Do LLMs Understand Constraint Programming? Zero-Shot Constraint Programming Model Generation Using LLMs

Yuliang Song ✉
Department of Mechanical and Industrial Engineering, University of Toronto, Canada

Eldan Cohen ✉
Department of Mechanical and Industrial Engineering, University of Toronto, Canada

Abstract

Large language models (LLMs) have gained significant attention for their ability to solve complex tasks such as coding and reasoning. In this work, we aim to evaluate their ability to generate constraint programming (CP) models in a zero-shot setting, emphasizing model correctness and conformity to user-specified output formats. We propose a novel, iterative approach for zero-shot CP modeling that translates natural language problem descriptions into valid CP models and supports solution extraction to predefined output formats to facilitate effective adoption by domain experts and enable automated performance evaluation. To evaluate our approach, we introduce the Constraint Programming Evaluation (CPEVAL) benchmark, derived from a diverse set of CP problems in CSPLib, coupled with an automated evaluation suite for large-scale assessment. We augment CPEVAL with paraphrased variants to assess robustness across linguistic variation and mitigate bias in the evaluation due to data memorization. Our extensive experiments across eight prominent LLMs and two CP modeling languages, MiniZinc and PyCSP3, show that our proposed iterative Two-Step method significantly enhances model correctness and conformity to user-specified output formats. Furthermore, we observe that larger LLMs demonstrate superior performance, with DeepSeek-R1 emerging as the top performer across both CP languages. We also observe that LLMs generally perform better in MiniZinc than in PyCSP3.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Natural language processing

Keywords and phrases Constraint Programming, Large Language Models, Benchmark, MiniZinc

This paper is based on accepted paper at LION 2025 [17].

1 Introduction

Constraint Programming (CP) is a powerful paradigm for solving complex combinatorial problems across diverse domains [16]. However, the specialized expertise required to translate domain requirements into formal CP models remains a barrier to broader adoption, underscoring the demand for automated tools to simplify CP model generation from natural language descriptions [14, 18]. Large language models (LLMs) have gained growing attention for their capabilities in problem solving and reasoning [20], making them promising tools for CP modeling. Previous work has explored their ability to generate CP models from natural language descriptions [18, 6], including MiniZinc code template generation [3], scheduling-specific constraint generation [9], and few-shot CP modeling enhanced by retrieval-augmented generation [12]. In this work, we evaluate LLMs' ability to generate CP models in a zero-shot setting, emphasizing both model correctness and conformity to user-specified output formats. Our contributions are:

1. We propose a novel, iterative approach for zero-shot CP modeling that translates natural language problem descriptions into valid CP models and extracts solutions in user-specified formats, facilitating both adoption by domain experts and automated evaluation.

2. We introduce the Constraint Programming Evaluation (CPEVAL) benchmark, derived from a diverse set of CP problems in CSPLib, along with an automated evaluation suite for large-scale assessment. We augment CPEVAL with paraphrased variants to assess robustness across linguistic variation and mitigate evaluation bias from data memorization.
3. We perform extensive experiments across eight prominent LLMs and two widely-used CP languages (MiniZinc[13] and PyCSP3[10]). Results show that (1) our proposed iterative Two-Step method significantly improves model correctness and output format conformity, with DeepSeek-R1 excelling in both CP languages; and (2) all LLMs experience performance degradation on paraphrased problems with varying extent.

2 Zero-Shot CP Modeling with LLMs

2.1 Problem Definition

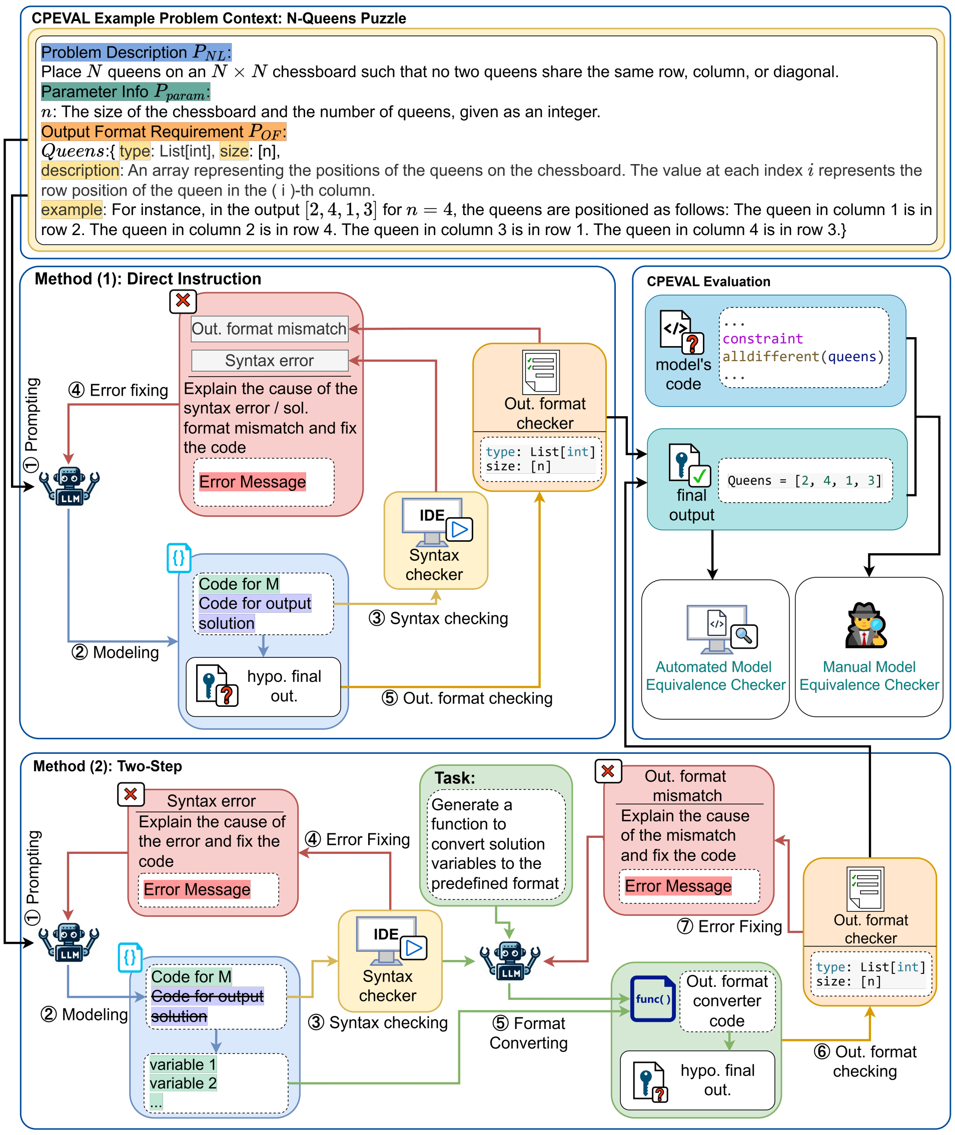
The input to our system consists of: (1) $\mathbf{P_{NL}}$: A natural-language description of the problem; (2) $\mathbf{P_{param}}$: A concise description of input parameters, specifying their meanings, data formats, and types; (3) $\mathbf{P_{OF}}$: Specifications of the expected output formats, namely a list of required variables and their description and data type, as well as an example output. Subsequently, our system, denoted as $\mathcal{F}$, transforms these components into a CP model $\mathcal{M}$, formally, $\mathcal{M} = \mathcal{F}(P_{NL}, P_{param}, P_{OF})$. See Fig. 1 for example P_{NL}, P_{OF}, and P_{param}.

The motivation for including explicit output format specification stems from the variety of valid equivalent ways to model the same problem (e.g., disjunctive vs. time-indexed formulation for scheduling problems [4]). Consequently, the representation of solutions can vary significantly, placing an additional burden on users who must interpret the generated model with potentially unfamiliar solution representations.

2.2 Iterative Modeling Workflow

We consider two workflows for iterative generation. First, we introduce the *Direct Instruction* method, which takes the problem context and explicitly instructs the LLM to generate code that solves the problem and outputs the solution in the user-specified output format. However, enforcing a complex output format increases the difficulty of the generation task, especially when the required format has an intricate structure (e.g., scheduling timetable or game board layout). To address this, we propose a *Two-Step* method, where the first step focuses on generating the CP model, and the second focuses on extracting the solution variables and transforming them into the user-specified output format. For effective output format validation, we implement an automated output format checker that takes the data types from P_{OF}, assesses the output's conformity to the prescribed requirements, and produces error reports when it fails to meet the expected format.

Direct Instruction (DI) Fig. 1 (Method 1) illustrates the direct instruction workflow. The first step is to prompt the LLM with the problem context P, instructing it to generate a CP model in the chosen modeling language and output a final solution in the user-specified format P_{OF}. The generated code is extracted and compiled in an IDE with runtime information collected. If compilation fails, a self-improvement process (§2.2.1) is triggered to correct the code and reattempt compilation. Once the generated code passes the syntax check, it is executed, and the solver status is extracted upon completion or timeout. If the solver returns UNKNOWN or UNSATISFIABLE, the model is deemed semantically incorrect, and the generation workflow is aborted. Otherwise, if the solver returns a SATISFIABLE



CPEVAL Example Problem Context: N-Queens Puzzle

Problem Description P_{NL}:
Place N queens on an $N \times N$ chessboard such that no two queens share the same row, column, or diagonal.
Parameter Info P_{param}:
n: The size of the chessboard and the number of queens, given as an integer.
Output Format Requirement P_{OF}:
$Queens$:{ type: List[int], size: [n],
description: An array representing the positions of the queens on the chessboard. The value at each index i represents the row position of the queen in the (i)-th column.
example: For instance, in the output $[2, 4, 1, 3]$ for $n = 4$, the queens are positioned as follows: The queen in column 1 is in row 2. The queen in column 2 is in row 4. The queen in column 3 is in row 1. The queen in column 4 is in row 3.}

```
...
constraint alldifferent(queens)
...
```

```
Queens = [2, 4, 1, 3]
```

```
type: List[int]
size: [n]
```

Task: Generate a function to convert solution variables to the predefined format

Explain the cause of the syntax error / sol. format mismatch and fix the code

Explain the cause of the error and fix the code

Explain the cause of the mismatch and fix the code

Figure 1 Illustration of the proposed methods using the classic N-Queens problem from the CPEVAL benchmark.

or OPTIMAL status, an output format checker evaluates whether the data types of the output variables align with the user-specified formats P_{OF}. Subsequently, if discrepancies are detected, an error message detailing the discrepancies is sent to the LLM, prompting it to revise the output format through the self-improvement process. The generated code is accepted once its outputs conform to the user-specified output format.

Two-Step Method (2S) Fig. 1 (Method 2) presents the Two-Step Method workflow. In the *modeling stage*, the problem context (P_{NL}, P_{param}) and user-specified output format (P_{OF}) are provided to the LLM for modeling. However, the LLM is explicitly instructed to only consider the output format requirements but not to generate any code to comply with them. The hypothesis code is then evaluated for syntax correctness, and if it fails, an iterative self-improvement process is triggered for debugging. Upon successfully passing the syntax check, the solver must return a SATISFIABLE or OPTIMAL status; otherwise, the model is deemed semantically incorrect, and the workflow is aborted.

Once the model is solved, all decision variables used in the model are saved to a local file, and the workflow proceeds to the *formatting stage*. The LLM is now instructed to generate Python code that transforms the stored decision variables into the user-specified format. This step requires the LLM to interpret the problem context and may involve generating code for additional calculations or adjustments to ensure compliance with P_{OF}. Similar to the DI method, the generated output is then validated by an output format checker, and if mismatches arise, a self-improvement process is triggered to correct them.

2.2.1 Self-Improving

We employ an iterative self-debugging approach following [5], targeting two types of code defects: syntax errors identified through IDE compilation and output format mismatches detected by the output format checker. In the initial step, we input the defective code, along with any error messages, to the same LLM that generated the original code. In case of syntax errors, the IDE's runtime error messages are used to guide debugging. In cases of output format mismatches detected by the output format checker, a message outlining discrepancies between the generated output and the expected format is provided to facilitate refinement. Subsequently, the LLM is instructed to provide a concise explanation of the error's cause and to produce a revised version of the code which is then executed in the corresponding IDE. If new errors are encountered, the updated code and corresponding error messages are fed back to the LLM for further refinement. This iterative process continues until the hypothesis code passes its corresponding checker or a user-specified self-improving attempt limit is reached.

3 Experimental Setup

3.1 Models

We employed eight prominent open-source models including Llama-3.3-70B[1], DeepSeek-V3-685B [11], DeepSeek-R1 [8], QWen2.5-70B and QWen2.5-Coder-32B [19], Phi-3.5 mini (3.8B) [1], and Phi-4 (14.7B) [2], as well as the closed-source model ChatGPT-4o (2024-08-06)[2].

3.2 Datasets

We employ CSPLib [7] as the primary source of problems to construct the CPEVAL benchmark. Each problem consists of a natural language description, reference models in various CP languages, and instance data files, when applicable.

Due to time and computational constraints associated with large-scale evaluation, problems were selected based on criteria facilitating evaluation purposes. We included all problems with reference MiniZinc model that can be solved within 10 minutes. As a result, the CPEVAL dataset comprises 30 problems, including 9 constraint optimization problems (COP) and 21 constraint satisfaction problems (CSP) across seven categories and varying levels of complexity. Each problem in CPEVAL comprises three key components:

1. **Problem Description ($\mathbf{P_{NL}}$):** We pre-process the problem descriptions to exclude any images, references, unrelated information, and example solving steps.

[1] https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/
[2] https://openai.com/index/gpt-4o-system-card/

2. **Input Parameters ($\mathbf{P_{param}}$):** The input parameters are derived from the CSPLib parameter files, each accompanied by a description of its data type, structure, and meaning. We verify each instance for validity; if multiple instances are available, we select up to three of the simplest ones based on computational complexity (e.g., preferring a 4-queens instance over a 100-queens instance). During modeling, the LLM is instructed to generate code that loads these parameters from the file, conditioned on this description.
3. **Required Output Format ($\mathbf{P_{OF}}$):** For each CSPLib problem, we provide a predefined output format requirement that states the data type, structure, and representation of all needed output variables.

3.3 Paraphrase Generation

The original CSPLib problems are widely known, and their solutions are publicly available, raising concerns that evaluation problems have been used to train the LLM, which could lead to over-estimation of models' performance due to memorization of the solution rather than CP modeling capabilities. To assess whether LLMs can effectively interpret a given problem context and generate correct CP models, we therefore opt to evaluate LLMs' performance under linguistic variations. Specifically, an LLM is instructed to paraphrase the original problem description to simulate how a user might request a modeling service from a CP expert. The goal is to introduce varied linguistic expressions and simulate different linguistic framings, allowing for the evaluation of the LLM on problems presented in unfamiliar or diverse linguistic forms, all while preserving the original underlying semantics. To ensure a diverse set of paraphrases that could effectively challenge the LLM's understanding and generalization capabilities, we introduce two paraphrasing styles: *Precision* mode, which minimally alters semantics while preserving technical accuracy, and *Colloquial* mode, which uses a casual, conversational tone closer to a layperson's request for assistance. For each original problem, we employ Claude-3.5-Sonnet[3] to generate three paraphrased versions in each style, with each paraphrase treated as a distinct problem instance, resulting in 180 paraphrased problems overall. Since generating paraphrases with the same LLM being evaluated could bias the paraphrases toward that model's language patterns and understanding, we excluded the Claude series from the evaluation.

3.4 Implementation Details

We evaluated the Direct Instruction Method and the Two-Step Method across the CP modeling languages PyCSP3 (using the ACE solver) and MiniZinc (using Gecode). A timeout of 10 minutes was set for each solver. For both syntax and output format errors, the number of self-improvement attempts was limited to 3. The Direct Instruction method was also compared against a Standalone Mode, where the LLM receives the same prompt but is allowed only a single attempt to deliver the required output without iterative refinement.

For all LLMs, we set the temperature to zero for deterministic decoding and generate one model per problem. However, we observed that ChatGPT-4o, with a temperature setting of zero, did not follow greedy decoding, consistent with prior findings [15]. To mitigate bias to a single given sample, we sampled multiple models per problem from ChatGPT-4o and reported average performance across samples. Specifically, five models were sampled for each original problem, and three models were sampled for each paraphrased problem.

[3] https://www.anthropic.com/claude/sonnet

3.5 Evaluation

We use two criteria to evaluate system performance: (1) Output Format Alignment, which checks whether the generated output complies with the user-specified format; and (2) Model Equivalence, which verifies whether each generated model is semantically equivalent to a reference model. We begin with an output format alignment metric, followed by a manual model equivalence check. However, manually interpreting and aligning diverse output formats is expensive and only feasible at small scale. To enable large-scale evaluation, we also present an automated, unit-test-based approach that closely approximates our manual evaluation.

Output Format Alignment The required output variables are evaluated using the output format checker described in(§2.2). A generated model's output is deemed to align with the required output format P_{OF} if it passes this checker, denoted as $FC(P_{OF}, \mathcal{A}) = 1$; otherwise, 0. The output format alignment rate (OFAR) across N problems is then defined as OFAR $= \frac{1}{N}\sum_{i=1}^{N} FC(P_{OF}, \mathcal{A})$.

Manual Model Equivalence Checker We consider two CP models equivalent if their constraints and solutions are semantically aligned. To assess this, we manually inspect: (1) Model alignment: we inspect whether the generated model aligns with the reference model in terms of constraints and objectives; (2) Solution verification: we extract the final assignment of decision variables from the generated model, map them to the reference model's decision variables, and verify the consistency with its constraints. Both checks must pass for a generated model to be considered equivalent, denoted as $\mathbb{I}(\hat{M}_i \equiv M_i) = 1$; otherwise, 0. The *manual model equivalence rate* (MMER)—the fraction of generated models equivalent to their reference models across N problems—is then defined as MMER $= \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}(\hat{M}_i \equiv M_i)$.

Automated Model Equivalence Checker We approximate the human evaluation process by creating evaluation scripts for each problem. These scripts take solutions in the predefined output format and verify whether they satisfy the problem's constraints and logic as outlined in the problem description, as well as optimality for COPs. A solution that passes the checker is deemed equivalent to the reference model ($\mathbb{P}\mathbb{I}(\hat{M}_i \equiv M_i) = 1$); otherwise, 0. The *automated model equivalence rate* (AMER) is defined as AMER $= \frac{1}{N}\sum_{i=1}^{N} \mathbb{P}\mathbb{I}(\hat{M}_i \equiv M_i)$.

4 Results

In §4.1, we present results from a small-scale manual evaluation on the original CPEVAL problems, comparing the effectiveness of proposed methods and validating the effectiveness of the automated model equivalence checker with the manual approach. Then, in §4.2, we perform a large-scale automated evaluation across eight LLMs, evaluating their performance on the original CPEVAL problems as well as the paraphrased variants.

4.1 Small Scale Manual Evaluation

We start with a small-scale evaluation on the original CPEVAL problems using ChatGPT-4o with manual model equivalence checking, as shown in Table 1. Here, MMER ∩ OFAR denotes the proportion of generated models that pass both the manual model equivalence check and the output format checker. "Match" denotes the Jaccard similarity between the set of generated models passing MMER ∩ OFAR and the set of generated models passing AMER.

Table 1 Evaluation results on original problems using ChatGPT-4o with 5 samples per problem. Method marked with † indicates a standalone model without self-improvement.

Method	Lang.	MMER (%)	OFAR (%)	MMER ∩ OFAR (%)	AMER (%)	Match (Jaccard)
Direct Instr.†	mzn	50	49	31	31	1
	pycsp3	13	19	13	13	1
Direct Instr.	mzn	69	63	55	55	1
	pycsp3	53	60	53	53	1
Two-Step	mzn	67	79	67	66	0.98
	pycsp3	54	63	54	54	1

Comparison of Modeling Methods Direct Instruction† (ChatGPT-4o in standalone mode) exhibits relatively low performance in generating semantically correct models in both MiniZinc and PyCSP3, with MMER scores of 50% and 13%, respectively. Upon investigation into the failed cases, we observed that this is primarily due to syntax errors stemming from the absence of a self-improvement process. Moreover, its ability to align with the predefined output format is weak, resulting in the lowest MMER ∩ OFAR score for both MiniZinc and PyCSP3. In contrast, Direct Instruction with self-refinement significantly improves MMER scores for MiniZinc (50% → 69%) and PyCSP3 (13% → 53%), along with noticeably higher MMER ∩ OFAR scores for MiniZinc (31% → 55%) and PyCSP3 (13% → 53%).

While the Two-Step Method achieves comparable MMER scores to Direct Instruction in MiniZinc (67% vs. 69%), it is significantly more effective at ensuring correct output formatting, leading to a significantly higher OFAR score than Direct Instruction (79% vs. 63%). Additionally, the Two-Step Method consistently outperforms Direct Instruction in OFAR across all evaluated CP modeling languages and LLMs (results omitted due to space). Therefore, we focus on the Two-Step Method in the following sections.

Effectiveness of Automated Evaluation We gauge the effectiveness of the automated evaluation by comparing the alignment between the MMER ∩ OFAR score (models that are semantically correct and deliver solutions in the predefined format) and the AMER score.

Table 2 Automated evaluation on original and paraphrased problems using the Two-Step Method.

			Original		Paraphrased	
Model	Params	Lang.	OFAR (%)	AMER (%)	OFAR (%)	AMER (%)
DeepSeek-R1	685B	MZN	80	80	80	74
		PYCSP3	67	63	61	54
DeepSeek-V3	685B	MZN	80	70	76	58
		PYCSP3	60	50	62	51
ChatGPT-4o	Unknown	MZN	79	66	76	65
		PYCSP3	63	54	59	47
llama3.3	70B	MZN	67	57	57	45
		PYCSP3	43	30	33	23
QWen2.5	70B	MZN	63	60	59	49
		PYCSP3	47	33	46	33
QWen2.5-Coder	32B	MZN	63	47	54	41
		PYCSP3	53	43	45	30
Phi-4	14.7B	MZN	20	17	16	13
		PYCSP3	37	20	33	26
Phi-3.5 mini	3.8B	MZN	0	0	3	0
		PYCSP3	7	0	9	2

Notably, across all methods and CP languages, AMER aligns closely with MMER $\cap$ OFAR. We also observe consistently high Match scores, with a value of 1 across all methods and CP modeling languages, except for the Two-Step Method on MiniZinc (0.98). This indicates that our automated model equivalence checker reliably identifies models that are semantically equivalent to their reference counterparts and output solutions in the predefined format.

4.2 Large Scale Automated Evaluation

Table 2 presents the results of our large-scale automated (unit test-based) evaluation for original and paraphrased problems across eight prominent LLMs. Overall, we observe that larger models tend to achieve higher AMER scores. Among all LLMs, DeepSeek-R1 attains the highest AMER score with MiniZinc (80%) and PyCSP3 (63%), followed by DeepSeek-V3 (70%) for MiniZinc and ChatGPT-4o (54%) for PyCSP3. Interestingly, QWen2.5-Coder outperforms Llama3.3 and QWen2.5, both significantly larger, on PyCSP3, demonstrating the potential benefits of code-related pre-training in Python code generation tasks. The smaller model Phi-4 obtains significantly lower AMER scores, whereas the even smaller Phi-3.5 mini struggles to generate valid CP models.

CP Languages Comparison We observe that modeling with MiniZinc consistently outperforms PyCSP3 in the AMER score across all LLMs. This suggests that while LLMs exhibit strong proficiency in generating Python code, this ability does not directly translate into effective CP modeling with PyCSP3. Moreover, we observe that the robustness of models to linguistic variations varies across the two modeling languages, as discussed below.

Performance on Paraphrased Problems All LLMs exhibit degradation in AMER when moving to the paraphrased problems, though the extent of degradation varies. DeepSeek-R1 achieves the highest AMER; however, it experiences a moderate drop on paraphrased tasks for both MiniZinc (80% $\rightarrow$ 74%) and PyCSP3 (63% to 54%). DeepSeek-V3, another strong performer, undergoes a notable performance drop on MiniZinc for paraphrased problems (70%$\rightarrow$58%) but maintains relatively stable AMER scores for PyCSP3 (50% $\rightarrow$ 51%). In contrast, ChatGPT-4o's AMER on paraphrased tasks with MiniZinc remains closely aligned with its performance on the original problems (66% $\rightarrow$ 65%), while its AMER for PyCSP3 shows greater sensitivity to paraphrased problems, with a significant drop from 54% to 47%. Llama3.3-70B shows a consistent performance drop across languages, while QWen2.5-70B demonstrates a notable decline on MiniZinc (60% $\rightarrow$ 49%) but remains stable on PyCSP3. Conversely, QWen2.5-Coder-32B experiences a significant AMER reduction on both MiniZinc (47% $\rightarrow$ 41%) and PyCSP3 (43% $\rightarrow$ 30%), highlighting differences in sensitivity across problem formulations and languages.

5 Conclusion

We proposed novel iterative workflows that significantly improve LLMs' CP modeling performance in zero-shot settings. We also introduced CPEVAL, an automated benchmark augmented with paraphrased problems to account for potential data memorization. Our results show that DeepSeek-R1 obtains the best performance, that MiniZinc typically yields better results than PyCSP3, and that LLMs exhibit some performance degradation on paraphrased problems. Future work includes developing CP-specific prompting strategies and exploring reinforcement learning-based fine-tuning based on CPEVAL's feedback signals (e.g., syntax errors, semantic failures, and output format mismatches).

References

1 Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv:2404.14219*, 2024.

2 Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, et al. Phi-4 technical report. *arXiv:2412.08905*, 2024.

3 Boris Almonacid. Towards an automatic optimisation model generator assisted with generative pre-trained transformer. *arXiv:2305.05811*, 2023.

4 Sadia Azem, Riad Aggoune, and Stéphane Dauzère-Pérès. Disjunctive and time-indexed formulations for non-preemptive job shop scheduling with resource availability constraints. In *IEEE IEEM*, pages 787–791, 2007.

5 Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *ICLR*, 2024.

6 Eugene C Freuder. Conversational modeling for constraint satisfaction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 22592–22597, 2024.

7 Ian P Gent and Toby Walsh. Csplib: a benchmark library for constraints. In *CP*, pages 480–481, 1999.

8 Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv:2501.12948*, 2025.

9 Connor Lawless, Jakob Schoeffer, Lindy Le, Kael Rowan, Shilad Sen, Cristina St. Hill, Jina Suh, and Bahareh Sarrafzadeh. “i want it that way”: Enabling interactive decision support using large language models and constraint programming. *ACM TIIS*, 14(3):1–33, 2024.

10 Christophe Lecoutre and Nicolas Szczepanski. Pycsp3: modeling combinatorial constrained problems in python. *arXiv:2009.00326*, 2020.

11 Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv:2412.19437*, 2024.

12 Kostis Michailidis, Dimos Tsouros, and Tias Guns. Constraint modelling with llms using in-context learning. In *CP*, 2024.

13 Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *CP*, pages 529–543, 2007.

14 Barry O’Sullivan. Automated modelling and solving in constraint programming. In *AAAI*, pages 1493–1497, 2010.

15 Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. An empirical study of the non-determinism of chatgpt in code generation. *ACM TOSEM*, 2025.

16 Fotios Petropoulos, Gilbert Laporte, Emel Aktas, Sibel A Alumur, Claudia Archetti, Hayriye Ayhan, Maria Battarra, Julia A Bennell, Jean-Marie Bourjolly, John E Boylan, et al. Operational research: methods and applications. *JORS*, 75(3):423–617, 2024.

17 Yuliang Song and Eldan Cohen. Do llms understand constraint programming? zero-shot constraint programming model generation using llms. In *Proceedings of the 19th Learning and Intelligent Optimization Conference (LION-25)*, page In press, 2025.

18 Dimos Tsouros, Hélène Verhaeghe, Serdar Kadıoğlu, and Tias Guns. Holy grail 2.0: From natural language to constraint models. *arXiv:2308.01589*, 2023.

19 An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv:2412.15115*, 2024.

20 Fei Yu, Hongbo Zhang, Prayag Tiwari, and Benyou Wang. Natural language reasoning, a survey. *ACM CSUR*, 56(12):1–39, 2024.