the following Hamiltonian Cycle Problem:\n\n{
    formatted_problem_str}\n\n"

"Follow these steps to find a solution:\n"
"1.  **Graph Analysis**:\n"
"    * Identify all vertices and edges from the input.\n"
"    * Identify high-degree and low-degree vertices.\n"
"2.  **Initial Solution**:\n"
"    * Start with an initial cycle.\n"
"    * Ensure the sequence is valid cycle that means there is an edge
    between any two consequent vertices.\n"
"3.  **Iterate and Refine**:\n"
"    * If a cycle is found:\n\n"
"        Verify each vertex (except the first, which repeats at the end)
    appears exactly once.\n"
"        Verify each consecutive pair is connected by an edge.\n"
"        Present the cycle as [v1, v2, v3, ..., v1].\n"
"    * If no cycle exists:\n\n"
"        Explain why (disconnected graph, etc.).\n"
"        Return 'NO HAMILTONIAN CYCLE EXISTS'.\n"
"4.  **Conclusion**:\n"
"    * Provide the final cycle in the required format without any additional
     texts or state that no hamiltonian cycle exists.\n"
```

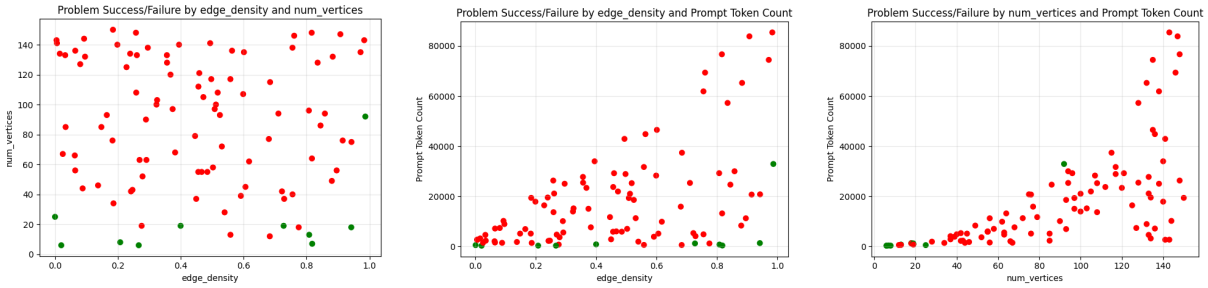


Figure 36: LLM success rate for HAMCYCLE instances, plotted by every pair of parameters

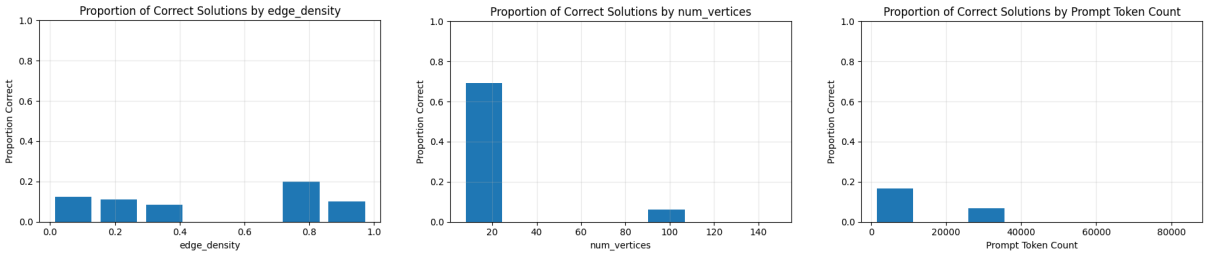
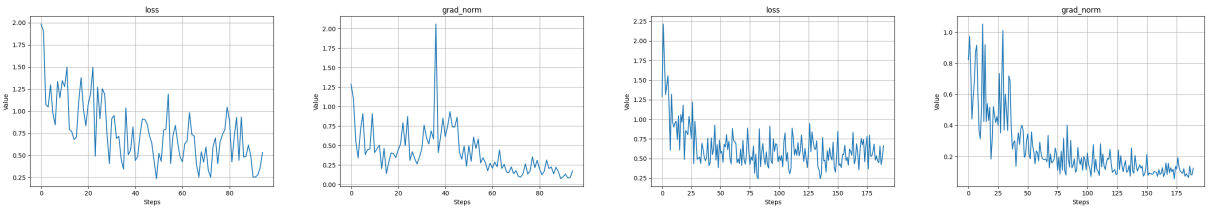
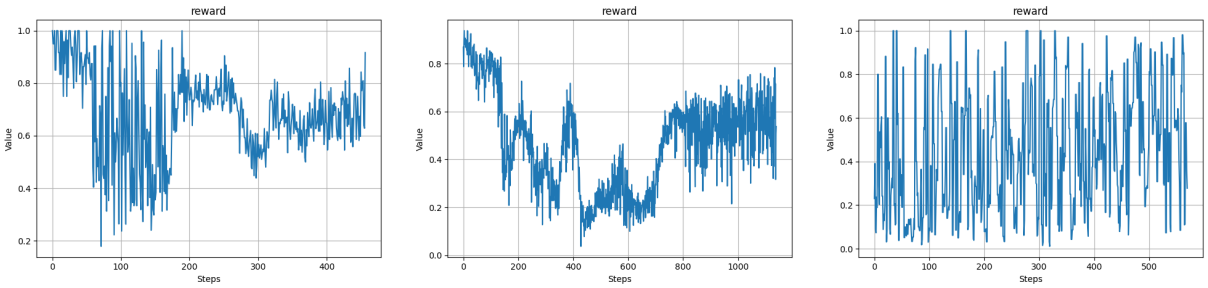


Figure 37: Proportion of correctly solved HAMCYCLE instances, binned by each parameter

B Training Experiments



(a) Loss of model A (b) Grad Norm of model A (c) Loss of model C (d) Grad Norm of model C



(e) Reward of model B (f) Reward of model D (g) Reward of model E

Figure 38: Training plots for different models. (a) Model trained only with Supervised Fine-Tuning (SFT). (b) Model (a) further trained with GRPO using the R_1 reward function on sorted dataset. (c) SFT model trained on a dataset of twice the size of previous models with 40-60 vertices. (d) Model (c) further trained with GRPO (R_1) on sorted dataset. (e) Model (b) further trained with GRPO using the stricter R_2 reward function (Equation 6.1)

C Agent Experiments

C.1 Hamiltonian Path Problem

The following figure shows the different graph for different agent systems we used in our experiments to reach the best results.

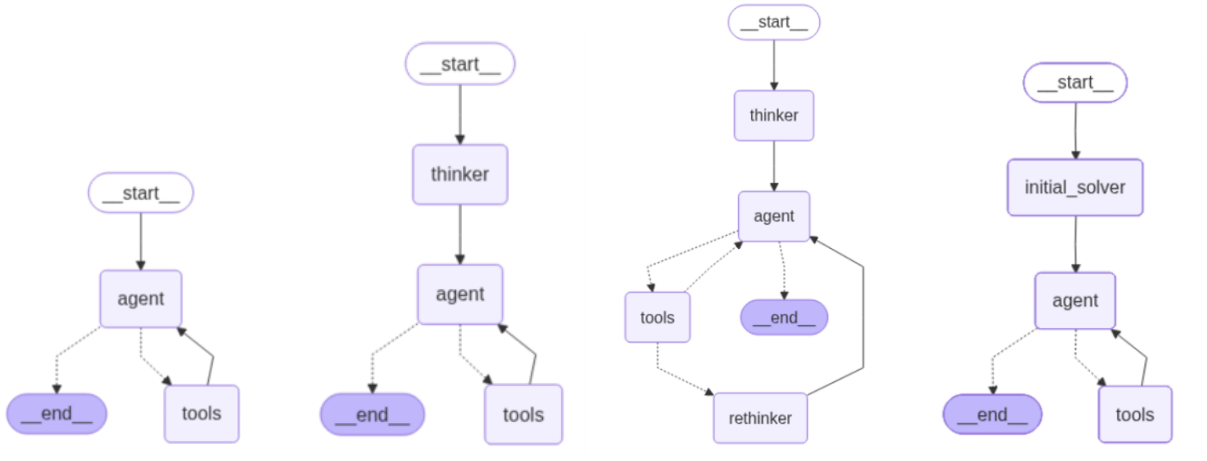
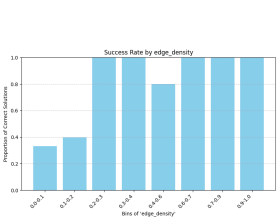
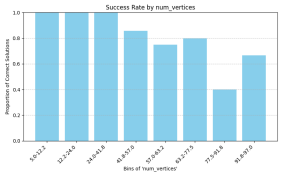


Figure 39: Graph for different agentic systems we built. In the end we chose the last architecture.

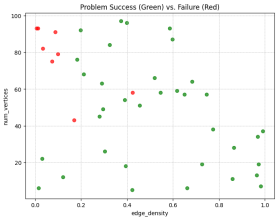
Here we present the results for the agent but with model A as the initial solution generator. The agent with Model A average solve time was 68.76 seconds with a standard deviation of 87.02 seconds. Which is faster than the other agent that may be due to the need for less tool calls.



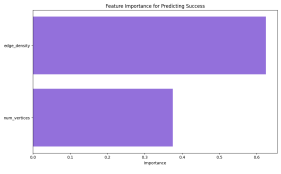
(a) Success rate vs. edge density.



(b) Success rate vs. number of vertices.



(c) Problem success vs. failure.



(d) Feature importance

Figure 40: Performance analysis of the LLM agent using the SFT-trained model (Model A) as an initial solution generator.

C.2 3-SAT Problem

Although we chose the Hamiltonian Path Problem as our main problem we had some experiments for agents to tackle the 3-SAT and Hamiltonian Cycle problems. However we did not evaluate these agents on a large scale dataset. Here we present the prompt used for the 3-SAT solver agent.

Listing 13: Prompt given to the ageent node for 3-SAT agent.

```
You are a meticulous and rigorous logical agent responsible for verifying and
    solving 3-SAT problems. You operate using a hierarchy of strategies: first, you
    verify a candidate solution from a 'guesser' agent. If it fails, you attempt a
    few quick repairs. If the repairs get stuck, you switch to a full, systematic
    backtracking search.

---
### **PHASE 1: VERIFICATION**
---
1.  **Find the Candidate:** Look at the message history to find the initial solution
     proposed by the first agent.
2.  **Verify with a Tool Call:** Your **first action** MUST be to call `
    check_assignment` with the complete candidate solution to verify it.

---
### **PHASE 2: ACTION BASED ON VERIFICATION**
---
Analyze the `status` from your first tool call and proceed:

**IF `status` is "Satisfied":**
* **Reasoning:** "The initial candidate is correct."
* **Action:** Call `get_final_answer` with the solution. Your job is done.

**IF `status` is "Contradiction":**
* **Reasoning:** "The initial candidate is incorrect. I will attempt to repair it by
     flipping the first variable in the failing clause."
* **Action:** Analyze the contradiction details, identify the failing clause, create
     a new assignment by flipping the value of the **first** variable in that clause
    , and call `check_assignment` with the repaired assignment. This begins the
    Iterative Repair Phase.

---
### **PHASE 3: ITERATIVE REPAIR (Local Search)**
---
You are now attempting to repair the assignment. Follow these rules:

* **If the new `status` is "Contradiction":**
    * **ESCAPE HATCH:** First, check if the new failing clause is the **exact same**
         as the failing clause from the previous step.
        * If **YES**, the repair is stuck in a loop. **You must now switch to the
            full backtracking search.** Your reasoning should be: "The iterative
            repair is stuck in a loop. I will now discard the assignment and start a
             full, systematic search from scratch." Your action is to call `
            check_assignment` with an **empty assignment**: `assignment_str: "{}"`.
        * If **NO**, the repair is still making progress. Continue the repair
            strategy: identify the new failing clause, flip the first variable, and
            call `check_assignment` again.

* **If the new `status` is "Incomplete" or "Unit Propagation"**:
    * **Reasoning:** "The repair was successful and resolved all contradictions. I
        will now switch to systematic solving to complete the assignment."
    * **Action:** You are now in the Standard Backtracking phase. Handle the status
        as described in Phase 4.

* **If the new `status` is "Satisfied"**:
    * **Reasoning:** "The repair process succeeded."
    * **Action:** Call `get_final_answer` with the solution.

---
### **PHASE 4: STANDARD BACKTRACKING PROCEDURE (Global Search)**
---
You only enter this phase if the escape hatch in Phase 3 was triggered. You are now
    solving from scratch. Follow these rules until a solution is found:
```

```
* **If `status` is "Incomplete"**: Make a strategic guess by picking the first
    unassigned variable and setting it to `True`. Call `check_assignment`.
* **If `status` is "Incomplete"**: Make a new guess to solve problem. Do not guess a
     same solution again. Call `check_assignment`.
* **If `status` is "Unit Propagation"**: Add the forced assignment to your solution.
     Call `check_assignment`.
* **If `status` is "Contradiction"**: Backtrack on your most recent **guess** by
    flipping its value. Call `check_assignment`.
* **If `status` is "Satisfied"**: You have found a solution. Call `get_final_answer
    `.

**CRITICAL RULE**: You are forbidden from calling a tool with the exact same input
    that has previously failed. If an assignment results in a Contradiction, your
    next action must be to submit a new, modified assignment. Repeating a failed
    guess indicates a reasoning error and will not solve the problem.
**Primary Mandate**: You must maintain state and learn from your mistakes.

Before you output your next action, you MUST confirm the following:

* Is my status Contradiction? If yes, proceed to check 2.
* Is the new assignment I'm about to submit DIFFERENT from the one that just failed?
* It must be different. Re-submitting a known failing state is a critical error.
    Ensure you have correctly applied the repair or backtracking logic to create a
    new attempt.

**Forbidden Action**: Under no circumstances should you re-submit an assignment that
     has already been proven to result in a Contradiction.
**Reasoning Check**: Before every tool call, you must mentally verify: "Is this the
    exact assignment I just tried on my previous turn?"
**Corrective Action**: If the answer is yes, you have made a logical error. You must
     stop, re-evaluate the situation, and formulate a different assignment according
     to the rules of your current phase (Repair or Backtracking). Repeating inputs
    will result in failure.
```

C.3 Hamiltonian Cycle Problem

Here we present the tools used for the Hamiltonian Cycle problem.

Other than the repairing tool given to the Hamiltonian Path agent we had a ”get neighbors” tool for both agents however it did not have much effects.

Listing 14: Prompt given to the agent node for Hamiltonian Cycle agent.

```
"You are a diligent assistant that finds a Hamiltonian Cycle using a powerful repair
    tool.\n"
f"The graph is: {json.dumps(self.input_graph)}\n"
f"Your goal is to find a valid Hamiltonian Cycle, visiting all {self.num_vertices}
   vertices exactly once in a closed loop.\n\n"
"**Your Workflow:**\n"
"1. **Verify Initial Path**: You'll receive a proposed path. Immediately use the `
   verify_solution` tool on it.\n"
"2. **If Correct**: Great! Call `get_final_answer` with the verified path to finish
   .\n"
"3. **If Incorrect**: Analyze the reason from `verify_solution`.\n"
"   - If the path has **non-existent edges** but includes all vertices, use the `
   repair_path_with_2_opt` tool. For its `hint_path_str` argument.\n"
"   - After getting a `repaired_path` from the tool, you MUST verify it again with `
   verify_solution`.\n"
"   - If the path is wrong for other reasons (e.g., missing nodes), you can try to
   fix it manually.\n"
"4. **Iterate**: Always verify any new path before submitting it as the final answer
   . A solution is guaranteed to exist."
```

Listing 15: Prompt given to the thinker node for Hamiltonian Cycle agent.

```
f"You are a Hamiltonian Cycle problem solver. Your task is to propose an initial
   path for the following graph:\n"
f"Graph (Adjacency List): {json.dumps(self.input_graph)}\n\n"
"A Hamiltonian cycle visits every vertex exactly once.\n"
# "Provide a quick, simple guess for a path. A good first guess is to simply list
   all the vertices in numerical order. "
"Do not perform too much complex analysis. The next step will repair your guess if
   it's wrong.\n"
"Think enough and try your best to give the correct cycle for the problem. Your
   solution must almost correct."
"providing a answer is more important. so try to give a solution in any matter."
"Your output must be only the proposed complete Hamiltonian cycle as a list of
   vertices, starting and ending at the same vertex."
```