PuzzleBench: Can LLMs Solve Challenging First-Order Combinatorial Reasoning Problems?

Anonymous ACL submission

Abstract

Recent works show that the largest of the large language models (LLMs) can solve many simple reasoning tasks expressed in natural language, without any/much supervision. But, can they also solve challenging first-order combinatorial reasoning problems, such as graph coloring, knapsack and cryptarithmetic? To answer this question, we present PuzzleBench, a dataset of 31 such challenging problems along with a few solved instances for each problem. These problems are all first order, i.e., they can be instantiated with problem instances of varying sizes, and most of them are NP-hard, requiring several reasoning steps to reach the solution. We first observe that LLMs, even when aided by symbolic solvers, perform rather poorly on our dataset. In response, we propose a new approach, Puzzle-LM, which combines LLMs with both symbolic solvers and program interpreters, along with feedback from solved examples, to achieve huge performance gains. Our extensive experimentation and analyses offer new insights into the reasoning abilities and limitations of present-day LLMs.

1 Introduction

Recent works have shown that large language models (LLMs) such as GPT4 (OpenAI, 2023) can reason like humans (Wei et al., 2022a), and solve diverse natural language reasoning tasks, without the need for any fine-tuning (Wei et al., 2022c; Kojima et al., 2022; Zhou et al., 2023; Zheng et al., 2023). We note that, while impressive, most of the tasks studied are simple reasoning problems, generally requiring only a handful of reasoning steps to reach a solution.

We are motivated by the goal of assessing the reasoning limits of modern-day LLMs. In this paper, we study the computationally intensive, first-order combinatorial problems posed in natural language. These problems (e.g., sudoku, knapsack, graph coloring, cryptarithmetic) have long served as important testbeds to assess the intelligence of AI systems (Russell and Norvig, 2010), and strong traditional AI methods have been developed for them. Can LLMs solve these directly? If not, can they solve these with the help of symbolic AI systems like SMT solvers?

To answer these questions, we release a dataset named PuzzleBench, consisting of 31 such problems along with a few solved instances of each problem (see Figure 1). We refer to such problems as *puzzles* / *fcore* (first-order combinatorial reasoning) problems. *Fcore* problems can be instantiated with instances of varying sizes, e.g., 9×9 and 16×16 sudoku. Most of the problems in PuzzleBench are NP-hard and solving them will require extensive planning and search over a large number of combinations. Not surprisingly, our initial experiments reveal that the best of LLMs can only solve less than a third of these problems.

We then turn to recent approaches that augment LLMs with tools for better reasoning. Program-aided Language models (PAL) (Gao et al., 2023) uses LLMs to generate programs, offloading execution to a program interpreter. Logic-LM (Pan et al., 2023) and SAT-LM (Ye et al., 2023) use LLMs to convert questions to symbolic representations, and external symbolic solvers perform the actual reasoning. Our experiments show that, by themselves, their performances are not that strong on PuzzleBench. At the same time, both these methods demonstrate complementary strengths – PAL can handle first-order structures well, whereas Logic-LM is better at complex reasoning.

In response, we propose a new approach named Puzzle-LM, which combines the powers of *both* PAL and symbolic solvers with LLMs. In particular, the LLM generates an instance-agnostic program for an *fcore* problem that converts any problem instance to a symbolic (in our case, SMT) representation. This program passes this representation to a symbolic SMT solver, Z3 (De Moura and

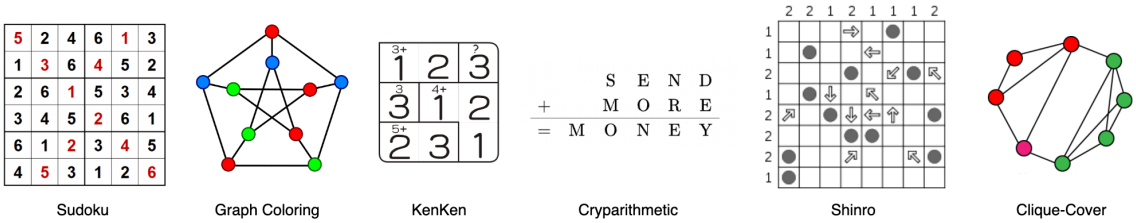


Figure 1: Illustrative Examples of problems in PuzzleBench.

Bjørner, 2008), which returns a solution back to the program. The program then converts the symbolic solution to the desired output representation, as per the natural language instruction. Interestingly, in contrast to LLMs with symbolic solvers, once this program is generated, inference on new *fcore* instances can be done *without* any LLM calls.

Puzzle-LM outperforms few-shot prompting by 29.98, PAL by 11.97 and Logic-LM by 23.14 percent points on PuzzleBench, with GPT-4-Turbo as the LLM. Given the structured nature of *fcore* problems, we also show how solved examples of small sizes can help improve the overall reasoning. Utilizing feedback from solved examples for just four rounds yields a further 20.5 percent points gain for Puzzle-LM. We perform additional analyses to understand the impact of other hyperparameters, and also describe the errors made by Puzzle-LM. We release the dataset and code for further research.

2 Related Work

Neuro-Symbolic AI: Our work falls in the broad category of neuro-symbolic AI (Yu et al., 2023) which builds models leveraging the complementary strengths of neural and symbolic methods. Several prior works build neuro-symbolic models for solving combinatorial reasoning problems (Palm et al., 2018; Wang et al., 2019; Paulus et al., 2021; Nandwani et al., 2022a,b). These develop specialized problem-specific modules (that are typically not size-invariant), which are trained over large training datasets. In contrast, Puzzle-LM uses LLMs and bypasses problem-specific architectures, generalizes to problems of varying sizes and is trained with very few solved instances.

Reasoning with Language Models: The previous paradigm to reasoning was fine-tuning of LLMs (Clark et al., 2021; Tafjord et al., 2021; Yang et al., 2022), but as LLMs scaled, they are found to reason well, when provided with in-context examples without any fine-tuning (Brown et al., 2020; Wei et al., 2022b). Since then, many prompting approaches have been developed that leverage in-context learning. Prominent ones include Chain of Thought (CoT) prompting (Wei et al., 2022c; Kojima et al., 2022), Least-To-Most prompting (Zhou et al., 2023), Progressive-Hint prompting (Zheng et al., 2023) and Tree-of-Thoughts (ToT) prompting (Yao et al., 2023). These works generally study relatively simple reasoning tasks.

Tool Augmented Language Models: Augmenting LLMs with external tools has emerged as a way to solve complex reasoning problems (Schick et al., 2023; Paranjape et al., 2023). The idea is to offload a part of the task to specialized external tools, thereby reducing error rates. Program-aided Language models (Gao et al., 2023) invoke a Python interpreter over a program generated by an LLM. Logic-LM (Pan et al., 2023) and SAT-LM (Ye et al., 2023) integrate reasoning of symbolic solvers with LLMs, which convert the natural language problem into a symbolic representation. Puzzle-LM falls in this category, but combines *both* PAL and symbolic solvers in a specific way to solve *fcore* problems.

Answer Verification: Several existing works verify the correctness of LLM's answers via an external validation, such as via symbolic solvers (Zhang and Callison-Burch, 2023), or external databases (Peng et al., 2023). In the same spirit, Puzzle-LM utilizes solved examples in training set for verifying LLM's output, and provide feedback to LLMs, in case of errors.

Logical Reasoning Benchmarks: There are several natural language reasoning benchmarks in literature, such as LogiQA (Liu et al., 2020) and ReClor (Yu et al., 2020) for mixed reasoning, GSM8K (Cobbe et al., 2021) for arithmetic reasoning, FOLIO (Han et al., 2022) for first-order logic, PrOntoQA (Saparov and He, 2023) and ProofWriter (Tafjord et al., 2021) for deductive reasoning, and AR-LSAT (Zhong et al., 2021) for analytical reasoning. We propose PuzzleBench, which substantially extends the complexity of these benchmarks by investigating computationally hard, first-order

combinatorial reasoning problems.

3 Problem Setup

A first-order combinatorial reasoning problem $\mathcal{P}$ has three components: a space of legal input problem instances ($\mathcal{X}$), a space of legal outputs ($\mathcal{Y}$), and a set of constraints ($\mathcal{C}$) that every input-output pair must satisfy. Example, for sudoku, input space is the space of grids with $n \times n$ cells (n being a perfect square), with cells partially filled with numbers from 1 to n. Output space is a fully-filled grid of the same size. And, constraints comprise row, column, and box *alldiff* constraints, along with input cell persistence.

To communicate a structured problem instance (or its output) to an NLP system, it must be serialized in text. We overload $\mathcal{X}$ and $\mathcal{Y}$ to denote the *formats* for these serialized input and output instances. Two instances of serialized inputs/outputs for sudoku are shown in Figure 2 (grey box). We are also provided (serialized) training data of input-output instance pairs, $\mathcal{D}_{\mathcal{P}} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, where $x^{(i)} \in \mathcal{X}, y^{(i)} \in \mathcal{Y}$, such that $(x^{(i)}, y^{(i)})$ honors all constraints in $\mathcal{C}$.

Further, we verbalize all three components – input/output formats and constraints – in natural language instructions. We denote these instructions by $NL(\mathcal{X})$, $NL(\mathcal{Y})$, and $NL(\mathcal{C})$, respectively. Figure 2 illustrates these for our running example.

With this notation, we summarize our setup as follows. For an abstract *fcore* problem $\mathcal{P} = \langle \mathcal{X}, \mathcal{Y}, \mathcal{C} \rangle$, we are provided $NL(\mathcal{X})$, $NL(\mathcal{Y})$, $NL(\mathcal{C})$ and training data $\mathcal{D}_{\mathcal{P}}$, and our goal is to learn a function $\mathcal{F}$, which maps any (serialized) $x \in \mathcal{X}$ to its corresponding (serialized) solution $y \in \mathcal{Y}$ such that (x, y) honors all constraints in $\mathcal{C}$.

4 PuzzleBench: Dataset Construction

First, we shortlist computationally challenging problems. For that we manually scan Wikipedia for diverse puzzles and NP-hard algorithmic problems. We also find puzzles from other publishing houses (e.g., Nikoli).[1] From this set, we select problems (1) whose rules are easy to describe in words, (2) for whom it is easy to create a serialized input/output format, (3) for whom, the training and test datasets can be created with a reasonable programming effort, and (4) whose constraints have an SMT 2.0 encoding. This manual process led to identification of 31 *fcore* problems (see Table 5 for a complete list), of which 24 are known to be NP-hard and others have unknown complexity. 10 problems are graph-based (e.g., graph coloring), 15 are grid based (e.g., sudoku), and 5 are set-based (e.g., knapsack).

[1] https://www.nikoli.co.jp/en/puzzles/

Natural Language Description of Rules ($NL(\mathcal{C})$)

- Empty cells of the grid must be filled using numbers from 1 to n
- Each row must have each number from 1 to n exactly once
- Each column must have each number from 1 to n exactly once
- Each of the n non-overlapping sub-grids of size $\sqrt{n} \times \sqrt{n}$ must have each number from 1 to n exactly once
- n is a perfect square

Natural Language Description of Input Format ($NL(\mathcal{X})$)

- There are n rows and n columns representing the $n \times n$ unsolved grid
- Each row represents the corresponding unsolved row of the grid
- Each row has n space separated numbers ranging from 0 to n representing the corresponding cells in the grid
- Empty cells are indicated by 0s
- The other filled cells will have numbers from 1 to n

Natural Language Description of Output Format ($NL(\mathcal{Y})$)

- There are n rows and n columns representing the $n \times n$ solved grid
- Each row represents the corresponding solved row of the grid
- Each row has n space separated numbers ranging from 1 to n representing the corresponding cells of the solved grid

Solved Examples in their Textual Representation ($\mathcal{D}_{\mathcal{P}}$)

Input - 1	Output-1	Input-2	Output-2
0 3 1 2	4 3 1 2	0 3 1 2	4 3 1 2
1 0 4 3	1 2 4 3	2 0 0 4	2 1 3 4
2 1 0 4	2 1 4 3	3 0 0 1	3 4 2 1
3 4 2 0	3 4 2 1	0 0 4 0	1 2 4 3

Figure 2: PuzzleBench Example: Filling a $n \times n$ Sudoku board along with its rules, input-output format and a couple of sample input-output pairs.

We then create natural language instructions for each problem. We first develop input-output formats and create instructions for them ($NL(\mathcal{X})$, $NL(\mathcal{Y})$). Second, we rewrite all constraints/rules of the problem in our own words ($NL(\mathcal{C})$). This is done so that an LLM cannot easily invoke its prior knowledge about the same problem. For the same reason, we hide the name of the puzzle. Appendices A.2 and A.3 have detailed examples of rules and formats, respectively.

Next, we create train/test data for each problem. These instances are generated synthetically when possible or else hand-crafted by human experts. A single problem instance can potentially have multiple correct solutions (Nandwani et al., 2021) – all solutions are provided for each training input. The instances in the test set are typically larger in size than those in training. Because of their size, test instances may have too many solutions, and computing all of them can be expensive. Instead, we provide a verification script for each

problem, which outputs the correctness of a candidate solution for any test instance. We construct 15 training instances per *fcore* problem, and 20-50 test instances.

5 Puzzle-LM

5.1 Preliminaries and Background

Preliminaries: In the following, we assume that we have access to an LLM $\mathcal{L}$, which can work with various prompting strategies, a program interpreter $\mathcal{I}$, which can execute programs written in its programming language and a symbolic solver $\mathcal{S}$, which takes as input a pair of the form (E, V), where E is set of equations (constraints) specified in the language of $\mathcal{S}$, and V is a set of (free) variables in E, and produces an assignment $\mathcal{A}$ to the variables in V that satisfies the set of equations in E. Given the an *fcore* problem $\mathcal{P} = \langle \mathcal{X}, \mathcal{Y}, \mathcal{C} \rangle$ described by $NL(\mathcal{C})$, $NL(\mathcal{X})$, $NL(\mathcal{Y})$ and $\mathcal{D}_{\mathcal{P}}$, we would like to make effective use of $\mathcal{L}$, $\mathcal{I}$ and $\mathcal{S}$, to learn the $\mathcal{F}$, which takes any input $x \in \mathcal{X}$, and maps it to $y \in \mathcal{Y}$, such that (x, y) honors the constraints in $\mathcal{C}$.

Background: We consider the following possible representations for $\mathcal{F}$ which cover existing work.

- **Exclusively LLM**: Many prompting strategies (Wei et al., 2022c; Zhou et al., 2023; Zheng et al., 2023) make exclusive use of $\mathcal{L}$ (potentially multiple times) to represent $\mathcal{F}$, i.e., $\mathcal{L}$ is supplied with a prompt consisting of the description of $\mathcal{P}$ in the natural language, via $NL(\mathcal{C})$, $NL(\mathcal{X})$, $NL(\mathcal{Y})$, the input x, along with specific instructions on how to solve the problem and asked to output y directly. This puts the entire burden of discovering the mapping $\mathcal{F}$ on $\mathcal{L}$, ignoring $\mathcal{S}$ and $\mathcal{I}$.
- **LLM → Program**: In a variation of this strategy, the LLM is prompted to instead output a program which then is interpreted by $\mathcal{I}$ on the input x to produce the output y. This is the approach taken by methods such as PAL (Gao et al., 2023)
- **LLM + Solver**: This strategy makes use of both the LLM $\mathcal{L}$ and the symbolic solver $\mathcal{S}$. Whereas the primary goal of $\mathcal{L}$ is to to act as interface for translating the problem description for $\mathcal{P}$ and the input x, to the language of the solver $\mathcal{S}$, and the primary burden of solving the problem is on $\mathcal{S}$, whose output is then parsed as y. This is the approach taken by works such as Logic-LM (Pan et al., 2023) and Sat-LM (Ye et al., 2023).

5.2 Our Approach

Our approach can be seen as a combination of LLM→Program and LLM+Solver strategies described above. While the primary role of the LLM is to do the interfacing between the natural language description of the problem $\mathcal{P}$, the task of solving the actual problem is delegated to the solver $\mathcal{S}$ as in LLM+Solver strategy. But unlike them, where the LLM directly calls the solver, we now prompt it to write an input agnostic program, ψ, which can work with any given input $x \in \mathcal{X}$. This allows us to get rid of the LLM calls at inference time, resulting in a "lifted" implementation. The program ψ internally represents the specification of the problem. It takes as argument an input x, and then converts it according to the inferred specification of the problem to a set of equations (E_x, V_x) in the language of the solver $\mathcal{S}$ to get the solution to the original problem. The solver S then ouptuts an assignment A_x in its own representation, which is then passed back to the program ψ which converts it back to the desired output format specified by $\mathcal{Y}$ and produces output $\hat{y}$. Broadly, our pipeline consists of the 3 components which we describe next in detail.

- **Prompting LLMs**: The LLM is prompted with $NL(\mathcal{C})$, $NL(\mathcal{X})$, $NL(\mathcal{Y})$ to generate an input-agnostic Python program ψ. The LLM is instructed to write ψ to read an input from a file, convert it to a symbolic representation according to the inferred specification of the problem, pass the symbolic representation to the solver and then use the solution from the solver to generate the output in the desired format and write it to another file. The LLM is also prompted with information about the solver and its the underlying language in which the symbolic representation is to be generated. Optionally we can also provide the LLM with a subset of $\mathcal{D}_{\mathcal{P}}$.
- **Symbolic Solver**: ψ can convert any input instance x to (E_x, V_x) which it passes to the symbolic solver. The solver is agnostic to how the representation (E_x, V_x) was created and tries to find an assignment A_x to V_x which satisfies E_x which is passed back to ψ.
- **Generating the Final Output**: ψ then uses $\mathcal{A}_x$ to generate the predicted output $\hat{y}$. This step is need because the symbolic representation was created by ψ and it must recover the desired output representation from $\mathcal{A}_x$ which might not be straightforward for all problem representations.

5.3 Verification from Solved Examples

We make use of $\mathcal{D}_{\mathcal{P}}$ to verify and correct (if needed) ψ. For each $(x, y) \in \mathcal{D}_{\mathcal{P}}$ (solved input-output pair), we run ψ on x to generate the prediction $\hat{y}$, during which the following can happen: 1) Errors during execution of ψ; 2) The solver is unable to find $\mathcal{A}_x$ under a certain time limit; 3) $\hat{y} \neq y$ i.e. the predicted output is incorrect; 4) $\hat{y} = y$ i.e. the predicted output is correct. If for any training input one of the first three cases occur we provide feedback to the LLM through prompts to improve and generate a new program. This process is repeated till all training examples are solved correctly or till a maximum number of feedback rounds is reached. The feedback is simple in nature and includes the nature of the error, the actual error from the interpreter/symbolic solver and the input instance on which the error was generated. For example, in the case where the output doesn't match the gold output we prompt the LLM with the solved example it got wrong and the expected solution. In the case where the solver is not able to solve for the constraints in the time limit, we prompt the LLM to optimize the programs. Appendix B contains details of feedback prompts.

It is possible that a single run of `Puzzle-LM` does not generate the correct solution for all training examples. `Puzzle-LM` is run multiple times for a given problem. Given the probabilistic nature of LLMs a new program is generated each time and a new feedback process continues. For the final program, we pick the best program generated during these runs, as judged by the accuracy on the training set. Figure 3 describes our entire approach diagrammatically.

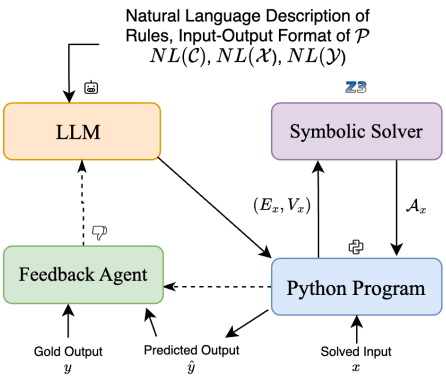


Figure 3: `Puzzle-LM`: Solid lines indicate the main flow and dotted lines indicate feedback pathways.

6 Experimental Setup

Our experiments answer these research questions:

1. How well do LLMs perform on `PuzzleBench`?
2. How does `Puzzle-LM` compare with other tool-augmented LLMs on *fcore* problems?
3. How useful is using feedback from solved examples and multiple runs for *fcore* problems?

Baselines: We compare our method with 3 baselines: 1) *Standard LLM prompting*, which leverages in-context learning to directly answer the questions; 2) *Program-aided Language Models*, which use imperative programs for reasoning and offload the solution step to a program interpreter; 3) *Logic-LM*, which offloads the reasoning to a symbolic solver. We use Z3 (De Moura and Bjørner, 2008) an efficient SMT solver for experiments with Logic-LM and `Puzzle-LM`. We use the Python interpreter for experiments with PAL and `Puzzle-LM`. We also evaluate *refinement* for PAL and `Puzzle-LM` by using 4 rounds of feedback on solved examples and 5 runs for each problem. We evaluate *refinement* for Logic-LM by providing 4 rounds of feedback to correct syntactic errors in constraints (if any) for each problem instance.

Models: For each method, we experiment with 3 LLMs: GPT-4-Turbo (`gpt-4-0125-preview`) (OpenAI, 2023) which is a SOTA LLM by OpenAI, GPT-3.5-Turbo (`gpt-3.5-turbo-0125`), a relatively smaller LLM by OpenAI and Mixtral 8x7B (`mistral-small`) (Jiang et al., 2024), an open-source mixture-of-experts model developed by Mistral AI. For experiments with few-shot prompting and Logic-LM, we set the temperature to 0 for reproducible results. For experiments with PAL and `Puzzle-LM` we use a temperature of 0.7 to sample several runs, each with a potentially different feedback cycle.

Prompting LLMs: For each method the prompt contains the natural language description of the rules of the problem and the input-output format. It also contains two solved examples – pairs of input and one its corresponding outputs – in the prompt. No additional intermediate supervision (e.g., SMT or Python program) is given in the prompt. For few-shot prompting we directly prompt the LLM to solve test set instances (each tested separately). For PAL we prompt the LLM to write an input-agnostic Python program which reads the input from a file, reasons to solve the input and then writes the solution to another file, the program generated is run on

Model	Random Guessing	Few-Shot Prompting	PAL		Logic-LM		Puzzle-LM	
			-	+	-	+	-	+
Mixtral 8x7B		23.6%	16.1%	37.92%	0.0%	1.61%	10.77%	36.72%
GPT-3.5-Turbo	17.91%	25.4%	35.82%	53.4%	6.87%	7.41%	30.74%	56.86%
GPT-4-Turbo		30.5%	48.51%	66.45%	37.34%	39.91%	60.48%	**93.55%**

Table 1: Main Results for PuzzleBench and Puzzle-LM. - and + indicate results before and after *refinement*.

each testing set instance. For Logic-LM for each test set instance we prompt the LLM to convert it into its symbolic representation which is then fed to a symbolic solver, the prompt additionally contains the description of the language of the solver. We then prompt the LLM with the solution from the solver and ask it to generate the output in the desired format. Details for Puzzle-LM can be found in Section 5.1. Prompt templates can be found in Appendix B and other experimental details can be found in Appendix C.

Metrics: For each problem we use the verification script to check if the candidate solution generated for each test set instance is correct. We use this script to compute the accuracy, i.e. fraction of test set instances solved correctly for each problem using binary marking i.e. 0 for incorrect solution and 1 for a correct solution. It is possible that no solution is generated or the solution generated is not in the output-format specified, we consider these to be incorrect and assign a score of 0. We use the macro-average of test set accuracies of all problems in PuzzleBench as our metric.

7 Results

Table 1 describes the main results for Puzzle-LM and other baselines on PuzzleBench. GPT-4-Turbo is significantly better than other models across all methods. Mixtral 8x7B struggles on our benchmark indicating that smaller LLMs (even with mixture of experts) are not as effective at complex reasoning tasks. Mixtral's performance with few-shot prompting is slightly better than random guessing and with other methods (without *refinement*) is even worse than random guessing. PAL and Puzzle-LM tend to perform better than other baselines benefiting from the vast pre-training of LLMs on code (Brown et al., 2020; Chen et al., 2021). Logic-LM performs rather poorly with smaller LLMs indicating that they struggle to invoke symbolic solvers directly. Hereafter, we focus on just GPT-4-Turbo's performance, since it is far superior to other models.

7.1 Puzzle-LM vs Baselines

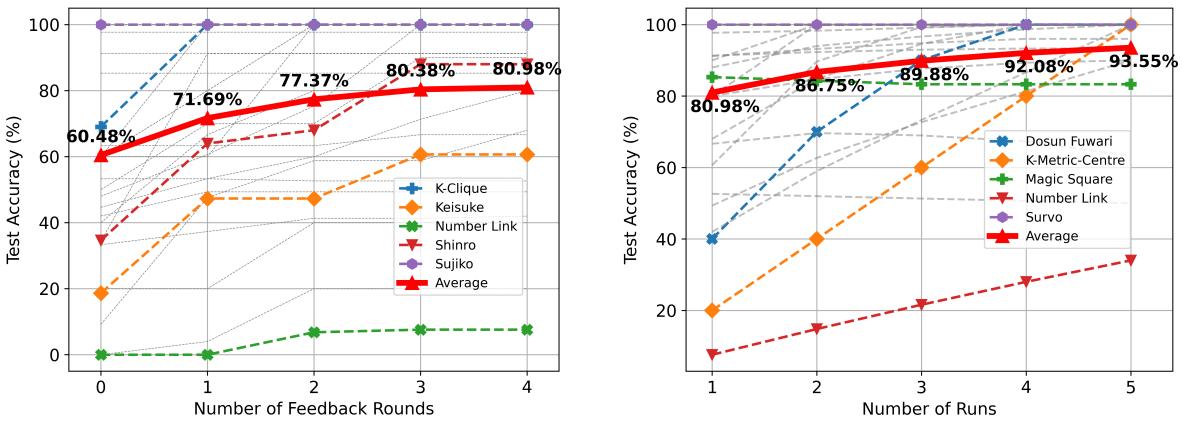
Puzzle-LM outperforms few-shot prompting and Logic-LM across all problems in PuzzleBench. On average with GPT-4-Turbo the improvements are by an impressive 63.05% against few-shot prompting and by 53.64% against Logic-LM on PuzzleBench (with *refinement*). Few-shot prompting struggles, solving less than a third of the problems even with GPT-4-Turbo indicating the need for augmenting LLMs with tools for complicated reasoning. While Logic-LM performs better, it still struggles indicating that combining LLMs with symbolic solvers is not enough for structured reasoning problems.

Our results indicate that few-shot prompting and Logic-LM are unable to leverage the structure of *fcore* problems. As problem instance size grows, Logic-LM tends to make syntactic mistakes in the generated constraints with smaller LLMs (Table 2). With larger LLMs syntax mistakes in the constraints reduce, but they still remain semantically incorrect and could not be corrected through feedback. In contrast, Puzzle-LM exploits the first order structure of these problems by generating instance-agnostic programs during training that can then be corrected through feedback from solved examples. They can then be used independently at inference to solve any input instance, highlighting the importance of using program interpreters along with symbolic solvers.

Outcome	GPT-3.5-Turbo	GPT-4-Turbo
Correct Output	7.41%	39.91%
Incorrect Output	46.32%	37.90%
Inefficient Solution	2.90%	2.90%
Syntactic Errors	25.62%	6.36%
Errors in Converting Symbolic Output to Text	17.73%	12.93%

Table 2: Logic-LM's performance on PuzzleBench evaluated with feedback to fix syntactic mistakes.

Table 3 compares the effect of feedback and multiple runs on PAL and Puzzle-LM. Puzzle-LM outperforms PAL by 27.1% on PuzzleBench (with



(a) Effect of Feedback

(b) Effect of Mutliple Runs

Figure 4: Dashed lines show results for individual problems in PuzzleBench, with coloured lines highlighting specific cases and grey lines others. The red bold represents the average effect across all problems with GPT-4-Turbo.

refinement). This highlights the benefits of offloading the reasoning to a symbolic solver. When LLMs are forced to perform complicated reasoning while writing programs, they tend to produce brute-force solutions which are either often incorrect or extremely slow (see Table-6 in the appendix). Puzzle-LM tends to be much more efficient in terms of inference-time given that the solver can exploit the nature of the input instance while performing the reasoning unlike brute-force programs. Interestingly, feedback from solved examples and re-runs are more effective in improving performance for Puzzle-LM as indicated by the larger gains with increasing number of feedback rounds and runs. We attribute this to the fact that the programs generated by Puzzle-LM are declarative in nature, while PAL produces imperative programs which are difficult to correct.

7.2 Effect of Feedback on Solved Examples

Figure 4 describes the effect of multiple rounds of feedback for Puzzle-LM. Feedback helps performance significantly; utilizing 4 rounds of feedback improves reasoning abilities by 25.36%. Even the largest LLMs commit errors, making it important to verify and correct their work. But feedback on its own is not enough, a single run might end-up in a wrong reasoning path, which is not corrected by feedback making it important to utilize multiple runs for effective reasoning. Utilizing 5 runs improves the performance by additional 12.32% after which the gains tend to saturate. Performance also increases with an increase in the number of solved examples (Table 4). Each solved example helps in detecting and correcting different errors. However, performance saturates at 7 solved examples and no new errors are discovered, even with additional training data.

	Number of Rounds of Feedback				
	0	**1**	**2**	**3**	**4**
PAL	48.51%	54.52%	57.83%	58.98%	59.80%
Puzzle-LM	60.48%	71.69%	77.37%	80.38%	80.98%
	↑ 11.97 %	↑ 17.17 %	↑ 19.99 %	↑ 21.4 %	↑ 21.18 %

(a) Effect of number of rounds of feedback for a single run

	Number of Runs				
	1	**2**	**3**	**4**	**5**
PAL	59.80%	62.45%	63.93%	65.23%	66.45&
Puzzle-LM	80.98%	86.75%	89.88%	92.08%	93.55%
	↑ 21.18 %	↑ 24.3 %	↑ 25.95 %	↑ 26.85 %	↑ 27.1 %

(b) Effect of number of runs each having 4 rounds of feedback

Table 3: Comparative analysis between PAL and Puzzle-LM on PuzzleBench for GPT-4-Turbo

	Number of Feedback Rounds				
	0	**1**	**2**	**3**	**4**
N = 0	**60.48%**	60.48%	60.48%	60.48%	60.48%
N = 1	60.48%	69.47%	73.99%	75.73%	75.75%
N = 4	60.48%	71.43%	76.87%	79.52%	79.54%
N = 7	60.48%	**71.69%**	**77.37%**	**80.38%**	**80.98%**
N = 10	60.48%	71.69%	77.37%	80.38%	80.98%

Table 4: Effect of number of solved examples (N) for feedback and number of feedback rounds for Puzzle-LM with GPT-4-Turbo on PuzzleBench for a single run.

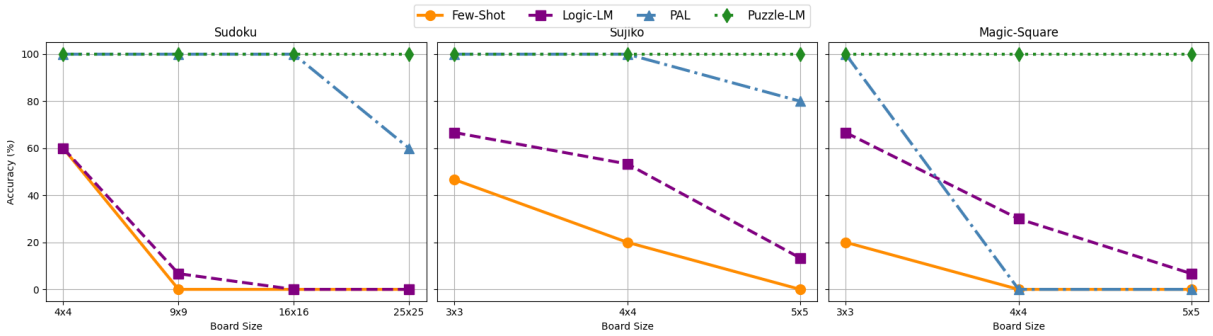


Figure 5: Effect of increasing problem instance size on baselines and Puzzle-LM for GPT-4-Turbo.

7.3 Effect of Problem Instance Size

We compare performance of Puzzle-LM and other baselines against varying problem instance sizes (see Figure 5) for 3 problems in PuzzleBench (sudoku, sujiko and magic-square). Increasing the problem instance size increases the number of variables, accompanying constraints and reasoning steps required to reach the solution. We observe that the performance of Puzzle-LM and PAL, which are programmatic approaches, is relatively robust against increase in size of input instances. In comparison, performance of Logic-LM and few-shot prompting declines sharply. Performance of PAL declines in some cases because of inefficient programs generated, which are unable to find a solution in the time limit.

8 Qualitative Analysis

We now discuss our insights on where Puzzle-LM excels and where the largest LLMs still struggle. Qualitatively, the problems in PuzzleBench can be broadly categorized into three categories:

1) Problems that our solved with 100% accuracy without any feedback from solved examples. 8 such problems exist out of the 31, including vertex-cover and latin-square. These problems have a one-to-one correspondence between the natural language description of the rules and the program for generating the constraints and the LLM essentially has to perform a pure translation task.

2) Problems that our solved with 100% accuracy but which need feedback from solved examples. There are 17 such problems. They typically do not have a one-to-one correspondence between rule descriptions and code, thus requiring some reasoning to encode the problem in the solver's language. GPT-4-Turbo initially misses constraints or encodes the problem incorrectly, but with feedback, it can spot its mistakes and corrects its programs. Examples include k-clique and binairo. In binairo, for example, GPT-4-Turbo incorrectly encodes the constraints for ensuring all columns and rows are distinct. In k-clique, it interprets the problem incorrectly and adds an extra constraint to ensure that if a vertex is part of a clique, then all its neighbours are also a part of the clique.

3) Problems with performance below 100% which are not corrected through feedback or utilizing multiple runs. For these 6 problems, LLM finds it difficult to encode some natural language constraint into SMT. Examples include number-link and hamiltonian path, where GPT-4-Turbo is not able to figure out how to encode existence of paths as SMT constraints. In our opinion, these conversions are peculiar, and may be hard even for average computer science students.

9 Conclusion and Future Work

We explore the reasoning abilities of LLMs on challenging first-order combinatorial reasoning problems. We present PuzzleBench, a novel benchmark of 31 such problems and find that existing tool-augmented techniques fair poorly. In response, we present Puzzle-LM a new technique to aid LLMs with both program interpreters and symbolic solvers. Our extensive experiments show Puzzle-LM's superior performance on our dataset. We also find that solved examples provide useful feedback to improve LLM reasoning. Further analyis reveals that Puzzle-LM struggles for a certain class of problems where conversion to symbolic representation is not straightforward. In such cases simple feedback strategies do not improve reasoning; exploring methods to alleviate such problems is a promising direction for future work. We release our code and data for further research.

10 Limitations

Dataset contamination is a big problem in the era of pretrained language models which have been trained on large web corpora. Therefore, its really hard to determine if a dataset has been seen as part of the training data. Determining the extent of contamination is also not easy. Since PuzzleBench does contain some standard problems from Computer Science literature, there is possibility of contamination. Usage of LLMs for reasoning on text is also limited by availability of large compute resources.

Our approach is also limited by the nature of the symbolic solver used. Some problems cannot be encoded in the symbolic language used by the solver, although our approach remains flexible enough to include several such solvers. Even in the case that a problem can be encoded in the symbolic language used by the solver, this encoding can potentially be complicated. Our method relies on the in-context learning ability of LLMs to achieve this. Additionally we assume that input instances and their outputs have a fixed pre-defined (serialized) representation.

11 Ethics Statement

(Strubell et al., 2019) show that the financial and environmental costs that are associated with the training of LLMs lead to significant contribution to global warming. We don't train the model from scratch and leverage in context learning, so the energy footprint of our work is less. The large language models whose API we use for inference, especially GPT-4, consume significant energy.

References

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Peter Clark, Oyvind Tafjord, and Kyle Richardson. 2021. Transformers as soft reasoners over language. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, IJCAI'20.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168.

Charles J. Colbourn. 1984. The complexity of completing partial latin squares. *Discrete Applied Mathematics*, 8(1):25–30.

Marzio De Biasi. 2013. Binary puzzle is np–complete.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.

D. Epstein. 1987. On the np-completeness of cryptarithms. *ACM SIGACT News*, 18(3):38–40.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Ian P. Gent, Christopher Jefferson, and Peter Nightingale. 2017. Complexity of n-queens completion. *Journal of Artificial Intelligence Research (JAIR)*, 58:1–16.

Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Luke Benson, Lucy Sun, Ekaterina Zubova, Yujie Qiao, Matthew Burtell, David Peng, Jonathan Fan, Yixin Liu, Brian Wong, Malcolm Sailor, Ansong Ni, Linyong Nan, Jungo Kasai, Tao Yu, Rui Zhang, Shafiq R. Joty, Alexander R. Fabbri, Wojciech Kryscinski, Xi Victoria Lin, Caiming Xiong, and Dragomir Radev. 2022. FOLIO: natural language reasoning with first-order logic. *CoRR*, abs/2209.00840.

Kazuya Haraguchi and Hirotaka Ono. 2015. How simple algorithms can solve latin square completion-type puzzles approximately. *Journal of Information Processing*, 23(3):276–283.

Chuzo Iwamoto and Tatsuaki Ibusuki. 2018. Dosunfuwari is np-complete. *Journal of Information Processing*, 26:358–361.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of experts.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Jian Liu, Leyang Cui, Hanmeng Liu, Dandan Huang, Yile Wang, and Yue Zhang. 2020. Logiqa: A challenge dataset for machine reading comprehension with logical reasoning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 3622–3628. ijcai.org.

Huw Lloyd, Matthew Crossley, Mark Sinclair, and Martyn Amos. 2022. J-pop: Japanese puzzles as optimization problems. *IEEE Transactions on Games*, 14(3):391–402.

Yatin Nandwani, Vidit Jain, Mausam, and Parag Singla. 2022a. Neural models for output-space invariance in combinatorial problems. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Yatin Nandwani, Deepanshu Jindal, Mausam, and Parag Singla. 2021. Neural learning of one-of-many solutions for combinatorial problems in structured output spaces. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Yatin Nandwani, Rishabh Ranjan, Mausam, and Parag Singla. 2022b. A solver-free framework for scalable learning in neural ILP architectures. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

OpenAI. 2023. Gpt-4 technical report.

Rasmus Berg Palm, Ulrich Paquet, and Ole Winther. 2018. Recurrent relational networks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 3372–3382.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. 2023. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 3806–3824. Association for Computational Linguistics.

Bhargavi Paranjape, Scott M. Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Túlio Ribeiro. 2023. ART: automatic multi-step reasoning and tool-use for large language models. *CoRR*, abs/2303.09014.

Anselm Paulus, Michal Rolínek, Vít Musil, Brandon Amos, and Georg Martius. 2021. Comboptnet: Fit the right np-hard problem by learning integer programming constraints. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 8443–8453. PMLR.

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

Stuart Russell and Peter Norvig. 2010. *Artificial Intelligence: A Modern Approach*, 3 edition. Prentice Hall.

Abulhair Saparov and He He. 2023. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *CoRR*, abs/2302.04761.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 3645–3650. Association for Computational Linguistics.

Oyvind Tafjord, Bhavana Dalvi, and Peter Clark. 2021. Proofwriter: Generating implications, proofs, and abductive statements over natural language. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021*, volume ACL/IJCNLP 2021 of *Findings of ACL*, pages 3621–3634. Association for Computational Linguistics.

Po-Wei Wang, Priya Donti, Bryan Wilder, and Zico Kolter. 2019. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6545–6554. PMLR.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022a. Emergent abilities of large language models. *Trans. Mach. Learn. Res.*, 2022.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022b. Emergent abilities of large language models. *Trans. Mach. Learn. Res.*, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022c. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Kaiyu Yang, Jia Deng, and Danqi Chen. 2022. Generating natural language proofs with verifier-guided search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 89–105, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *CoRR*, abs/2305.10601.

Takayuki YATO and Takahiro SETA. 2003. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2023. Satlm: Satisfiability-aided language models using declarative prompting. In *Proceedings of NeurIPS*.

Dongran Yu, Bo Yang, Dayou Liu, Hui Wang, and Shirui Pan. 2023. A survey on neural-symbolic learning systems. *Neural Networks*, 166:105–126.

Weihao Yu, Zihang Jiang, Yanfei Dong, and Jiashi Feng. 2020. Reclor: A reading comprehension dataset requiring logical reasoning. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Li Zhang and Chris Callison-Burch. 2023. Language models are drummers: Drum composition with natural language pre-training. *CoRR*, abs/2301.01162.

Chuanyang Zheng, Zhengying Liu, Enze Xie, Zhenguo Li, and Yu Li. 2023. Progressive-hint prompting improves reasoning in large language models. *CoRR*, abs/2304.09797.

Wanjun Zhong, Siyuan Wang, Duyu Tang, Zenan Xu, Daya Guo, Jiahai Wang, Jian Yin, Ming Zhou, and Nan Duan. 2021. AR-LSAT: investigating analytical reasoning of text. *CoRR*, abs/2104.06598.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. 2023. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

A PuzzleBench

A.1 Dataset Details and Statistics

Our dataset namely PuzzleBench has 31 different *fcore* problems that have been collected from various sources. Some of these problems have been collected from Nikoli and other problems are well-known computational problems from Computer Science literature such as hamiltonian path and minimum-dominating set. Table 5 gives the details of all problems in our dataset. To create our training and test sets, we synthetically generate problem instances when possible and we handcraft problem instances otherwise. Each problem has some solved training instances and a separate set of testing instances. Each problem also has a natural language description of its rules, and a natural language description of the input-format which specify how input problem instances and their solutions are represented in text. The next few sections give illustrative examples and other details.

A.2 Natural Language Description of Rules

This section describes how we create the natural language description of rules for problems in PuzzleBench. We extract rules from Wikipedia/Nikoli pages of the corresponding problems. These rules are reworded by a human expert to reduce dataset contamination. Another human expert ensures that there are no ambiguities in the reworded description of the rules. The rules are generalized, when needed (for eg. from a 9×9 Sudoku to a $n \times n$ Sudoku). The following sections provide few examples.

A.2.1 Example Problem: Survo

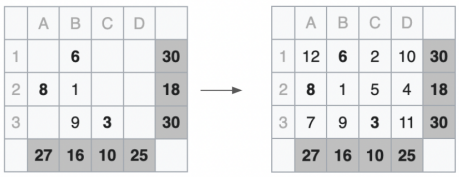


Figure 6: Conversion of an input survo problem instance to its solution.

Survo (Figure 6) is an example problem from PuzzleBench. The task is to fill a $m \times n$ rectangular board with numbers from $1 - m * n$ such that each row and column sums to an intended target. (Survo-Wikipedia). The box given below describes the rules of Survo more formally in natural language.

```
We are given a partially filled m × n rectangular
board, intended row sums and column sums.
- Empty cells are to be filled with numbers
- Numbers in the solved board can range from 1 to
m * n
- Numbers present in filled cells on the input
board cannot be removed
- Each number from 1 to m*n must appear exactly
once on the solved board
- All the empty cells should be filled such that
each row and each column of the solved board must
sum to the respective row sum and column sum as
specified in the input
```

A.2.2 Example Problem: Hamiltonian Path

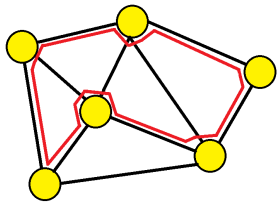


Figure 7: A sample input graph instance and its solution to the hamiltonian-path problem. Vertices are represented by yellow circles and the hamiltonian path is represented by the red line.

Hamiltonian path is a well-known problem in graph theory in which we have to find a path in an un-directed and an un-weighted graph such that each vertex is visited exactly once by the path. We consider the decision variant of this problem which is equally hard in terms of computational complexity. The box below shows the formal rules for this problem expressed in natural language.

```
We are given an un-directed and un-weighted graph.
- We have to determine if the graph contains a path
that visits every vertex exactly once.
```

A.2.3 Example Problem: Dosun Fuwari

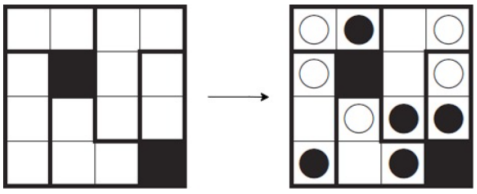


Figure 8: Conversion of an input dosun fuwari problem instance to its solution.

Dosun Fuwari (Nikoli) as shown in Figure 8 is another example problem from PuzzleBench. We are given a square board with regions (cells enclosed in bold lines) and we have to fill the board

with balloons and iron balls such that one balloon and one iron ball is placed in each region. Balloons are light and float, so they must be placed in one of the cells at the top, in a cell right under a black cell (filled-in cell), or under other balloons. Iron balls are heavy and sink, so they must be placed in one of the cells at the bottom, or in a cell right over a black cell or over other iron balls. The box given below gives the more formal description of the rules of dosun fuwari in natural language.

```
We are given a partially filled n*n square board.
We are also given subgrids of the input board.
Cells in the input board can either be empty or
filled (that is, nothing can be placed in them,
they are blackened) or can be balloons or iron
balls.
- The only thing we can do is place balloons or
iron balls in some of or all of the empty cells
- Each subgrid specified in the input should have
exactly one balloon and iron ball in the solved
board
- Because balloons are buoyant, they should be
positioned either in one of the cells located at
the top of the board or in a cell directly below a
filled cell (i.e., one of the blackened cells in
the input) or below other balloons.
- Iron balls, being dense, will sink and should
therefore be positioned either directly on one of
the cells located at the bottom of the input board,
or on a cell directly above a filled cell (i.e.,
one of the blackened cells in the input), or above
another iron ball.
```

A.3 Natural Language Description of Input and Output format

For many problems we consider input-output instances are typically not represented in text. For each problem we describe a straightforward conversion of the input and output space to text in natural language. The following sections consider examples of a few problems from PuzzleBench.

A.3.1 Example Problem: Survo

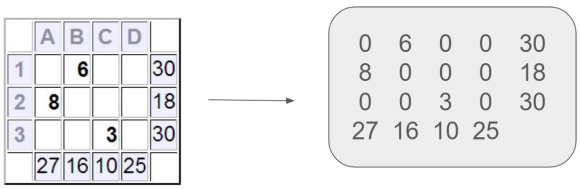


Figure 9: Representation of input instances of survo as text.

Figure 9 represents the conversion of the inputs to survo, originally represented as grid images to text. Here empty cells are denoted by 0's and the filled cells have corresponding values. For a given $m \times n$ board, each row has $m + 1$ space separated integers with the first m integers representing the first row of the input board and the $(m{+}1)^{th}$ integer representing the row sum. The last row contains n integers represent the column sums. The box below describes this conversion more formally in natural language.

```
Input Format:
- The input will have m + 1 lines
- The first m lines will have n+1 space-separated
integers
- Each of these m lines represents one row of
the partially solved input board (n integers),
followed by the required row sum (a single
integer)
- The last line of the input will have n
space-separated integers each of which represents
the required column sum in the solved board
Sample Input:
0 6 0 0 0 30
8 1 0 0 0 17
0 9 3 0 30
27 16 10 25

Output Format:
- The output should have m lines, each representing
one row of the solved board
- Each of these m lines should have n
space-separated integers representing the cells of
the solved board
- Each integer should be from 1 to m * n
Sample Output:
12 6 2 10
8 1 5 4
7 9 3 11
```

A.3.2 Example Problem: Dosun Fuwari

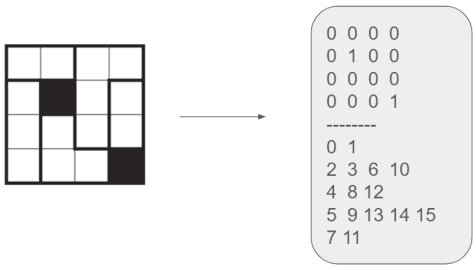


Figure 10: Representation of inputs instances to dosun-fuwari as text.

Figure 10 represents conversion of the inputs to dosun fuwari, originally represented as grid images to text. Here the first few lines represent the input board followed by a string '——' which acts as a separator following which each of the lines has space-separated integers representing the subgrids of the input board. Cells are numbered in row-major order starting from 0, and this numbering is

used to represent cells in each of the lines describing the subgrids. In the first few lines representing the input board, 0's represent the empty cells that must be filled. 1's denote the blackened cell, 2s denote the balloons and 3's denote the iron balls. The box below describes these rules more formally in natural language

```
Input-Format:
- The first few lines represent the input board, followed by a line containing ---, which acts as a separator, followed by several lines where each line represents one subgrid
- Each of the lines representing the input board will have space-separated integers ranging from 0 to 3
- 0 denotes empty cells, 1 denotes a filled cell (blackened cell), 2 denotes a cell with a balloon, 3 denotes a cell with an iron ball
- After the board, there is a separator line containing ---
- Each of the following lines has space-separated elements representing the subgrids on the input board
- Each of these lines has integers representing cells of a subgrid
- Cells are numbered in row-major order starting from 0, and this numbering is used to represent cells in each of the lines describing the subgrids

Sample-Input:
0 0 0 0
0 1 0 0
0 0 0 0
0 0 0 1
---
0 1
2 3 6 10
4 8 12
5 9 13 14 15
7 11

Output Format:
- The output should contain as many lines as the size of the input board, each representing one row of the solved board
- Each row should have n space separate integers (ranging from 0-3) where n is the size of the input board
- Empty cells will be denoted by 0s, filled cells (blackened) by 1s, balloons by 2s and iron balls by 3s

Sample-Output:
2 3 0 2
2 1 0 2
0 2 3 3
3 0 3 1
```

A.3.3 Example Problem: Hamiltonian Path

Figure 11 represents the conversion of inputs to hamiltonian-path, originally represented as graph image to text. The first line denotes the number of vertices present in the graph followed by which each node of the graph will be numbered from 0 - N-1. Each of the subsequent lines represents an edge of the graph and will contain two space-separated integers (according to the numbering defined previously). The output is a single word (YES/NO) indicating if a hamiltonian path exists in the graph. The box below describes this more formally in natural language.

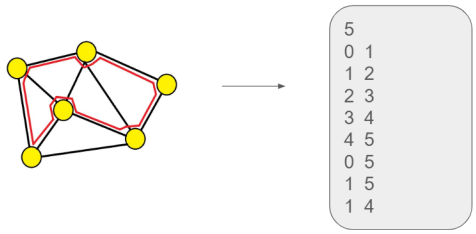


Figure 11: Representation of input instances to hamiltonian-path as text.

```
Input Format:
- The first line will contain a single integer N, the number of nodes in the graph
- The nodes of the graph will be numbered from 0 to N-1
- Each of the subsequent lines will represent an edge of the graph and will contain two space-separated integers (according to the numbering defined above)

Sample-Input:
5
0 1
1 2
2 3
3 4

Output Format:
- The output should contain a single line with a single word
- The word should be YES if a path exists in the input graph according to constraints specified above and NO otherwise

Sample Output:
YES
```

Problem Name	Training Set Size	Test Set Size	Average Size of Input Instances in Training Set	Average Size of Input Instances in Test Set	Computational Complexity
3-Partition (Non Decision)	15	30	12 (array size)	17.7	NP-Hard
3-Partition (Decision)	15	30	12 (array size)	17.7	NP-Complete
Binario	15	50	4.0×4.0 (grid size)	6.96×6.96	NP-Hard (De Biasi, 2013)
Clique Cover	15	30	6.26, 9.4 (# of nodes, # of edges)	12.9, 31.4	NP-Complete
Cryptarithmetic	15	25	4.32 (Average # of digits in the two operands)	3.26	NP-Hard (Epstein, 1987)
Dosun Fuwari	15	30	3.066×3.066 (grid size)	5.23×5.23	NP-Hard (Iwamoto and Ibusuki, 2018)
Futoshiki	15	47	5×5 (grid size)	7.57×7.57	NP-Hard (Lloyd et al., 2022)
Graph Coloring	15	30	5.13, 6.8 (# of nodes, # of edges)	9, 21.06	NP-Complete (Gent et al., 2017)
Hamiltonian Path	15	30	5.93, 8.6 (# of nodes, # of edges)	13.0, 19.77	NP-Complete
Hamiltonian Cycle	15	27	5.93, 8.6 (# of nodes, # of edges)	11.07, 18.67	NP-Complete
Independent Set	12	30	5.83, 7.25 (# of nodes, # of edges)	14.26, 29.8	NP-Complete
Inshi-No-Heya	15	49	5.0×5.0 (grid size)	6.5×6.5	?
K-Clique	15	31	4.87, 7.67 (# of nodes, # of edges)	8.84, 26.97	NP-Complete
Keisuke	15	30	4.33×4.33 (grid size)	5.83×5.83	?
Ken Ken	15	20	3.26×3.26 (grid size)	5.2×5.2	NP-Hard (Haraguchi and Ono, 2015)
Latin Square	15	50	6.0×6.0 (grid size)	14.3×14.3	NP-Hard (Colbourn, 1984)
Longest Path Problem	15	30	6.2, 5.87 (# of nodes, # of edges)	12.6, 16.3	NP-Complete
Magic Square	15	30	3.0×3.0 (grid size)	4.33×4.33	?
Minimum Dominating Set	15	30	6.0, 17.73 (# of nodes, # of edges)	14.53, 45.0	NP-Complete
Number Link	15	50	4×4 (grid size)	7.1×7.1	NP-Hard
Partition Problem	15	35	7.06 (array size)	15	NP-Complete
Shinro	15	25	5.13×5.13 (grid size)	9.2×9.2	?
Subset Sum	15	30	3.67 (array size)	11.87	NP-Complete
Sudoku	15	50	4.0×4.0 (grid size)	13.3×13.3	NP-Hard (YATO and SETA, 2003)
Sujiko	15	45	3.0×3.0 (grid size)	4.0×4.0	?
Survo	15	47	13.5 (area of grid)	20.25	?
Symmetric Sudoku	15	30	4×4 (grid size)	6.5×6.5	?
Vertex Cover	14	30	6.42,1 3.42 (# of nodes, # of edges)	12.6, 40.47	NP-Complete
N-Queens	15	30	3.8×3.8 (grid size)	6.33×6.33	NP-Hard (Gent et al., 2017)
Knapsack	15	30	4.8 (array size)	24.56	NP-Hard
K Metric Centre	15	30	4.5 (# of nodes)	7	NP-Hard

Table 5: Names of problems in the PuzzleBench, number of samples in the training set, number of samples in the test set, average size of input instances in training set, average size of input instances in test set and computational complexity. The brackets in the 4th column describe how input instance sizes are measured. ? in the computational complexity column indicates that results are not available for the corresponding problem.

B Prompt Templates

In this section we provide prompt templates used for our experiments on PuzzleBench, including the templates for the baselines we experimented with, Puzzle-LM as well as prompt templates for providing feedback.

B.1 Few-Shot Prompt Template

```
Task:
<Description of the Rules of the problems>

Input-Format:
<Description of Textual Representation of Inputs>
<Input Few Shot Example-1>
<Input Few Shot Example-2>
.......................
.......................
<Input Few Shot Example-n>

Output-Format
<Description of Textual Representation of Outputs>

<Output of Few Shot Example-1>
<Output of Few Shot Example-2>
...........................
...........................
<Output of Few Shot Example-n>

Input problem instance to be solved:
<Puzzle from the Test Set>
```

B.2 PAL Prompt Template

The following box describes the base prompt template used for PAL experiments with PuzzleBench.

```
Write a Python program to solve the following
problem:

Task:
<Description of the Rules of the problem>

Input-Format:
<Description of Textual Representation of Inputs>
Sample-Input:
<Sample Input from Feedback Set>

Output-Format:
<Description of Textual Representation of Outputs>
Sample-Output:
<Output of Sample Input from Feedback Set>

Don't write anything apart from the Python program;
use Python comments if needed.

The Python program is expected to read the input
from input.txt and write the output to a file
named output.txt.

The Python program must only use standard Python
libraries.
```

B.3 Puzzle-LM Template

B.3.1 Base Prompt

```
Write a Python program to solve the following
problem:

Task:
<Description of the Rules of the problem>

Input-Format:
<Description of Textual Representation of Inputs>
Sample-Input:
<Sample Input from Feedback Set>

Output-Format:
<Description of Textual Representation of Outputs>
Sample-Output:
<Output of Sample Input from Feedback Set>

The Python program must read the input from
input.txt and convert that particular input to the
corresponding constraints, which it should pass
to the Z3 solver, and then it should use the Z3
solver's output to write the solution to a file
named output.txt

Don't write anything apart from the Python program;
use Python comments if needed.
```

B.4 Feedback Prompt Templates

These prompt templates are used to provide feedback in the case of Puzzle-LM or PAL.

B.4.1 Programming Errors

```
Your code is incorrect and produces the following
runtime error:<RUN TIME ERROR> for the following input:
<INPUT> rewrite your code and fix the mistake
```

B.4.2 Verification Error

```
Your code is incorrect, when run on the input: <INPUT>
the output produced is <OUTPUT-GENERATED> which
is incorrect whereas one of the correct output is
<GOLD-OUTPUT>.
Rewrite your code and fix the mistake.
```

B.4.3 Timeout Error

```
Your code was inefficient and took more than <TIME-LIMIT>
seconds to execute for the following input: <INPUT>.
Rewrite the code and optimize it.
```

B.5 Logic-LM Prompt Template

The following box describes the prompt for Logic-LM experiments with PuzzleBench, the prompt is used to convert the input to its symbolic representation.

```
Task:
<Description of the Rules of the problem>

Input-Format:
<Description of Textual Representation of Inputs>
Sample-Input:
<Sample Input from Feedback Set>

Output-Format:
<Description of Textual Representation of Outputs>
Sample-Output:
<Output of Sample Input from Feedback Set>

Input problem to be solved:
<Puzzle from the Test Set>

The task is to declare variables and the corresponding constraints on them in SMT2 for the input mentioned above. The variables and constraints should be such that once the variables are solved for, one can use the solution to the variables (which satisfies the constraints) to get to the output in the desired format for the above mentioned input.

Only Write the SMT2 code and nothing else. Write the complete set of SMT2 variables and constraints. Enclose SMT2 code in "'smt2 "'
```

C Experimental Details

C.1 PuzzleBench

All methods are evaluated zero-shot, meaning no in-context demonstrations for the task are provided to the LLM. We choose the zero-shot setting for PuzzleBench because of the structured nature of problems, making it unfair to provide demonstrations of highly related problems instances to the LLM. The LLM is only given a description of the rules of the problem and the task it has to perform. For PAL and Puzzle-LM we present results with 10 solved examples for feedback.

C.2 Hardware Details

All experiments were conducted on an Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz, 32 cores, 64-bit, with 512 KiB L1 cache, 16 MiB L2 cache, and 22 MiB L3 cache. We accessed GPT-4-Turbo and GPT-3.5-Turbo by invoking both models via the OpenAI API. Mixtral 8x7B was also accessed by using the Mistral AI API although the model weights are available publicly. We preferred the API, over running the model locally given the ease of setup because all our other experiments were with APIs.

D Additional Results

D.1 Inference Time

The following tables describes the average inference time for test set instances of a few illustrative problems in PuzzleBench. Puzzle-LM performs much better compared to PAL because PAL programs often tend to be brute force and inefficient whereas the solver can exploit the nature of the input-instance while performing the reasoning with Puzzle-LM.

Problem	PAL	Puzzle-LM
Sudoku	2.01	0.215
Latin Square	5.46	0.2
Cryptarithmetic	0.83	0.73
Independent Set	1.438	0.106
Minimum Dominating Set	0.98	0.112
Sujiko	0.742	0.102
Vertex Cover	1.58	0.105

Table 6: Average inference time in seconds of Puzzle-LM and PAL for test set instances for selected problems in PuzzleBench

E Examples

E.1 Puzzle-LM

This section includes example programs generated by Puzzle-LM for some illustrative problems in PuzzleBench. Each program reads the input from a file, generates the corresponding constraints, calls the solver internally and then uses the solution from the solver to write the output in the desired format to a file.

```python
from z3 import Solver, Int, And, Distinct, Or
import math

# Function to read the board from the file
def read_board(filename):
    with open(filename, 'r') as file:
        board = [[int(num) for num in line.split()] for line in file]
    return board

# Function to write the solved board to the file
def write_board(filename, board):
    with open(filename, 'w') as file:
        for row in board:
            file.write(' '.join(map(str, row))+'\n')

# Function to solve the Sudoku-like puzzle
def solve_puzzle(board):
    n = len(board)
    sqrt_n = int(math.sqrt(n))
    s = Solver()

    # Create a matrix of integer variables
    X = [[Int(f'X_{i+1}_{j+1}') for j in range(n)] for i in range(n)]

    # Constraints for each cell to be between 1 and n
    cells_c = [And(1 <= X[i][j], X[i][j] <= n) for i in range(n) for j in range(n)]

    # Constraints for rows and columns to be distinct
    rows_c = [Distinct(X[i]) for i in range(n)]
    cols_c = [Distinct([X[i][j] for i in range(n)]) for j in range(n)]

    # Constraints for subgrids to be distinct
    subgrids_c = [Distinct([X[i][j]
                        for i in range(k*sqrt_n, (k+1)*sqrt_n)
                            for j in range(l*sqrt_n, (l+1)*sqrt_n)])
                                for k in range(sqrt_n) for l in range(sqrt_n)]

    # Constraints for the given cells to match the input
    given_c = [X[i][j] == board[i][j] for i in range(n) for j in range(n) if board[i][j] != 0]

    # Add all constraints to the solver
    s.add(cells_c + rows_c + cols_c + subgrids_c + given_c)

    if s.check() == 'sat':
        m = s.model()
        solved_board = [[m.evaluate(X[i][j]).as_long() for j in range(n)] for i in range(n)]
        return solved_board
    else:
        return None

# Main process
if _name_ == '_main_':
    input_filename = 'input.txt'
    output_filename = 'output.txt'

    board = read_board(input_filename)
    solved_board = solve_puzzle(board)

    if solved_board:
        write_board(output_filename, solved_board)
    else:
        print("Failed to solve the puzzle")
```

Figure 12: Puzzle-LM example: correct program for sudoku generated by GPT-4-Turbo.

```python
from z3 import *

def parse_input(file_path):
    with open(file_path, 'r') as file:
        lines = [line.strip() for line in file.readlines()]
    board = [list(line.split()) for line in lines[:-2]]
    across_nums = lines[-2].split()
    down_nums = lines[-1].split()
    return board, across_nums, down_nums

def solve_crossword(board, across_nums, down_nums):
    size = len(board)
    solver = Solver()

    # Create a 2D array of Z3 integer variables
    X = [[Int(f"x_{i}_{j}") for j in range(size)] for i in range(size)]

    # Apply constraints based on the initial board
    for i in range(size):
        for j in range(size):
            if board[i][j] == "-":
                solver.add(X[i][j] == -1)
            elif board[i][j] != "0":
                solver.add(X[i][j] == int(board[i][j]))
            else:
                solver.add(And(X[i][j] >= 0, X[i][j] <= 9))

    # Function to generate constraints for placing numbers
    def place_numbers(numbers, vertical=False):
        for num in numbers:
            num_len = len(num)
            placements = []
            for i in range(size):
                for j in range(size):
                    if vertical:
                        if i + num_len <= size:
                            seq = [X[i+k][j] == int(num[k]) for k in range(num_len)]
                            placements.append(And(*seq))
                    else:
                        if j + num_len <= size:
                            seq = [X[i][j+k] == int(num[k]) for k in range(num_len)]
                            placements.append(And(*seq))
            solver.add(Or(*placements))

    place_numbers(across_nums, vertical=False)
    place_numbers(down_nums, vertical=True)

    if solver.check() == sat:
        model = solver.model()
        solution = [[model.evaluate(X[i][j]) for j in range(size)] for i in range(size)]
        return solution
    else:
        return None

def write_output(file_path, solution):
    with open(file_path, 'w') as file:
        for row in solution:
            line = ' '.join('-' if val.as_long() == -1 else str(val.as_long()) for val in row)
            file.write(f"{line}\n")

def main():
    input_file = 'input.txt'
    output_file = 'output.txt'
    board, across_nums, down_nums = parse_input(input_file)
    solution = solve_crossword(board, across_nums, down_nums)
    if solution:
        write_output(output_file, solution)
    else:
        print("No solution found.")

if __name__ == "__main__":
    main()
```

Figure 13: Puzzle-LM example: correct program for keisuke generated by GPT-4-Turbo.

```python
from z3 import *

def read_input(filename):
    with open(filename, 'r') as file:
        n, k = map(int, file.readline().split())
        edges = [tuple(map(int, line.split())) for line in file]
    return n, k, edges

def write_output(filename, answer):
    with open(filename, 'w') as file:
        file.write(answer)

def solve_graph_path(n, k, edges):
    s = Solver()
    path = [Int(f'path_{i}') for i in range(k+1)] # Representing vertices along the path

    # Every vertex in the path must be within graph vertices
    for vertex in path:
        s.add(vertex >= 0, vertex < n)

    # Edges constraint
    for i in range(k):
        s.add(Or([And(path[i] == edge[0], path[i+1] == edge[1])
                for edge in edges] +
                [And(path[i] == edge[1], path[i+1] == edge[0])
                for edge in edges]))

    # Making sure all vertices in the path are distinct
    for i in range(k+1):
        for j in range(i+1, k+1):
            s.add(path[i] != path[j])

    if s.check() == sat:
        return "YES"
    return "NO"

def main():
    n, k, edges = read_input("input.txt")
    answer = solve_graph_path(n, k, edges)
    write_output("output.txt", answer)

if __name__ == "__main__":
    main()
```

Figure 14: Puzzle-LM example: correct program for hamiltonian path generated by GPT-4-Turbo.

```python
from z3 import *

# Function to read the graph and the integer K from the file input.txt
def read_input(filename):
    with open(filename, 'r') as file:
        n, k = map(int, file.readline().split())
        edges = [tuple(map(int, line.split())) for line in file]
    return n, k, edges

# Function to check if it's possible to select at most K vertices
def is_possible(n, k, edges):
    # Initialize Z3 Solver
    solver = Solver()

    # Create a list of boolean variables representing whether a vertex is selected
    vertices = [Bool(f'vertex_{i}') for i in range(n)]

    # Add constraint that at most K vertices can be selected
    solver.add(Sum([If(v, 1, 0) for v in vertices]) <= k)

    # For each edge, add constraint that at least one of the vertices should be selected
    for u, v in edges:
        solver.add(Or(vertices[u], vertices[v]))

    # Check if the problem is solvable with the given constraints
    if solver.check() == sat:
        return "YES"
    else:
        return "NO"

# Function to write the output to a file
def write_output(filename, result):
    with open(filename, 'w') as file:
        file.write(result)

# Main function
def main():
    n, k, edges = read_input('input.txt')
    result = is_possible(n, k, edges)
    write_output('output.txt', result)

if __name__ == "__main__":
    main()
```

Figure 15: Puzzle-LM example: correct program for vertex-cover generated by GPT-4-Turbo.

```python
...

### Constraint for ensuring distinct rows/columns is incorrect
### This constraint ensures that each row/column has distinct elements which over-constraints the problem.
# Constraints for rows and columns to be distinct
for i in range(n):
    solver.add(Distinct([grid[i][j] for j in range(n)]))  # Rows distinct
    solver.add(Distinct([grid[j][i] for j in range(n)]))  # Columns distinct

...
```

```python
...
# Enforcing distinct rows and columns by ensuring sum of differences is non-zero
for i in range(n):
    for j in range(i + 1, n):
        # Ensure row i and row j are different
        solver.add(Sum([If(grid[i][k] != grid[j][k], 1, 0) for k in range(n)]) > 0)
        # Ensure column i and column j are different
        solver.add(Sum([If(grid[k][i] != grid[k][j], 1, 0) for k in range(n)]) > 0)

...
```

Figure 16: Puzzle-LM example: snippet of incorrect program for binairo generated by GPT-4-Turbo and same snippet after correction by feedback.

```python
from z3 import *

# Function to write the solution to the output file
def write_solution_to_file(solution):
    with open("output.txt", "w") as f:
        if solution is None:
            f.write("None")
        else:
            f.write(" ".join(map(str, solution)))

# Function to solve the problem with Z3
def solve_with_z3(array, target_sum):
    solver = Solver()

    # Create a boolean variable for each element in the array to indicate if it is included in the sum
    include_vars = [Bool(f"include_{i}") for i in range(len(array))]

    # Add constraint that the sum of the selected elements should be equal to the target sum
    solver.add(Sum([If(include_vars[i], array[i], 0) for i in range(len(array))]) == target_sum)

    # Try to find a solution
    if solver.check() == sat:
        model = solver.model()
        # Extract the solution: which elements to include
        solution = [array[i] for i in range(len(array)) if is_true(model[include_vars[i]])]
        return solution
    else:
        return None

# Main function to read the input, solve the problem, and write the output
def main():
    with open("input.txt", "r") as f:
        lines = f.read().strip().split("\n")
        array = list(map(int, lines[0].split()))
        target_sum = int(lines[1])

    solution = solve_with_z3(array, target_sum)
    write_solution_to_file(solution)

# Execute the main function
if __name__ == "__main__":
    main()
```

Figure 17: `Puzzle-LM` example: correct program for subset-sum generated by GPT-4-Turbo.

E.2 PAL

This section includes example programs generated by PAL for some illustrative problems in PuzzleBench. Each program reads the input from a file, performs the reasoning and writes the output to another text file.

```
import math

def read_board(filename):
    with open(filename, 'r') as file:
        board = [[int(num) for num in line.split()] for line in file]
    return board

def write_board(filename, board):
    with open(filename, 'w') as file:
        for row in board:
            file.write(' '.join(map(str, row)) + '\n')

def is_valid(board, row, col, num):
    n = len(board)
    box_size = int(math.sqrt(n))
    row_start = (row // box_size) * box_size
    col_start = (col // box_size) * box_size

    if any(num == board[row][i] for i in range(n)):
        return False
    if any(num == board[i][col] for i in range(n)):
        return False
    if any(num == board[i][j] for i in range(row_start, row_start + box_size) for j in range(col_start, col_start +
          box_size)):
        return False
    return True

def solve(board):
    n = len(board)
    for row in range(n):
        for col in range(n):
            if board[row][col] == 0:
                for num in range(1, n+1):
                    if is_valid(board, row, col, num):
                        board[row][col] = num
                        if solve(board):
                            return True
                        board[row][col] = 0
                return False
    return True

def main():
    board = read_board('input.txt')
    if solve(board):
        write_board('output.txt', board)
    else:
        print("No solution exists.")

if __name__ == "__main__":
    main()
```

Figure 18: PAL example: correct program for sudoku generated by GPT-4-Turbo.

```python
import itertools

def read_input(filename):
    """Read the input file and return the board and constraints."""
    with open(filename, "r") as file:
        lines = file.readlines()

    n = len(lines[0].strip().split())
    board = [list(map(int, line.strip().split())) for line in lines[:n]]
    constraints = [tuple(map(int, line.strip().split())) for line in lines[n:]]

    return board, constraints

def write_output(filename, board):
    """Write the solution to the output file."""
    with open(filename, "w") as file:
        for row in board:
            file.write(' '.join(map(str, row)) + '\n')

def check_constraints(board, constraints):
    """Check if the board satisfies all the constraints."""
    n = len(board)
    for x, y in constraints:
        x_row, x_col = divmod(x, n)
        y_row, y_col = divmod(y, n)
        if board[x_row][x_col] >= board[y_row][y_col]:
            return False
    return True

def solve_board(board, constraints):
    """Solve the board with the given constraints."""
    n = len(board)
    rows = [set(range(1, n + 1)) - set(row) for row in board]
    cols = [set(range(1, n + 1)) - set(board[row][col] for row in range(n)) for col in range(n)]

    def backtrack(row, col):
        if row == n:
            return check_constraints(board, constraints)
        if col == n:
            return backtrack(row + 1, 0)
        if board[row][col] == 0:
            for num in rows[row] & cols[col]:
                board[row][col] = num
                rows[row].remove(num)
                cols[col].remove(num)

                if backtrack(row, col + 1):
                    return True

                board[row][col] = 0
                rows[row].add(num)
                cols[col].add(num)
        else:
            return backtrack(row, col + 1)
        return False

    backtrack(0, 0)
    return board

def main():
    input_filename = 'input.txt'
    output_filename = 'output.txt'

    board, constraints = read_input(input_filename)
    solved_board = solve_board(board, constraints)
    write_output(output_filename, solved_board)

if __name__ == "__main__":
    main()
```

Figure 19: PAL example: correct program for futoshiki generated by GPT-4-Turbo.

```python
def dfs(graph, node, visited, N, path_length):
    if path_length == N:
        # If all nodes have been visited, we found a Hamiltonian path
        return True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            visited[neighbor] = True
            if dfs(graph, neighbor, visited, N, path_length + 1):
                return True
            visited[neighbor] = False
    return False

def has_hamiltonian_path(N, edges):
    # Initialize the graph as an adjacency list
    graph = {i: [] for i in range(N)}
    for edge in edges:
        graph[edge[0]].append(edge[1])
        graph[edge[1]].append(edge[0])

    # Try to find Hamiltonian path starting from each node
    for start_node in range(N):
        visited = [False] * N
        visited[start_node] = True
        if dfs(graph, start_node, visited, N, 1):
            return True
    return False

def main():
    # Read the input file
    with open('input.txt', 'r') as file:
        lines = file.readlines()

    # Parse the number of nodes
    N = int(lines[0].strip())

    # Parse the edges
    edges = [tuple(map(int, line.strip().split())) for line in lines[1:]]

    # Check if a Hamiltonian path exists
    path_exists = has_hamiltonian_path(N, edges)

    # Write the output to a file
    with open('output.txt', 'w') as file:
        file.write('YES' if path_exists else 'NO')

# Execute the program
if __name__ == '__main__':
    main()
```

Figure 20: PAL example: correct program for hamiltonian path generated by GPT-4-Turbo.

```python
from itertools import combinations

# Function to read the graph from a file
def read_graph(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()
    N, K = map(int, lines[0].split())
    edges = [tuple(map(int, line.split())) for line in lines[1:]]
    return N, K, edges

# Function to check if a set of vertices is a valid cover
def is_valid_cover(vertices, edges):
    for edge in edges:
        if edge[0] not in vertices and edge[1] not in vertices:
            return False
    return True

# Loading the graph
N, K, edges = read_graph('input.txt')

# Attempting to find a valid cover with at most K vertices
found = False
for num_vertices in range(1, K+1):
    for vertices in combinations(range(N), num_vertices):
        if is_valid_cover(vertices, edges):
            found = True
            break
    if found:
        break

# Writing the result to the file
with open('output.txt', 'w') as file:
    file.write('YES' if found else 'NO')
```

Figure 21: PAL example: correct program for vertex cover generated by GPT-4-Turbo.

```python
def find_combinations_that_sum_to_target(array, target, start=0, path=[]):
    """
    This function finds combinations in 'array' that sum to 'target'.
    It returns a list of all possible combinations that sum up to the target.
    """
    if target == 0:
        return [path]
    if target < 0:
        return []
    combinations = []
    for i in range(start, len(array)):
        if i > start and array[i] == array[i-1]:
            continue
        combinations += find_combinations_that_sum_to_target(array, target-array[i], i+1, path+[array[i]])
    return combinations

# Read input from input.txt
with open('input.txt', 'r') as file:
    input_lines = file.readlines()
    array = list(map(int, input_lines[0].strip().split()))
    target = int(input_lines[1].strip())

# Find combinations
combinations = find_combinations_that_sum_to_target(array, target)

# Write output to output.txt
with open('output.txt', 'w') as file:
    if combinations:
        # Just write the first combination found to the output file
        file.write(' '.join(map(str, combinations[0])))
    else:
        file.write('None')
```

Figure 22: PAL example: correct program for subset sum generated by GPT-4-Turbo.

F Logic-LM

This section describes example runs of Logic-LM for certain problems in PuzzleBench.

```
### Input Instance
## 0 4 3 2
## 2 0 1 0
## 0 0 0 0
## 0 0 2 3


; Declare the variables
(declare-const a11 Int)
(declare-const a12 Int)
(declare-const a13 Int)
(declare-const a14 Int)
(declare-const a21 Int)
(declare-const a22 Int)
(declare-const a23 Int)
(declare-const a24 Int)
(declare-const a31 Int)
(declare-const a32 Int)
(declare-const a33 Int)
(declare-const a34 Int)
(declare-const a41 Int)
(declare-const a42 Int)
(declare-const a43 Int)
(declare-const a44 Int)

; Initialize known values
(assert (= a12 4))
(assert (= a13 3))
(assert (= a14 2))
(assert (= a21 2))
(assert (= a23 1))
(assert (= a43 2))
(assert (= a44 3))

; Each cell must be between 1 and 4
(assert (and (>= a11 1) (<= a11 4)))
(assert (and (>= a22 1) (<= a22 4)))
(assert (and (>= a24 1) (<= a24 4)))
(assert (and (>= a31 1) (<= a31 4)))
(assert (and (>= a32 1) (<= a32 4)))
(assert (and (>= a33 1) (<= a33 4)))
(assert (and (>= a34 1) (<= a34 4)))
(assert (and (>= a41 1) (<= a41 4)))
(assert (and (>= a42 1) (<= a42 4)))

; Each number must appear exactly once in each row
(assert (distinct a11 a12 a13 a14))
(assert (distinct a21 a22 a23 a24))
(assert (distinct a31 a32 a33 a34))
(assert (distinct a41 a42 a43 a44))

; Each number must appear exactly once in each column
(assert (distinct a11 a21 a31 a41))
(assert (distinct a12 a22 a32 a42))
(assert (distinct a13 a23 a33 a43))
(assert (distinct a14 a24 a34 a44))

; Each number must appear exactly once in each 2x2 subgrid
(assert (distinct a11 a12 a21 a22))
(assert (distinct a13 a14 a23 a24))
(assert (distinct a31 a32 a41 a42))
(assert (distinct a33 a34 a43 a44))

; Check and get the model
(check-sat)
(get-model)
```

Figure 23: Logic-LM example: correct constraints for a sudoku instance generated by GPT-4-Turbo.

```
## Input Instance
# 1 2 3 4 5 6 7 8 9 10
# 55


(declare-const a1 Bool)
(declare-const a2 Bool)
(declare-const a3 Bool)
(declare-const a4 Bool)
(declare-const a5 Bool)
(declare-const a6 Bool)
(declare-const a7 Bool)
(declare-const a8 Bool)
(declare-const a9 Bool)
(declare-const a10 Bool)

(assert (= (+ (ite a1 1 0) (ite a2 2 0) (ite a3 3 0) (ite a4 4 0) (ite a5 5 0) (ite a6 6 0) (ite a7 7 0) (ite a8 8 0)
       (ite a9 9 0) (ite a10 10 0)) 55))

(check-sat)
(get-model)
```

Figure 24: Logic-LM example: correct constraints for a subset sum instance generated by GPT-4-Turbo.

```
## Input Instance
# 8 3
# 0 1
# 0 2
# 2 4
# 3 4
# 3 7
# 6 7
# 0 6
# 5 1
# 5 0
# 5 2
# 5 4
# 5 3
# 5 7
# 5 6


; Declare colour variables for each vertex
(declare-const c0 Int)
(declare-const c1 Int)
(declare-const c2 Int)
(declare-const c3 Int)
(declare-const c4 Int)
(declare-const c5 Int)
(declare-const c6 Int)
(declare-const c7 Int)

; Constraint: Each colour variable must be in the range [0, K-1], where K=3 for this problem
(assert (and (>= c0 0) (< c0 3)))
(assert (and (>= c1 0) (< c1 3)))
(assert (and (>= c2 0) (< c2 3)))
(assert (and (>= c3 0) (< c3 3)))
(assert (and (>= c4 0) (< c4 3)))
(assert (and (>= c5 0) (< c5 3)))
(assert (and (>= c6 0) (< c6 3)))
(assert (and (>= c7 0) (< c7 3)))

; Constraint: Adjacent vertices must have different colours
(assert (not (= c0 c1)))
(assert (not (= c0 c2)))
(assert (not (= c2 c4)))
(assert (not (= c3 c4)))
(assert (not (= c3 c7)))
(assert (not (= c6 c7)))
(assert (not (= c0 c6)))
(assert (not (= c5 c1)))
(assert (not (= c5 c0)))
(assert (not (= c5 c2)))
(assert (not (= c5 c4)))
(assert (not (= c5 c3)))
(assert (not (= c5 c7)))
(assert (not (= c5 c6)))

; Check if there is a solution
(check-sat)
```

Figure 25: Logic-LM example: correct constraints for graph coloring instance generated by GPT-4-Turbo.

```
## Input Instance:
# 0 1 3 0
# 4 3 2 1
# 0 4 1 0
# 1 0 0 0
(declare-const a11 Int)
(declare-const a12 Int)
(declare-const a13 Int)
(declare-const a14 Int)

(declare-const a21 Int)
(declare-const a22 Int)
(declare-const a23 Int)
(declare-const a24 Int)

(declare-const a31 Int)
(declare-const a32 Int)
(declare-const a33 Int)
(declare-const a34 Int)

(declare-const a41 Int)
(declare-const a42 Int)
(declare-const a43 Int)
(declare-const a44 Int)

(assert (and (>= a11 1) (<= a11 4)))
(assert (and (>= a12 1) (<= a12 4)))
(assert (and (>= a13 1) (<= a13 4)))
(assert (and (>= a14 1) (<= a14 4)))

(assert (and (>= a21 1) (<= a21 4)))
(assert (and (>= a22 1) (<= a22 4)))
(assert (and (>= a23 1) (<= a23 4)))
(assert (and (>= a24 1) (<= a24 4)))

(assert (and (>= a31 1) (<= a31 4)))
(assert (and (>= a32 1) (<= a32 4)))
(assert (and (>= a33 1) (<= a33 4)))
(assert (and (>= a34 1) (<= a34 4)))

(assert (and (>= a41 1) (<= a41 4)))
(assert (and (>= a42 1) (<= a42 4)))
### SYNTAX ERROR
(assert (>= a43 1) (<= a43 4)))
### SYNTAX ERROR
(assert (and (>= a44 1) (<= a44 4)))

(assert (distinct a11 a12 a13 a14))
(assert (distinct a21 a22 a23 a24))
(assert (distinct a31 a32 a33 a34))
(assert (distinct a41 a42 a43 a44))

(assert (distinct a11 a21 a31 a41))
(assert (distinct a12 a22 a32 a42))
(assert (distinct a13 a23 a33 a43))
(assert (distinct a14 a24 a34 a44))

(assert (distinct a11 a12 a21 a22))
(assert (distinct a13 a14 a23 a24))
(assert (distinct a31 a32 a41 a42))
(assert (distinct a33 a34 a43 a44))

(assert (distinct a11 a12 a13 a14))
(assert (distinct a21 a22 a23 a24))
(assert (distinct a31 a32 a33 a34))
(assert (distinct a41 a42 a43 a44))

(check-sat)
(get-model)
```

Figure 26: Logic-LM example: syntax error (highlighted by a comment) in constraints for sudoku instance generated by GPT-3.5-Turbo.