LLMs as Translators, Not Thinkers: Structured Output Enables Stronger NP-Hard Problem Solving

Haoyu Wang
University of Pennsylvania, USA

Dan Roth
University of Pennsylvania, USA

Abstract

We study how large language models (LLMs) tackle NP-hard problems described in natural language, comparing direct reasoning, search-based refinement, and translation to structured solver inputs. Using models like GPT-4o and OpenAI o1, we analyze trade-offs between accuracy and efficiency. Our results highlight both the potential and limitations of LLMs in constraint-based tasks, offering insights for hybrid LLM-solver systems[1].

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases LLM, NP-hard, Combinatorial Optimization

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

LLMs are increasingly being applied to combinatorial problems, offering natural language interfaces and flexible reasoning capabilities. Yet, their behavior on classically hard problems—such as Hamiltonian Cycle and Traveling Salesman—remains poorly understood. In this work, we investigate how LLMs solve NP-hard problems across three paradigms: (i) direct reasoning via chain-of-thought, (ii) iterative search guided by local feedback, and (iii) translation-based solving where LLMs convert textual descriptions of problems into structured solver input. We benchmark general-purpose models (e.g., GPT-4o [3]) and reasoning-optimized models (e.g., OpenAI o1 [4]), using natural language inputs with varying phrasing and complexity. Our results highlight how solution quality depends on problem scale, amount of distractors in the problem description, and the strategy of using LLMs. We find that the most reliable way to solve NP-hard problems with LLMs is not by having them reason directly or using them as heuristics to guide tree search, but by prompting them to translate problem descriptions into structured formats compatible with classical solvers.

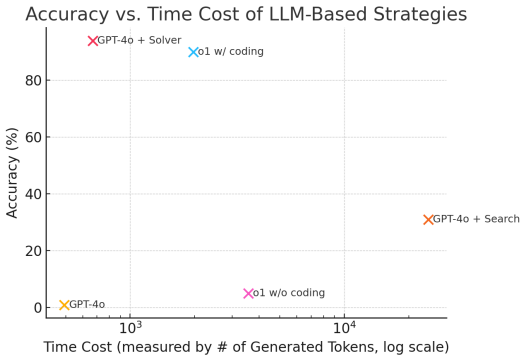

Figure 1 Accuracy vs. time cost (measured by # of generated tokens) for different LLM strategies. Translation-based solving yields the best trade-off, while iterative tree search improves over direct reasoning but is more expensive.

[1] We plan to release a full version of this paper with complete results and extended discussion.

2 Methods

We compare three approaches for solving combinatorial problems with LLMs: direct reasoning, translation with solver, and LLM-guided Monte Carlo tree search (MCTS) [1]. These methods vary in how they leverage the LLM and how they handle problem constraints and search.

Direct reasoning via chain-of-thought. In this approach, we prompt the LLM to directly generate a complete solution based on the natural language description of the problem, using a chain-of-thought (CoT) prompt[2] to encourage step-by-step reasoning.

LLM as Translator and Solver Pipeline. This approach employs the LLM to convert a natural language problem description into a structured format (e.g., a graph definition, item list with weights and values) that can be directly consumed by a conventional solver. Once the structured input is produced, a deterministic algorithm—such as DFS for Hamiltonian Cycle or dynamic programming for 0-1 Knapsack—is applied to obtain a solution.

LLM-Guided MCTS Framework. We develop a framework that integrates MCTS with LLMs to optimize solutions for combinatorial tasks. The algorithm begins with a feasible initial solution obtained from the direct reasoning method, represented as the root of the search tree. At each expansion step, the LLM is prompted to generate a set of candidate modifications to the current solution. These candidates are assessed by a task-specific evaluator[3] which computes the objective value and/or constraint satisfaction, according to different NP-hard problems. Each valid candidate is added to the tree as a child node, and MCTS selects promising branches using either UCT or PUCT scoring. During rollouts, we simulate random paths through the tree to estimate the best achievable reward from a given node. The maximal reward encountered is then backpropagated to update node statistics. In this setup, the LLM provides a learned prior over the action space, while MCTS systematically explores and refines high-quality solutions.

3 Experiments and Results

We evaluate the aforementioned methods on four NP-hard problems: Hamiltonian Cycle, Vertex Cover, 0-1 Knapsack, and TSP with time window, with instances varying in size and the presence of distractors—irrelevant or misleading information embedded in the textual description. For graph-based problems, we compare three translation output formats[4]: (1) edge list, (2) lexicographically ordered adjacency list, and (3) adjacency list following the order of appearance in the input. Evaluation metrics include accuracy or approximation score relative to optimal, and efficiency (measured by the number of generated tokens).

As shown in Fig 1, translation-based method with the third format yields the best performance. Iterative methods improve over direct LLM guesses but remain limited by token budgets and rollout depth. Reasoning-optimized models occasionally succeed on small, well-structured instances but degrade with problem size or added noise. We find that using LLMs as translators into structured formats, followed by solvers, is the most reliable and efficient strategy for tackling NP-hard problems described in natural language[5]. Yet the effectiveness of the approach is highly sensitive to the format choice.

---

[2] Details see Appendix A.
[3] We translate the problem description into a structured format as in the second method. The evaluator operates solely on this structured representation to assess objective value and constraint satisfaction.
[4] Details see Appendix B.
[5] We find that finetuning does not show significant improvement, which is consistent with findings in [2].

References

1 Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

2 Lizhou Fan, Wenyue Hua, Lingyao Li, Haoyang Ling, and Yongfeng Zhang. NPHardEval: Dynamic benchmark on reasoning ability of large language models via complexity classes. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4092–4114, Bangkok, Thailand, August 2024. Association for Computational Linguistics. URL: https://aclanthology.org/2024.acl-long.225/, doi:10.18653/v1/2024.acl-long.225.

3 Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

4 Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

A Example Prompts for Direct Reasoning

Example Prompt: Hamiltonian Cycle in a Real-World Graph

System: You are a helpful assistant that solves graph problems. Think step-by-step and use <think> tag to enclose your thinking thoughts. Give me an answer with your own reasoning. Refrain from writing code to solve it. Always verify the constraints and provide a valid solution.
User: In a metropolitan area, there exist 30 delivery hubs identified as A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, AA, AB, AC, AD.
Hub AA processes around 90 packages daily.
Station F has direct connections to D, E, G.
Hub H is linked directly to A, B, C.
...
E connects directly to D, O, P, AB, and AC.
Create a route that commences from one station, visits each station precisely once, and ultimately returns to the original station.

B Prompts for Translation-based Methods

Example Prompt: Translation to Edge List Format

System: You are an expert at converting real-world problem descriptions into structured formats for Python solvers. Your task is to extract graph edges from the description below. Output a single line of JSON with exactly the following keys: `"sha"`, `"labels"`, and `"edges"`.

- `"sha"` is a made-up 12-character string (e.g., "abc123ef4567") representing the graph ID.
- `"labels"` is the full list of nodes in lexicographic order.
- `"edges"` is a list of two-element lists, each representing an undirected edge as a pair of node labels in lexicographic order.

Output nothing but the JSON.
User: In a metropolitan area, there exist 30 delivery hubs identified as A, B, C, ..., AD.
Hub F is connected to D, E, G.
Hub H is linked to A, B, C.
...
E connects directly to D, O, P, AB, and AC.
Create a route that starts and ends at the same station, visiting each exactly once.

Example Prompt: Translation to Lexicographic Adjacency List

System: You are an expert at converting real-world problem descriptions into structured formats for Python solvers. Your task is to extract lexicographically ordered adjacency list from the description below. Output a single JSON object with: `"sha"`, `"labels"`, and `"adjacency"`.

- `"sha"` is a made-up 12-character string.
- `"labels"` lists all nodes in lexicographic order.
- `"adjacency"` maps each node (as key) to a list of neighbors, also in lexicographic order.

Ensure that all edges are bidirectional, and no duplicates exist.
Output nothing else but the JSON.
User: In a metropolitan area, there exist 30 delivery hubs identified as A, B, C, ..., AD.
Hub F is connected to D, E, G.
Hub H is linked to A, B, C.
...
E connects directly to D, O, P, AB, and AC.
Create a route that starts and ends at the same station, visiting each exactly once.

Example Prompt: Translation to Lexicographic Adjacency List

System: You are an expert at converting real-world problem descriptions into structured formats for Python solvers. Parse the delivery station connections and return an adjacency list that respects the order in which nodes appear in the description. Output one line of JSON with the following fields: `"sha"`, `"labels"`, `"adjacency"`.

- `"labels"` should list station names in the order they first appear in the description.
- `"adjacency"` is a dictionary that maps each station to its neighbors, preserving the order of mention.

Each connection is undirected. Output strictly one line of JSON, no explanations.
User: In a metropolitan area, there exist 30 delivery hubs identified as A, B, C, ..., AD.
Hub F is connected to D, E, G.
Hub H is linked to A, B, C.
...
E connects directly to D, O, P, AB, and AC.
Create a route that starts and ends at the same station, visiting each exactly once.