Navigating the Labyrinth: Evaluating LLMs' Ability to Reason About Search Problems

Anonymous authors
Paper under double-blind review

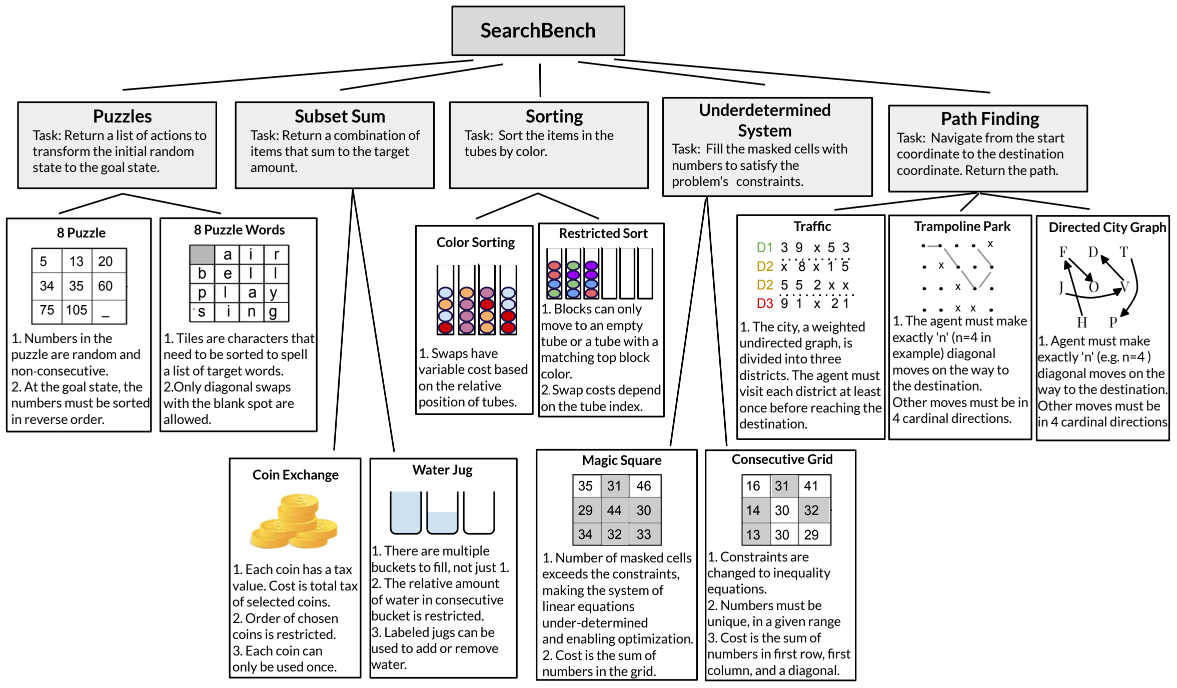
Abstract

Large Language Models (LLMs) have recently achieved impressive performance in math and reasoning benchmarks. However, they often struggle with logic problems and puzzles that are relatively easy for humans. To further investigate this, we introduce a new benchmark, SearchBench, which contains 11 unique search problems inspired by intuitive puzzles. Each SearchBench problem type is equipped with automated pipelines to generate an arbitrary number of instances and analyze the feasibility, correctness, and optimality of LLM-generated solutions. We show that using step-by-step, language-only reasoning, even the most advanced LLMs fail to solve SearchBench; for example, OpenAI's frontier models GPT-4 and o1-preview solve only 1.4% and 18.6% of problems, respectively. The reason is that SearchBench problems require considering multiple pathways and performing backtracking, posing a significant challenge to auto-regressive models. Interestingly, performance is significantly boosted when we prompt models to generate a complete A* search algorithm—a comparatively more cognitively difficult task. This approach effectively offloads the iterative search and backtracking process from the models, which they struggles with in text. This in-context learning baseline is further enhanced via a Multi-Stage-Multi-Try (MSMT) inference method, increasing GPT-4's rate of correct solutions to over 57%.

1 Introduction

The advent of Large Language Models (LLMs) has revolutionized the field of natural language processing, with models such as Llama3.1 (Meta, 2024), GPT-4 (OpenAI, 2023), and o1-preview (OpenAI, 2024) demonstrating unprecedented performance on math and science QA benchmarks, such as GSM8k (Cobbe et al., 2021) and GPQA (Rein et al., 2023). However, LLMs still exhibit surprising failures on some intuitive tasks (Bian et al., 2023; Qin et al., 2023; Marcus, 2020) and struggle with multi-step compositional reasoning, combinatorial problems, and planning (Dziri et al., 2024; Valmeekam et al., 2022; Wu et al., 2023). Inspired by these observations and to further investigate LLMs' reasoning abilities, we offer a new benchmark of combinatorial search problems, SearchBench. The problems of SearchBench are inspired by popular puzzles, are predominantly NP-hard combinatorial problems, and necessitate exploring multiple action paths and backtracking to previous states.

SearchBench is challenging for LLMs due to several factors. The autoregressive architecture of current LLMs forces solving problems sequentially, which makes tasks requiring backtracking challenging (Dziri et al., 2024). Additionally, natural language is inherently linear and inefficient for representing complex relational structures (Dziri et al., 2024; Valmeekam et al., 2023; Borazjanizadeh & Piantadosi, 2024). Moreover, in combinatorial problems, the number of possible states increases exponentially with the number of actions, making naive exploration of the state space ineffective. Our empirical results show that even the most capable models solve less than 20% of SearchBench problems end-to-end. To successfully solve SearchBench, a model must backtrack, consider multiple reasoning paths, and identify the most optimal outcome among many feasible options. These capabilities are essential for robust reasoning, making SearchBench a valuable benchmark for evaluating LLMs' reasoning capabilities as they continue to evolve.



Puzzles — Task: Return a list of actions to transform the initial random state to the goal state.

Subset Sum — Task: Return a combination of items that sum to the target amount.

Sorting — Task: Sort the items in the tubes by color.

Underdetermined System — Task: Fill the masked cells with numbers to satisfy the problem's constraints.

Path Finding — Task: Navigate from the start coordinate to the destination coordinate. Return the path.

8 Puzzle — 1. Numbers in the puzzle are random and non-consecutive. 2. At the goal state, the numbers must be sorted in reverse order.

8 Puzzle Words — 1. Tiles are characters that need to be sorted to spell a list of target words. 2.Only diagonal swaps with the blank spot are allowed.

Color Sorting — 1. Swaps have variable cost based on the relative position of tubes.

Restricted Sort — 1. Blocks can only move to an empty tube or a tube with a matching top block color. 2. Swap costs depend on the tube index.

Traffic — 1. The city, a weighted undirected graph, is divided into three districts. The agent must visit each district at least once before reaching the destination.

Trampoline Park — 1. The agent must make exactly 'n' (n=4 in example) diagonal moves on the way to the destination. Other moves must be in 4 cardinal directions.

Directed City Graph — 1. Agent must make exactly 'n' (e.g. n=4) diagonal moves on the way to the destination. Other moves must be in 4 cardinal directions

Coin Exchange — 1. Each coin has a tax value. Cost is total tax of selected coins. 2. Order of chosen coins is restricted. 3. Each coin can only be used once.

Water Jug — 1. There are multiple buckets to fill, not just 1. 2. The relative amount of water in consecutive bucket is restricted. 3. Labeled jugs can be used to add or remove water.

Magic Square — 1. Number of masked cells exceeds the constraints, making the system of linear equations under-determined and enabling optimization. 2. Cost is the sum of numbers in the grid.

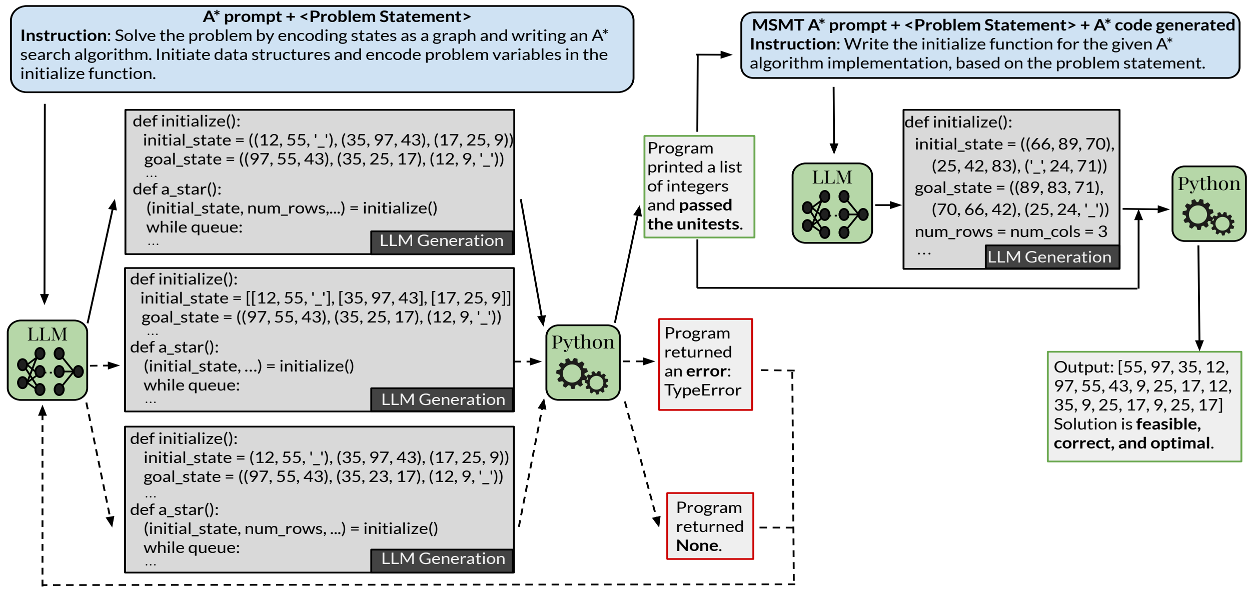
Consecutive Grid — 1. Constraints are changed to inequality equations. 2. Numbers must be unique, in a given range 3. Cost is the sum of numbers in first row, first column, and a diagonal.

Figure 1: The taxonomy of SearchBench. The five nodes in level one represent the problem categories, and the 11 nodes in level two represent the problem types. We detail how the rules of known puzzles and combinatorial problems are modified in SearchBench to ensure LLMs haven't encountered solved instances of the problems during training.

SearchBench has five problem categories: (i) pathfinding, (ii) puzzles, (iii) subset sum, (iv) sorting, and (v) under-determined systems; further divided into 11 unique problem types. Each problem type is inspired by known puzzles and combinatorial problems, but includes modified rules to ensure they differ substantially from solved instances of the original problems that appear on the internet and are likely observed by LLMs during their pre-training. We generate about 100 instances of varying difficulty per problem type using an automatic pipeline, totaling 1,107 fixed instances. Each problem type in SearchBench also includes an automatic evaluation pipeline that assesses LLM-generated solutions on three dimensions: feasibility (choosing sequence of actions that adhere to the problem's rules), correctness (achieving the goal state), and optimality (finding the least cost solution).

Our analysis of LLMs' performance on our benchmark reveals a paradox. While models fail at the cognitively simple task of solving these puzzles step-by-step, we find their performance is significantly boosted when prompted to generate a complete A search algorithm. This approach succeeds because it leverages the model's strength in code generation, which is not an iterative task, offloading the exploration of a large action space from LLMs to code execution, resulting in improved performance (Fig. 3). A* algorithm itself is a heuristic-based search algorithm known for its time efficiency and provable optimality, offering advantages over BFS, which is computationally inefficient (Wikipedia, 2025a), and DFS, which does not guarantee an optimal solution (Wikipedia, 2025b).

Additionally, to improve the quality of the generated A* codes, motivated by recent work showing that multiple inferences and task decomposition improve LLM performance (Wang et al., 2022; Yao et al., 2023a; Long, 2023), we introduce the Multi-Stage-Multi-Try (MSMT) inference strategy. In this approach, we divide code generation into two stages. First, we prompt the model to write an instance-agnostic A* algorithm for the problem type. We then verify this implementation against a set of unit tests, without evaluating the solution, to check if (i) the code is executable, (ii) returns a list as output, and (iii) the data type of list elements is correct. Second, we instruct the model to implement the 'initialize' function, which encodes the variables specific to each problem instance. Our MSMT A* method (Fig. 2) significantly improves LLMs'



A* prompt + <Problem Statement>
Instruction: Solve the problem by encoding states as a graph and writing an A* search algorithm. Initiate data structures and encode problem variables in the initialize function.

```
def initialize():
   initial_state = ((12, 55, '_'), (35, 97, 43), (17, 25, 9))
   goal_state = ((97, 55, 43), (35, 25, 17), (12, 9, '_'))
   ...
def a_star():
   (initial_state, num_rows,...) = initialize()
   while queue:
   ...
```

```
def initialize():
  initial_state = [[12, 55, '_'], [35, 97, 43], [17, 25, 9]]
  goal_state = ((97, 55, 43), (35, 25, 17), (12, 9, '_'))
   ...
def a_star():
   (initial_state, ...) = initialize()
   while queue:
   ...
```

```
def initialize():
   initial_state = (12, 55, '_'), (35, 97, 43), (17, 25, 9))
   goal_state = ((97, 55, 43), (35, 23, 17), (12, 9, '_'))
   ...
def a_star():
   (initial_state, num_rows, ...) = initialize()
   while queue:
   ...
```

Program printed a list of integers and **passed the unitests**.

Program returned an **error**: TypeError

Program returned **None**.

MSMT A* prompt + <Problem Statement> + A* code generated
Instruction: Write the initialize function for the given A* algorithm implementation, based on the problem statement.

```
def initialize():
   initial_state = ((66, 89, 70),
      (25, 42, 83), ('_', 24, 71))
   goal_state = ((89, 83, 71),
      (70, 66, 42), (25, 24, '_'))
   num_rows = num_cols = 3
   ...
```

Output: [55, 97, 35, 12, 97, 55, 43, 9, 25, 17, 12, 35, 9, 25, 17, 9, 25, 17]
Solution is **feasible, correct, and optimal**.

Figure 2: Our Multi-Stage-Multi-Try (MSMT) A* prompting approach.

ability to solve search problems, outperforming other prompting strategies we used, including 0-shot text, 4-shot Chain-of-Thought (CoT) (Wei et al., 2022) text, 0-shot code, and 4-shot A* prompting with naive greedy decoding. This method constitutes our strongest baseline on SearchBench; GPT-4 prompted with MSMT A* surpassing the o1-preview model. However, even using this approach, there remains considerable room for improvement on SearchBench, underscoring the challenge it presents in advancing LLMs' reasoning capabilities.

To summarize, our main contributions are as follows: (i) We introduce the SearchBench benchmark, designed to evaluate LLMs' ability to solve combinatorial problems that require search and backtracking. (ii) We demonstrate a key LLM bottleneck by showing that while models fail at iterative reasoning, they can successfully generate complex search algorithms; we then harness this capability with the MSMT A* framework to boost LLMs' performance on SearchBench. (iii) We use SearchBench to thoroughly analyze the reasoning capabilities of several frontier LLMs, including GPT-4 and o1-preview, by employing various prompting and inference methods. This analysis uncovers the limitations of LLMs in tasks requiring iterative reasoning and highlights the potential for improving LLM performance on SearchBench.

2 SearchBench Benchmark

SearchBench includes five categories of problems: puzzles, subset sum, sorting, pathfinding, and underdetermined systems. In theoretical computer science, combinatorial problems are classified into four types: existence, construction, enumeration, and optimization problems (Wilson, 2016). To ensure broad representation, we selected one problem category from each of these types for SearchBench. Particularly, subset sum problems represent the existence category, where the task is to determine if a subset of a given set sums to a specified value (refer to Tab. 1 for an example problem in this category). The 8-puzzle and 8-puzzle words fall under construction problems, which involve solving puzzles. Sorting problems, such as color sort and restricted sorting, are enumeration problems. Pathfinding problems are categorized as optimization problems.

Additionally, we introduce a new category of NP-hard combinatorial problmes in SearchBench, underdetermined system problems. These problems consist of constraint satisfaction problems which are typically solved by defining a system of linear equations, and do not require search over states. We modified them to include fewer constraints than unknown variables, allowing for multiple correct solutions, and defined a

Table 1: An instance of the 'Coin Exchange' problem: Green indicated instance-specific components, and orange highlights the modified rules specific to SearchBench. GPT4 fails to generate a feasible solution for this instance using the three baseline prompting methods, but produces a correct, non-optimal solution using A* and MSMT A*.

Problem statement
In the 'taxed coin exchange' problem, you are required to choose a subset of coins from this list [3, 6, 9, 10, 13, 15, 18, 5, 21, 19, 12, 15, 5, 9, 4, 16, 8, 4, 7, 7, 7, 2, 16, 14, 18, 3, 89, 21, 12, 10, 7, 14, 4, 11, 6, 20], such that the sum of the chosen coins adds up to 229. Each coin in the list is unique and can only be used once. Also coins carry a tax value. The tax values for each coin is 14: 1, 89: 13, 2: 2, 5: 2, 4: 4, 6: 6, 8: 2, 16: 5, 21: 4, 20: 2, 18: 9, 11: 10, 10: 3, 12: 12, 15: 5, 13: 1, 3: 1, 19: 19, 7: 7, 9: 3, where the tax for coins of the same value is the same. Also, if the coin chosen is smaller than the previous one, it must have an even value, otherwise, if the coin is larger than or equal to the previous coin chosen, it must have an odd value. The objective is to determine which subset of coins should be selected to minimize the total tax paid. The solution should be presented as a list of numbers, representing the value of the coins chosen in order, with the first coins chosen being in index 0, formatted in Python syntax.

unique cost function to enable search for a single optimal solution. This category was added in order to evaluate models' ability to generalize to novel combinatorial problems.

We selected 2-3 problem types for each category, resulting in 11 total problem types. Each type has a unique state space. For example, in 8-puzzle words, each state is an $n \times m$ table of characters, while in coin exchange, each state is an ordered subset of given coins. Generally, our problems involve an initial state, a goal state, and a set of possible actions, and the task is to find a sequence of actions from the initial to the goal state with minimum cost. We modified the rules to ensure that solved instances of SearchBench were not encountered during the LLMs' massive internet-scale training. The SearchBench taxonomy and rule modifications are illustrated in Fig. 1.

To construct SearchBench, we implemented an automatic generation pipeline for each problem type, ensuring each generated instance is solvable. We generated approximately 100 instances per type, resulting in a total of 1107 problem instances. The benchmark is then fixed. The generation pipelines can create instances with adjustable difficulty levels. Difficulty is defined by the state space size of the instance, with minimum difficulty requiring a few actions and maximum difficulty set such that problems could be solved correctly but not optimally by humans (See Appendix Sec. F for an analysis of the search space size). Hence, maximum human performance on SearchBench could be considered approximately 100%. Moreover, studies like (Pizlo & Li, 2005; Chronicle et al., 2006) show that humans can solve the classic versions of SearchBench problems, but their performance declines as the state space size increases.

In contrast to other reasoning benchmarks (Saparov & He, 2022; Cobbe et al., 2021; Hendrycks et al., 2021; Patel et al., 2021; Clark et al., 2020; Tafjord et al., 2020; Sap et al., 2019; Le et al., 2019) that only measure correctness, to gain a more comprehensive understanding of LLM performance on SearchBench, our evaluation pipeline assesses LLM solutions across 3 dimensions: Feasibility, Correctness, and Optimality. Feasibility determines if any of the actions chosen violate the problem rules (e.g. passing through labyrinth walls). Correctness requires that the solution is both feasible and reaches the goal state from the given start state. Optimality indicates that the solution is both correct and has the minimum cost w.r.t. known optimum. For each SearchBench problem, we implemented a fast A* algorithm with a provably admissible and consistent heuristic, to produce the optimal solution. We ran this implementation for each instance in the benchmark to obtain its unique optimal solution.

We note that even though correctness implies feasibility, and optimally implies correctness, feasibility and correctness are valuable intermediate metrics in determining how close the models are to generating the fully correct solution. For example, in traffic problems, GPT4 often fails to record the first city visited, resulting in a feasible but incorrect solution. Defining feasibility helps distinguish this mostly correct implementation from more erroneous solutions. Correctness is stricter than feasibility and indicates that search-related tasks were implemented correctly, but the heuristic or recorded cost is incorrect, leading to non-optimal solutions.

3 Evaluated Methods

We use three baseline prompting methods to evaluate LLMs on SearchBench: 0-shot text, 4-shot CoT text, and 0-shot code. Additionally, we use two new code-based methods: 4-shot A* prompting and MSMT A*. See supplementary materiel for full prompts for each of the five approaches and GPT4's generated solution.

To ensure the generality of our prompting methods, we selected one in-context example from each of the four SearchBench categories that are different from the category of the evaluated problem. This minimizes the similarity between the rules and context of the solved examples and the evaluated problem, and tests whether the model can solve unrelated combinatorial problems. Thus, if a model finds an optimal solution using these methods, it demonstrates true generalization rather than prompt-specific improvements. In Sec. 6, we further analyze the impact of including an example from the same top-level problem category. Additionally, 4-shot is the upper limit on the number of in-context examples due to the models' context length limit. For an analysis of the effect of fewer demonstrations (shots) on performance, see Appendix Sec. B.

0-shot text and 4-shot CoT text prompting methods: In the text-based prompting methods, we instruct the model to solve the problem in an end-to-end manner, using text only. In 4-shot CoT prompts, the in-context examples include a representation of the intermediate states drawn using ASCII characters after each action to prevent hallucinations and illogical leaps in reasoning.

0-shot code prompting method: This method instructs the LLM to produce a Python code that solves the given problem. The generated code is then executed to derive the final answer.

A* Prompting: In this approach, we prompt the LLM to implement an A* algorithm that solves $\mathcal{P}_i^C$ - a problem instance number i of problem category C, providing four in-context examples of A* codes for four problems $\mathcal{P}_j^{\hat{C}}$, each selected from a different category $\hat{C} \neq C$. To implement A* for the target SearchBench problem, the LLM must perform abstract reasoning to devise a search strategy applicable to any state within the search space. This contrasts with solving problems end-to-end in text, where the model has access to the variables of each state, eliminating the need for abstract reasoning. However, end-to-end approaches require the model to perform every step of the iterative computations involved in searching the state space.

The in-context examples include detailed comments before each code segment, explaining the reasoning used to develop the strategy implemented within the code segment. These comments serve as CoT reasoning for devising the search strategy implemented in the code.

Multi-Stage-Multi-Try (MSMT) A* Prompting: In this method, the model receives the same in-context examples as the 'A* prompting', with different instructions. Here, inference is done in two stages as demonstrated in Fig. 2. In the first stage, the model is instructed to implement the code as two functions: the 'a_star' function includes an instance-agnostic A* algorithm for the target problem type, and the 'initialize' function encodes the variables given in the problem statement. We then verify if the generated code satisfies the following set of unit tests: (i) code is executable; (ii) code returns a list; (iii) and the list elements match the data type specified by the problem statement. If the code fails any unit test, MSMT re-generate the code. Next, in the second stage, the LLM is instructed to implement an 'initialize' function, conditioned on the verified 'a_star' function from stage 1 for each instance of the problem type. The inclusion of simple unit tests, which can be expanded to more detailed tests if needed, offers a robust method for filtering out erroneous samples from the model's generations.

In our MSMT A* prompting approach, the model generates the full A* algorithm end-to-end without any external feedback, similar to how text-based prompting methods operate. Importantly, our MSMT A* does not rely on the majority vote of multiple solutions. Instead, the solution returned by the first model-generated code that passes the unit tests is taken as the final answer. This results in increased efficiency of MSMT A*, requiring only up to 1.5x number of inferences per problem on average compared to 5x-100x in majority vote approaches (Wang et al., 2022).

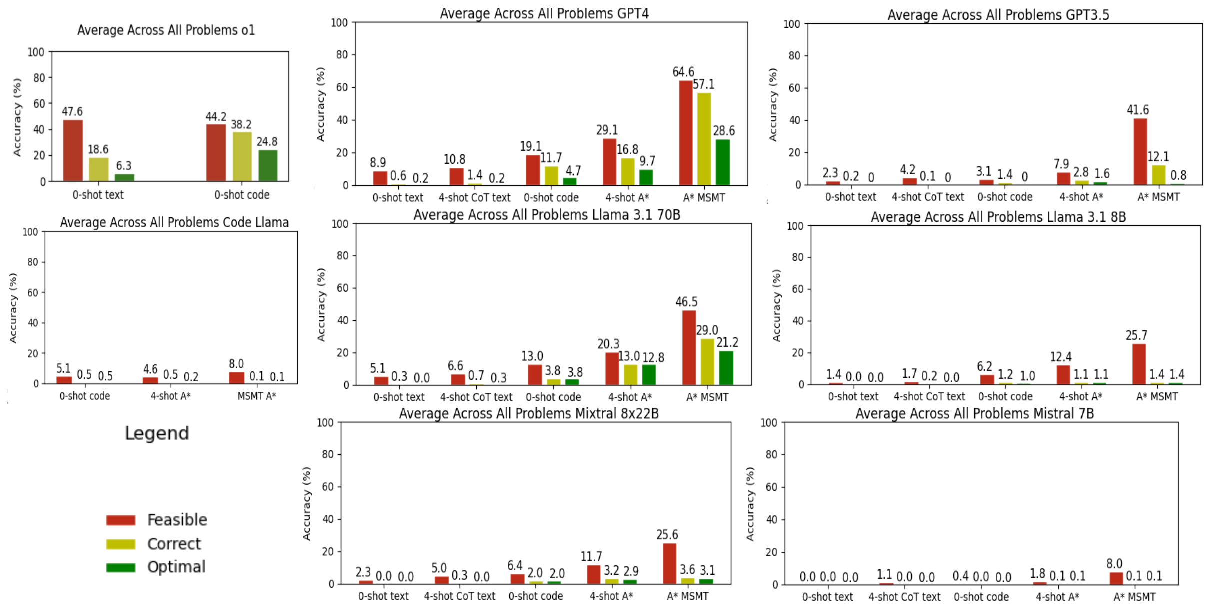


Figure 3: Average rate of feasible, correct, and optimal solutions for all problems using o1, GPT4, GPT3.5, Code Llama, Llama 3.1 70B, Llama 3.1 8B, Mixtral 8x22B, and Mistral 7B.

4 Related Work

Mathematical and Reasoning Benchmarks: Evaluating LLMs (Brown et al., 2020; OpenAI, 2023; 2022; Chung et al., 2024; Chowdhery et al., 2022; Rae et al., 2021; Taylor et al., 2022; Thoppilan et al., 2022) on mathematical and reasoning tasks has been a focus of recent research in natural language processing, leading to the development of benchmarks such as BIG-BENCH (Srivastava et al., 2022), GSM8K (Cobbe et al., 2021), AQUA (Ling et al., 2017), SVAMP (Patel et al., 2021), CommonsenseQA (Talmor et al., 2018), StrategyQA (Geva et al., 2021), and MATH (Hendrycks et al., 2021). However, these benchmarks have limitations. GSM8K problems are relatively simple and often require a repetitive reasoning pattern to solve, and problems in BIG-BENCH are mostly single-step reasoning tasks. The MATH dataset, while more challenging, may not accurately assess a model's generalizable reasoning capabilities due to the advanced mathematical skills required. When prompted to solve problems using CoT prompting in text, LLMs perform well on these tasks; however, they fail on SearchBench's problems, indicating that these benchmarks offer limited insight into LLMs' ability to systematically explore a state space.

Application of LLMs to Combinatorial Problems: Few studies such as (Yang et al., 2023; Liu et al., 2024; Masoud et al., 2024) have explored solving select combinatorial problems like the Traveling Salesman Problem with LLMs. (Mittal et al., 2024) introduced a dataset of combinatorial problems, "PuzzleBench". However, they only selected problems that can be represented in a symbolic solver (SMT2.0) and assumed there exists fixed pre-defined symbolic representations for input problems and outputs, limiting their datasets' generalizability. Moreover, problems selected by (Mittal et al., 2024) and (Iklassov et al., 2024) are instances of the classical combinatorial problems, raising issues of memorization, as algorithm implementations for instances of such problems are available online.

SearchBench stands out in several ways (i) Generalizability: Unlike PuzzleBench, SearchBench problems are described only in natural language, with no restrictions on rules or actions, and cover a wide verity of combinatorial problems, ensuring that a model capable of solving SearchBench can generalize to other combinatorial problems. (ii) Uniquely Modified Rules: This prevents memorization, as algorithms for classic versions of the problem are available online. (iii) Optimal Solution: Each problem type has a uniquely defined cost, ensuring a single optimal solution and avoiding multiple valid answers. (iv) Multi-Dimensional Evaluation: This provides deeper insights into how close models are to deriving the unique optimal solution.

(v) Automated Instance Generation: This avoids data leakage or contamination, as new instances can be generated on demand.

Prompting Strategies: Sophisticated prompting strategies have been developed to enhance models' reasoning abilities. One notable approach is Chain-of-Thought (CoT) prompting (Wei et al., 2022), which prompts LLMs to generate intermediate steps leading to the final output. This technique has led to advanced variations, including Tree-of-Thoughts (Yao et al., 2023a; Long, 2023), and Graph-of-Thought (Yao et al., 2023b; Lei et al., 2023; Besta et al., 2023) methods that maintain a tree of intermediate generations. However, these methods rely on evaluating and rejecting intermediate steps, which does not integrate well with our problems. In search problems, intermediate states can't be easily classified as correct or incorrect, and all possible actions must be considered to find the optimal solution. Additionally, the state space of combinatorial problems grows exponentially, making it impractical for LLMs to navigate the frontier of the search tree without incorrectly disregarding most feasible states.

Other prompting methods, such as Decomposition strategies (Khot et al., 2022; Zhou et al., 2022; Zhang et al., 2023), simplify complex tasks into smaller, manageable subtasks to improve performance. Additionally, systems like LLM-Augmenter (Peng et al., 2023) rely on external databases to verify segments of the LLM's output. In this work, we propose the A* prompting strategy, where we prompt the model to solve problems by implementing a unique A* algorithm. Similarly, our A* MSMT approach decomposes the task of implementing the search algorithm into two stages and checks the model's generations against external validators; we use simple unit tests instead of external data sources or solved solution instances.

5 Experiments

We evaluated the performance of GPT4, GPT-3.5, and Code Llama Instruct 34B (Roziere et al., 2023) [1], Llama 3.1 70B, Llama 3.1 8B, Mixtral 8x22B (Mistral, 2023b), and Mistral 7B (Mistral, 2023a) on SearchBench, using the following five prompting methods described in Sec. 3: 0-shot text, 4-shot CoT text, 0-shot code, 4-shot A*, and 4-shot MSMT A*. Results are summarized in Fig. 3.

Implementation details: We used GPT4, GPT-3.5 Turbo (GPT3.5 hereafter), and o1-preview (o1 hereafter) via OpenAI platform APIs. All code evaluations were conducted on a machine with 96 64-bit Intel Xeon Gold 5220R CPUs, a maximum speed of 4GHz, and a 71.5 MiB Level 3 cache.

0-shot text and 4-shot CoT text prompting methods: As shown in Fig. 3, the correct solutions rate is below 1% for all of the models using 0-shot text prompting, and less than 9% of GPT4 solutions are feasible (follow the problem rules) using this method. This is expected as the exponentially growing state space size of SearchBench problems and the difficulty of backtracking during auto-regressive generation make it challenging to solve SearchBench problems using text-based prompting, even with the strongest LLMs. Moreover, 4-shot CoT text prompting only improves the rate of feasible solutions generated by less than 3% for all models. This shows that the inherent complexity of search problems from SearchBench cannot be effectively addressed by text-based prompting alone.

Finally, we evaluate the recent o1 model (OpenAI, 2024), which is designed for comprehensive reasoning. As shown, this model struggles with SearchBench problems, solving less than 19% correctly with 0-shot text; however, it significantly outperformed other models in solving the problems end-to-end. We did not evaluate this model with A* prompting and MSMT A* due to its limited context length. Additionally, o1 solves problems with scaled compute at inference time, making MSMT unnecessary.

0-shot code prompting method: This prompting method improves performance over text-based prompting for all models except Mistral 7B, which remained close to 0%. This is expected, as using Python to compute intermediate steps and execute the iterations of the algorithms devised by the LLMs reduces the load on the models. As seen in Fig. 3, o1 solved 38.2% of the problems correctly, 19.1% of GPT4's code generations result in a feasible solution, with only 11.7% being correct. The next best performance was achieved by Llama 3.1 70B, which solved 13% of the problems correctly. For an analysis of the computation time of programs generated by the LLMs, please refer to Appendix Sec. D.

[1]Finetuned on the Phind dataset (Phind, 2023)

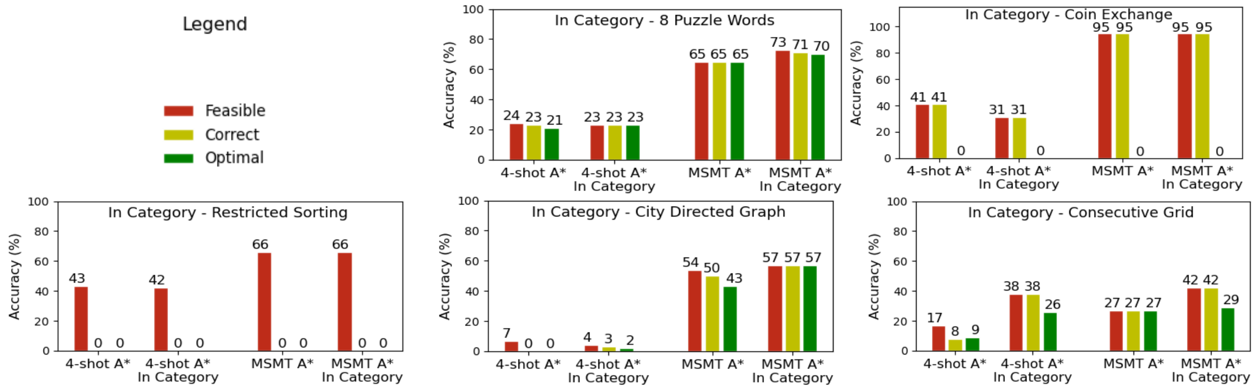


Figure 4: Comparing GPT4's performance, using A* prompting approaches, when one of the in-context examples is switched to a problem that shares the same category as the inference problem.

A* Prompting: As shown in Fig. 3, A* prompting improves the performance of all models on SearchBench except for Code Llama, which shows almost no improvement, indicating potential limitations of this model in in-context learning or following the given instructions. GPT-4's rates of feasible, correct, and optimal solutions increase by 10%, 5%, and 5%, respectively, and Llama 3.1 70B's rates increase by 7%, 9%, and 9%.

MSMT A*: As shown in Fig. 3, MSMT A* prompting significantly boosts the performance of all models. Using this method, GPT4 correctly solved 57.1% of SearchBench problems and achieved a 28.6% rate of optimal solutions, outperforming o1. GPT4's performance improved consistently across all problem types compared to other prompting strategies (see Appendix Sec. C for for a detailed analysis of performance on each problem type). Other LLMs also showed strong improvements, except for Code Llama, which only improved in feasibility due as it still struggled to follow the instructions. However, the 28.6% optimal performance of GPT4 using MSMT A*, although inspiring, still leaves room for further improvements, underlining the importance of SearchBench for future research.

6 Ablations and Analysis

Here we provide a comprehensive analysis of SearchBench using GPT4. For further analysis, please refer to Appendix Sec. B, C, D, and F.

Does including a more similar problem in prompt improve GPT4's performance? In our main experiments with A* and MSMT A* (Fig. 3), we used four in-context examples, each from a different category than the target problem (Sec. 3). This ensured that no exact segment of the target solution was included in the prompt, hence better measuring LLM's reasoning generalization. Here, we evaluated GPT4's performance when a solved example from the same category but a different type as the evaluated problem, is included in the prompt. Results are summarized in Fig. 4. We observed small improvements, with up to 15 additional instances solved. This indicates that SearchBench problems within the same category still differ significantly in rules, constraints, and target A* algorithm implementations.

The most significant improvement was observed for the Consecutive Grid problems from the underdetermined systems category which involve filling in masked numbers according to constraints on the order of integers in a table. This category differs more significantly from other combinatorial problems, showing that including similar problems in the prompt leads to greater improvement for new tasks.

What types of runtime errors occur, and how often, when executing GPT4's implementations? We analyzed the errors returned by GPT4's generated codes that resulted in infeasible solutions. The results are shown in Fig. 5. We categorized errors into six types: (i) Solution Not Feasible: code executed but returned an infeasible solution; (ii) Program Returned None: program failed to find a solution; (iii) Program Killed: program did not finish within the allotted time; (iv) Incorrect Solution Type: returned solution had the wrong data type; (v) Incomplete Generation: model ran out of tokens; and (vi) Program Didn't Compile.

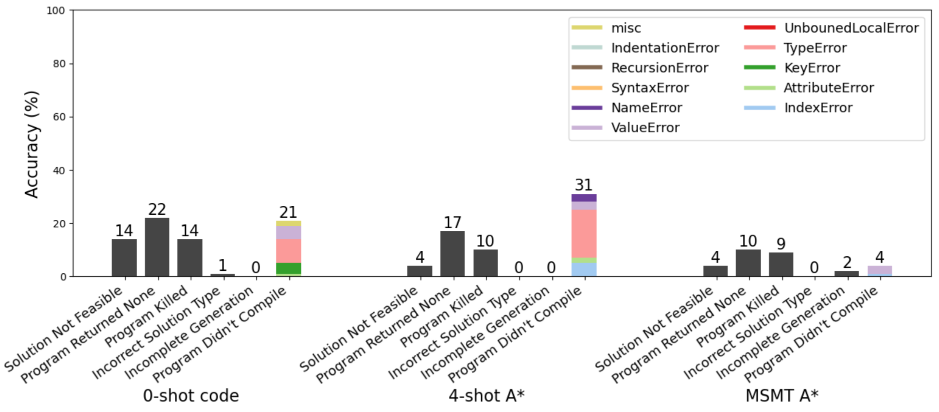


Figure 5: Rate of errors returned by python programs generated by GPT4, categorized into 6 error types, calculated across all SearchBench problems with an infeasible solution.

As shown in Fig. 5, prompting the model with the A* method results in more non-compiling implementations compared to 0-shot code prompting. This is expected as A* is more complex and longer compared to the simpler algorithms typically implemented by the model using 0-shot code prompting, such as the greedy algorithm, BFS, or DFS. However, the number of infeasible solutions decreases with A* prompting, indicating that the model can better reason about the problem using this method. Comparing A* prompting to the MSMT A* method, we observe a significant decrease in errors that fail at least one unit test, such as 'Program Returned None', 'Program Killed', 'Incorrect Solution Type', 'Incomplete Generation', and 'Program Didn't Compile'.

What are the most common reasoning errors made in GPT4's A* implementations? We manually analyzed 50 A* codes generated by GPT4 that returned non-optimal solutions across five problems types: three pathfinding problems and two puzzle problems. These problems were chosen because GPT4 showed the least and greatest performance improvement, respectively, using A* prompting compared to 0-shot code (see Appendix Sec. 3). We identified seven distinct failure modes in the GPT4-generated A* implementations, each corresponding to a critical non-optimal within the overall search strategy, where failing any subtask results in a suboptimal solution. The results are summarized in Table 2, showing the percentage of 'correct reasoning' for each subtask, excluding coding errors. As shown, in pathfinding problems, the most common mistake was not recording the list of visited coordinates, a 13% success rate, with the model often omitting the start coordinate in the path, leading to feasible but incorrect solutions. In puzzle problems, the frequent error was encoding the goal state, likely because our puzzles have unique goal states unlike the conventional 8-puzzle problem.

Table 2: The average accuracy of GPT4 on identified A* subtasks (failure modes) was analyzed using 50 code implementations for pathfinding and puzzle problems.

	Pathfinding Problems	**Puzzle Problems**
Encoding Initial State	47%	100%
Encoding Goal State	74%	20%
Recording the Path	13%	70%
Exit Condition	70%	100%
Iterating Successor States	57%	100%
Generating New State	87%	100%
Admissible & Consistent Heuristic	93%	60%

7 Conclusion

In this work, we introduced SearchBench, a pioneering benchmark designed to evaluate LLMs' ability to solve new combinatorial search problems that require backtracking and considering multiple action sequences. We assessed LLMs using various text-based and code-based prompting methods and showed that while models fail at solving these puzzles step by step in natural language, their performance improves considerably when asked to implement an A* search algorithm—a cognitively more challenging task for humans but one that shifts the state space exploration burden from the model to code execution. This contrast reveals a key bottleneck in iterative reasoning and backtracking for autoregressive models, while also highlighting how their strengths in structured code generation, validated by simple unit tests, can be harnessed to overcome these limitations. For broader impact, see Appendix Sec. G.

8 Limitations

The primary challenge in developing SearchBench was scaling the number of problem types. Designing unique search problems and creating pipelines to generate numerous instances with guaranteed solutions is both time-consuming and complex. Additionally, implementing a fast, instance-agnostic A* algorithm and developing evaluation pipelines to assess LLM-proposed solutions on multiple criteria further adds to the complexity.

References

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models. *ArXiv*, abs/2308.09687, 2023. URL `https://api.semanticscholar.org/CorpusID:261030303`.

Ning Bian, Xianpei Han, Le Sun, Hongyu Lin, Yaojie Lu, Ben He, Shanshan Jiang, and Bin Dong. Chatgpt is a knowledgeable but inexperienced solver: An investigation of commonsense problem in large language models. *arXiv preprint arXiv:2303.16421*, 2023.

Nasim Borazjanizadeh and Steven T Piantadosi. Reliable reasoning beyond natural language. *arXiv preprint arXiv:2407.11373*, 2024.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam M. Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Benton C. Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier García, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Díaz, Orhan Firat, Michele Catasta, Jason Wei, Kathleen S. Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24:240:1–240:113, 2022. URL `https://api.semanticscholar.org/CorpusID:247951931`.

Edward Chronicle, James Macgregor, Thomas Ormerod, and Alistair Burr. It looks easy! heuristics for combinatorial optimization problems. *Quarterly journal of experimental psychology (2006)*, 59:783–800, 05 2006. doi: 10.1080/02724980543000033.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70):1–53, 2024.

Peter Clark, Oyvind Tafjord, and Kyle Richardson. Transformers as soft reasoners over language. *arXiv preprint arXiv:2002.05867*, 2020.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, et al. Faith and fate: Limits of transformers on compositionality. *Advances in Neural Information Processing Systems*, 36, 2024.

Mor Geva, Daniel Khashabi, Elad Segal, Tushar Khot, Dan Roth, and Jonathan Berant. Did aristotle use a laptop? a question answering benchmark with implicit reasoning strategies. *Transactions of the Association for Computational Linguistics*, 9:346–361, 2021.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Zangir Iklassov, Yali Du, Farkhad Akimov, and Martin Takac. Self-guiding exploration for combinatorial problems. *arXiv preprint arXiv:2405.17950*, 2024.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*, 2022.

Matthew Le, Y-Lan Boureau, and Maximilian Nickel. Revisiting the evaluation of theory of mind through question answering. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (eds.), *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 5872–5877, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1598. URL `https://aclanthology.org/D19-1598`.

Bin Lei, Pei-Hung Lin, Chunhua Liao, and Caiwen Ding. Boosting logical reasoning in large language models through a new framework: The graph of thought. *ArXiv*, abs/2308.08614, 2023. URL `https://api.semanticscholar.org/CorpusID:261030743`.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. *arXiv preprint arXiv:1705.04146*, 2017.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. An example of evolutionary computation+ large language model beating human: Design of efficient guided local search. *arXiv preprint arXiv:2401.02051*, 2024.

Jieyi Long. Large language model guided tree-of-thought. *arXiv preprint arXiv:2305.08291*, 2023.

Gary Marcus. The next decade in ai: four steps towards robust artificial intelligence. *arXiv preprint arXiv:2002.06177*, 2020.

Mahmoud Masoud, Ahmed Abdelhay, and Mohammed Elhenawy. Exploring combinatorial problem solving with large language models: A case study on the travelling salesman problem using gpt-3.5 turbo. *arXiv preprint arXiv:2405.01997*, 2024.

Meta. Introducing llama 3.1: Our most capable models to date, 2024. URL `https://ai.meta.com/blog/meta-llama-3-1/`.

Mistral. Mistral 7b, 2023a. URL `https://mistral.ai/news/announcing-mistral-7b/`.

Mistral. Mixtral of experts, 2023b. URL `https://mistral.ai/news/mixtral-of-experts/`.

Chinmay Mittal, Krishna Kartik, Parag Singla, et al. Puzzlebench: Can llms solve challenging first-order combinatorial reasoning problems? *arXiv preprint arXiv:2402.02611*, 2024.

OpenAI. Chatgpt: Optimizing language models for dialogue, 2022. URL `https://openai.com/index/chatgpt/`.

OpenAI. Gpt-4 technical report. *ArXiv*, abs/2303.08774, 2023. URL `https://api.semanticscholar.org/CorpusID:257532815`.

OpenAI. Learning to reason with llms, 2024. URL `https://openai.com/index/learning-to-reason-with-llms/`.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. Are nlp models really able to solve simple math word problems? *arXiv preprint arXiv:2103.07191*, 2021.

Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, et al. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813*, 2023.

Phind. Beating gpt-4 on humaneval with a fine-tuned codellama-34b, 2023. URL `https://www.phind.com/blog/code-llama-beats-gpt4`.

Zygmunt Pizlo and Zheng Li. Solving combinatorial problems: The 15-puzzle. *Memory & cognition*, 33: 1069–84, 10 2005. doi: 10.3758/BF03193214.

Chengwei Qin, Aston Zhang, Zhuosheng Zhang, Jiaao Chen, Michihiro Yasunaga, and Diyi Yang. Is chatgpt a general-purpose natural language processing task solver? *arXiv preprint arXiv:2302.06476*, 2023.

Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John F. J. Mellor, Irina Higgins, Antonia Creswell, Nathan McAleese, Amy Wu, Erich Elsen, Siddhant M. Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, L. Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, N. K. Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Tobias Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew G. Johnson, Blake A. Hechtman, Laura Weidinger, Iason Gabriel, William S. Isaac, Edward Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem W. Ayoub, Jeff Stanway, L. L. Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training gopher. *ArXiv*, abs/2112.11446, 2021. URL `https://api.semanticscholar.org/CorpusID:245353475`.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Maarten Sap, Hannah Rashkin, Derek Chen, Ronan Le Bras, and Yejin Choi. Social IQa: Commonsense reasoning about social interactions. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (eds.), *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th*

International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pp. 4463–4473, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1454. URL `https://aclanthology.org/D19-1454`.

Abulhair Saparov and He He. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. *arXiv preprint arXiv:2210.01240*, 2022.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

Oyvind Tafjord, Bhavana Dalvi Mishra, and Peter Clark. Proofwriter: Generating implications, proofs, and abductive statements over natural language. *arXiv preprint arXiv:2012.13048*, 2020.

Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. Commonsenseqa: A question answering challenge targeting commonsense knowledge. *arXiv preprint arXiv:1811.00937*, 2018.

Ross Taylor, Marcin Kardas, Guillem Cucurull, Thomas Scialom, Anthony Hartshorn, Elvis Saravia, Andrew Poulton, Viktor Kerkez, and Robert Stojnic. Galactica: A large language model for science. *arXiv preprint arXiv:2211.09085*, 2022.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.

Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Large language models still can't plan (a benchmark for llms on planning and reasoning about change). *arXiv preprint arXiv:2206.10498*, 2022.

Karthik Valmeekam, Raymond Nguyen, Quoc V Le, Lin Li, Nate Kushman, and Oleksandr Polozov. Planning with large language models for code generation. *arXiv preprint arXiv:2302.04761*, 2023.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Huai hsin Chi, F. Xia, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *ArXiv*, abs/2201.11903, 2022. URL `https://api.semanticscholar.org/CorpusID:246411621`.

Wikipedia. Breadth-first search, 2025a. URL `https://en.wikipedia.org/wiki/Breadth-first_search`.

Wikipedia. Depth-first search, 2025b. URL `https://en.wikipedia.org/wiki/Depth-first_search`.

R.J. Wilson. *Combinatorics: A Very Short Introduction*. Very short introductions. Oxford University Press, 2016. ISBN 9780191790324. URL `https://books.google.com/books?id=r2hdAQAACAAJ`.

Zhaofeng Wu, Linlu Qiu, Alexis Ross, Ekin Akyürek, Boyuan Chen, Bailin Wang, Najoung Kim, Jacob Andreas, and Yoon Kim. Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks. *arXiv preprint arXiv:2307.02477*, 2023.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2023.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *ArXiv*, abs/2305.10601, 2023a. URL `https://api.semanticscholar.org/CorpusID:258762525`.

Yao Yao, Z. Li, and Hai Zhao. Beyond chain-of-thought, effective graph-of-thought reasoning in large language models. *ArXiv*, abs/2305.16582, 2023b. URL `https://api.semanticscholar.org/CorpusID:258947684`.

Tianhua Zhang, Jiaxin Ge, Hongyin Luo, Yung-Sung Chuang, Mingye Gao, Yuan Gong, Xixin Wu, Yoon Kim, Helen Meng, and James Glass. Natural language embedded programs for hybrid language symbolic reasoning. *arXiv preprint arXiv:2309.10814*, 2023.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

A Appendix

B n-shot Ablation Experiments

To examine the effect of different numbers of demonstrations on GPT4's performance using A* and MSMT A* prompting methods, we performed ablation experiments with 2-shot and 3-shot A* prompts. 4-shot is upper limit on the number of in-context examples due to the context length constraints of the models, including GPT4. In all few-shot experiments, the examples used in the prompts were not from the evaluated problem category. The results, summarized in Fig. 6, show a consistent trend of performance improvement with the addition of more examples, as expected.

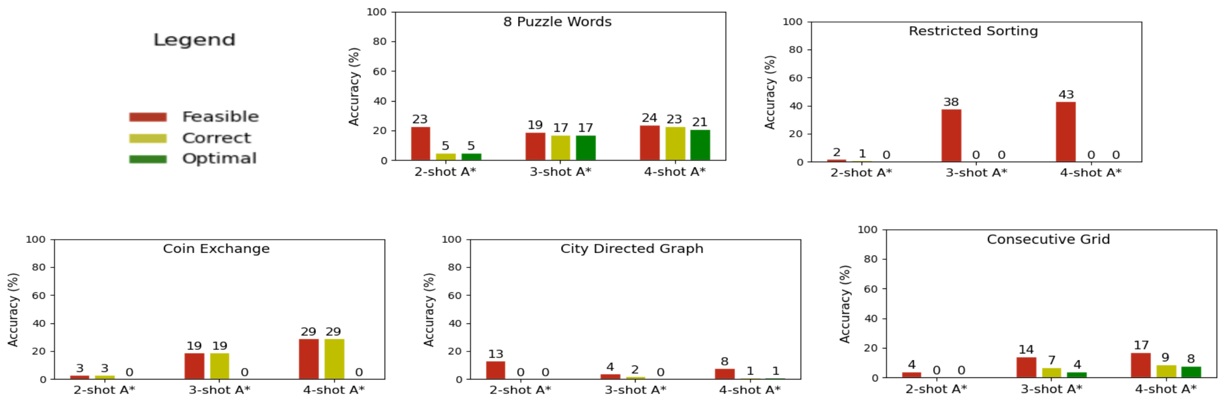


Figure 6: Comparing 2-shot, 3-shot and 4-shot performance of GPT4 between A*-prompting approaches.

C Detailed analysis of GPT4's performance on SearchBench

Tab. 3 details GPT4 code-based method performance for each of SearchBench's 11 problems. Consistently 4-shot A* prompting outperforms 0-shot code for most problems. Interestingly for problems in the pathfinding category, prompting GPT4 with 0-shot code outperforms A* prompting.

Examining closer, GPT4 mainly uses DFS for pathfinding in 0-shot code. While simpler than A*, DFS doesn't guarantee optimal solutions, as reflected in GPT4's high feasible and correct rates but lower optimal rates. Implementing A* with an admissible and consistent heuristic requires the model to implement a more complex strategy in the code involving additional constraints and more sophisticated data structures. This increases the likelihood of reasoning or coding errors, which could explain the dip in GPT4's performance using A* prompting compared to 0-shot code when solving these problems.

Figure 7 further analyzes the relationship between problem difficulty (quantified by state space size of the problem) and the performance of GPT4. As observed, the model's performance is generally higher on easier problems, particularly in terms of the rate of correct solutions. This is expected, as easier problems have a smaller state space to explore. However, the performance of the model does not change drastically across different difficulty levels. This indicates that the combinatorial problems in SearchBench are intrinsically hard for LLMs to solve in text due to the requirement for backtracking. Moreover, the difference in implementing an A* search algorithm for a difficult or easy instance of SearchBench is limited to encoding the initial and goal states. The rest of the algorithm implementation task remains the same. This is the reason why the model's performance is comparable across different difficulty levels, both using text-based and code-based methods.

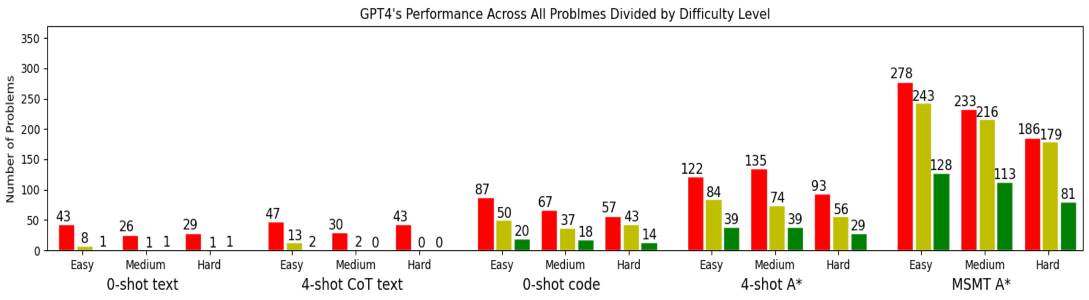


Figure 7: Count of feasible, correct, and optimal solutions generated by GPT4 via code-based methods for 3 levels of problem difficulty.

Table 3: GPT4's performance when prompted with our code-based approaches, on each problem type. The values are percentages of the feasible (F), correct (C), and optimal (O) solutions.

Problem	0-shot code	4-shot A*	MSMT A*
8 Puzzle	F: 3 C: 0 O: 0	F: 63 C: 60 O: 60	F: 76 C: 68 O: 68
8 Puzzle Words	F: 5 C: 5 O: 5	F: 24 C: 23 O: 21	F: 66 C: 65 O: 65
Color Sorting	F: 17 C: 1 O: 1	F: 41 C: 35 O: 6	F: 91 C: 91 O: 0
Restricted Sorting	F: 32 C: 0 O: 0	F: 43 C: 0 O: 0	F: 66 C: 0 O: 0
Water Jug	F: 7 C: 7 O: 6	F: 8 C: 8 O: 0	F: 95 C: 95 O: 0
Coin Exchange	F: 2 C: 1 O: 0	F: 31 C: 31 O: 0	F: 95 C: 95 O: 0
Traffic	F: 65 C: 50 O: 13	F: 24 C: 5 O: 5	F: 65 C: 60 O: 60
Trampoline Matrix	F: 27 C: 27 O: 22	F: 51 C: 4 O: 4	F: 57 C: 53 O: 46
City Directed Graph	F: 29 C: 28 O: 1	F: 7 C: 0 O: 0	F: 55 C: 51 O: 45
Magic Square	F: 3 C: 1 O: 0	F: 8 C: 5 O: 0	F: 14 C: 14 O: 0
Consecutive Grid	F: 15 C: 2 O: 0	F: 17 C: 9 O: 8	F: 27 C: 27 O: 27

D Compute Time of LLM-Generated Codes

In this section, we analyze the computation time of programs generated by LLMs that produce correct solutions. We compare this time to the duration required to calculate the optimal solution for the problem instance using our fast A* implementation. This comparison provides insights into the efficiency of the algorithms generated by the LLMs. The average compute time of LLM-generated codes, normalized against the compute time of our A* implementation for the given instance, is reported in Fig. 8.

Our findings indicate that LLM-generated implementations are significantly slower than our A* implementation. Specifically, GPT4's A* implementations were 213 times slower than the optimal A* solution, suggesting that GPT4's heuristics are still less efficient. Additionally, on average, GPT4's 0-shot code generations that return a correct solution run 900 times slower than the optimal A* implementation. These results underscore the intrinsic difficulty of SearchBench problems, even when addressed through code generation.

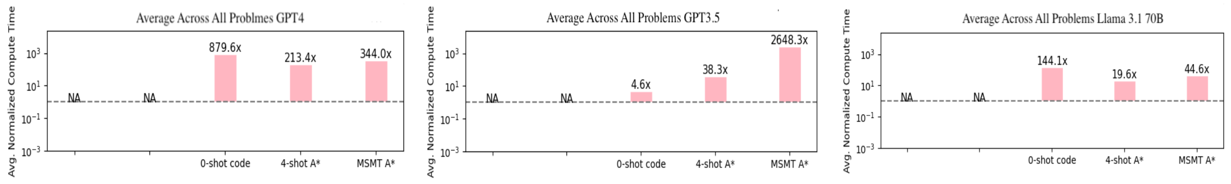


Figure 8: Average compute time of codes returning a correct solution normalized against the compute time of out A* implementation for all problems using GPT4, GPT3.5, Llama 3.1 70B.

E SearchBench Variables

Table 4: This table provides a description of each column in SearchBench. Each row in SeacrhBench is an specific problem instance, and columns are fields of each instance.

Variables	
diff_sorted_id	A unique numeric identifier assigned to each problem instance within a specific problem type. These identifiers are ordered by difficulty level, that is the problem instance with diff_sorted_id of 1 is easier than the instance with diff_sorted_id of 50.
problem_statement	A natural language description that outlines the problem to be solved. The problem statement is the sole piece of information given to language models when they are instructed to solve SearchBench problems.
problem_type	Indicates the problem type, out of 11 problem types in SearchBench, that this particular problem is an instance of.
problem_category	The specific category, out of the five predefined problem categories in SearchBench, to which this problem belongs.
relative_diff_score	A numeric score that indicates the difficulty of this problem instance relative to other instances within the same problem type. This value is not comparable across different problem types.
opt_solution	A list of actions that, starting from the given initial state, lead to the goal state with the minimum cost as defined by the problem's criteria.
opt_solution_cost	The cost of the optimal solution for this problem instance.
opt_solution_compute_t	The time, in seconds, that our instance-agnostic A* implementation for the problem type took to solve this specific problem instance.
solution_depth	The number of actions required to reach the goal state from the given initial state with the minimum cost. This metric can be used to calculate an upper bound on the size of the search tree, represented as b^d, for this instance, where, b is an upper bound on the branching factor of the tree, which indicates the maximum number of actions leading to successor states from any given state, and d is the solution depth, representing the number of actions in the optimal solution.
max_successor_states	The maximum number of successor states that can be reached from any given state in this problem. This value is an upper bound on the branching factor of the state search tree for this problem.
num_vars_per_state	An upper bound on the number of variables in each state of the problem. Given that the number of states grows exponentially for SearchBench problems, this value provides an estimate of the memory required to traverse the search tree of the problem.
is_feasible_args	A list of variables of the problem instance that must be passed to the 'is_feasible' function of the evaluation pipeline to determine whether a suggested solution adheres to the rules and constraints of the problem.
is_correct_args	A list of variables in the problem statement of this instance that must be passed as arguments to the 'is_correct' function in the evaluation pipeline, in order to evaluate the correctness of a suggested solution.
A*_args	Variables of this problem instance that must be passed to our A* implementation for the problem type to obtain the optimal solution for the instance.

Table 5: Statistics of metrics pertaining to the search-tree-size of a specific instance, compared across all instances within SearchBench.

Statistics

name	type	min	median	max	mean	standard deviation	missing
opt_solution_compute_t	float (seconds)	0.018	0.068	599.044	17.363	67.513	0%
solution_depth	int	4	14	46	15.516	7.89	0%
max_successor_states	int	4	12	132	24.633	24.622	0%
num_vars_per_state	int	2	13	60	14.785	12.05	0%

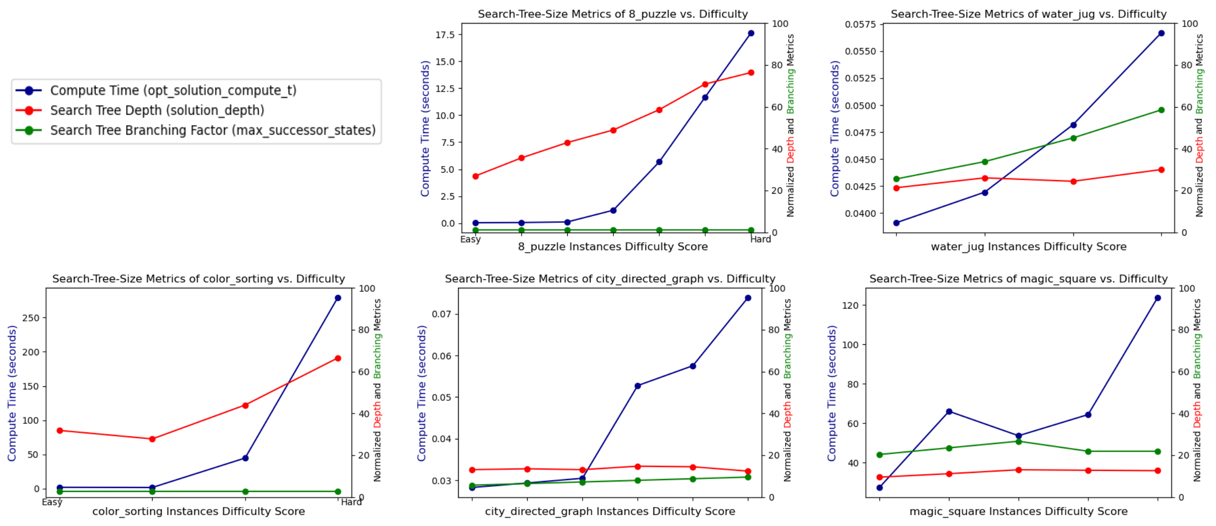


Figure 9: The plots depict the correlation between the increasing difficulty level and the corresponding increase in three metrics: the average depth of the solutions, the branching factor of the state search tree, and the exponential growth of the time required by our A* algorithm to solve the instances, demonstrated across five problem types in SearchBench.

F Search Tree Size Analysis

Figure 9 presents the relationship between the size of the state search tree and the difficulty levels of instances in SearchBench. It displays the average solution-depth and max_successor_state (normalized against the maximum and minimum solution_depth and max_successor_state across all instances in SearchBench) for one problem type from each of the five categories in SearchBench. Additionally, it shows the time our A* algorithm took to navigate the search tree for instances of variable difficulty (compute time is averaged across instances with the same difficulty). We used a machine with 96 64-bit Intel Xeon Gold 5220R CPUs with a maximum speed of 4GHz, and 71.5 MiB Level 3 cache to run the A* implementations.

The figure shows that the solution depth increases linearly with the difficulty scores of problem instances. However, for the city graph, it remains relatively constant, suggesting that the optimal number of hops to reach a destination node from a start node is consistent for our chosen range of directed graph connectivity and sizes (10 to 15 nodes).The max_successor_states, which represents the upper bound on the number of actions leading to successor states from each state, either remains constant or grows linearly with increasing difficulty level. This metric indicates the branching factor of the search tree size.

However, the compute time required to navigate this search tree grows much faster, exponentially, for most problems, as expected, given the search tree size is b^d, where b is the branching factor, and d is the solution depth. It's worth noting that we used a fast heuristic A* algorithm, which doesn't navigate the full search tree. An exhaustive algorithm like BFS, which explores every node, would result in a much faster exponential growth of compute times. In our experiments, a BFS implementation didn't finish executing even for some of the easiest instances within a 12-hour window.

G Broader Impact

Our research, which aims to assist the development of models capable of general reasoning and reliable problem-solving, has the potential to yield significant societal benefits. Combinatorial problems, like those in our dataset, are fundamental in fields such as robotics, logistics, network design, and industrial optimization. Developing models that can tackle unique versions of these problems by designing efficient algorithms or performing systematic searches end-to-end could greatly enhance AI's applicability across various domains. However, this improvement in the reasoning capabilities of language models could also lead to job displacement, as these models could increasingly automate complex tasks traditionally performed by humans.

H Hosting, Licensing, and Maintenance

We accept responsibility for any violations of rights that might have occurred in the curation of this dataset. We affirm that the dataset is composed solely of search problems and does not include any sensitive information. The data and code associated with SearchBench are licensed under the Creative Commons (CC BY-SA) license, ensuring open access and usability for the research community.

To ensure the long-term availability and preservation of the SearchBench dataset, we have hosted it on both Hugging Face and GitHub. Moreover, we will provide full access to the code for prompting and inference methods, as well as automated pipelines for generating and evaluating an arbitrary number of instances though these platforms, after the double blind review period. We are committed to maintaining the dataset on these platforms with continued open access. Additionally, we anticipate releasing future versions of this dataset with increased scalability.