

# Test-driven Generation of Constraint Satisfaction Problems Using Large Language Models

Christian Bähnisch<sup>1,†</sup>, Lothar Hotz<sup>1,\*,†</sup>, Alexander Felfernig<sup>2</sup> and Sebastian Lubos<sup>2</sup>

<sup>1</sup>Hamburger Informatik Technologie-Center e.V., Hamburg, Germany

<sup>2</sup>Graz University of Technology, Graz, Austria

## Abstract

Constraint Satisfaction Problems (CSPs) are a foundational technology used to solve a wide range of real-world problems. A critical factor of the success of constraint-based systems is the efficient acquisition of knowledge, where domain experts and knowledge engineers must reach an agreement on the correctness of the evolving knowledge base as early as possible. In this paper, we introduce a novel approach to automate the generation of CSP-based knowledge bases by following a test-driven approach. We demonstrate how large language models (LLMs) can be leveraged to generate knowledge bases where validation is performed on the basis of pre-defined basic test routines. This approach reduces the overhead typically associated with knowledge base development and maintenance.

## Keywords

Constraint Satisfaction Problems, Large Language Models, Knowledge Acquisition, Automated Generation, Test-Driven Development

## 1. Introduction

Knowledge acquisition for constraint-based systems is a complex and often time-consuming task. It involves formalizing sometimes intangible domain knowledge into structured models consisting of variables, domain definitions, and corresponding constraints [1, 2, 3]. Through their ability to compute and reason about possible solutions, constraint-based systems are particularly popular to support tasks such as knowledge-based configuration [4] and scheduling [5].

In the field of software engineering, large language models (LLMs) have become invaluable tools, assisting developers in a range of tasks, from code completion and review to generating test cases and documentation [6]. LLMs such as ChatGPT<sup>1</sup> and Copilot<sup>2</sup> have shown significant potential in supporting software development tasks, including debugging and refactoring. Particularly, the combination of LLMs with compilers or interpreters has facilitated self-evaluating workflows, where generated code is immediately tested, and errors are fed back to the model for iterative improvement [7, 8, 9]. Such test-driven approaches have proven to be effective in refining code and maintaining syntactic correctness [6].

A simple example of applying LLMs in the context of code generation is GPT-ENGINEER<sup>3</sup>, which exploits natural language prompts to generate game logic with a corresponding HTML-based user interface. By integrating a large language model with a compiler or interpreter, the model can self-correct by using error messages from failed executions – an approach supported by tools such as LANGCHAIN.<sup>4</sup>

*ConfWS'25: 27th International Workshop on Configuration, Oct 25–26, 2025, Bologna, Italy*

\*Corresponding author.

<sup>†</sup>These authors contributed equally.

✉ christian.baehnisch@uni-hamburg.de (C. Bähnisch); lothar.hotz@uni-hamburg.de (L. Hotz); alexander.felfernig@tugraz.at (A. Felfernig); sebastian.lubos@tugraz.at (S. Lubos)

ORCID 0000-0001-7370-7726 (L. Hotz)



© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

<sup>1</sup><https://chatgpt.com/> (accessed: May 2025)

<sup>2</sup><https://copilot.microsoft.com/> (accessed: May 2025)

<sup>3</sup><https://github.com/AntonOsika/gpt-engineer> (accessed: May 2025)

<sup>4</sup><https://www.langchain.com/> (accessed: May 2025)

system:

You are an agent designed to solve constraint problems, you write python code to solve and verify constraint problems. Your code is run and verified by the user. If you get an error, debug your code and try again. You only use the constraint solver `pychoco`. You conclude your python code with a variable "solutions" and verify it with the method call `verify_solutions(solutions, <problem_name>`, where `<problem_name>` is given to you as the first word of the instructions. "solutions" should be a list of all solutions to the constraint problem. If verification fails, you correct your solution.

Use the following example as a template for solving constraint problems with `pychoco`:

User: `ints10`. find numbers one to 10.

Assistant:

```
``python
from pychoco.model import Model
from test_tool import verify_solutions

model = Model()
x = model.intvar(1, 10, "x")

solver = model.get_solver()
solutions = [s.get_int_val(x) for s in solver.find_all_solutions()]
verify_solutions(solutions, "ints10")
``
```

---

user:

`tsp`. solve the traveling salesman problem, use the following problem instance:

```
# Number of cities
C = 4
```

```
# Distance matrix
D = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
```

the solution should be a list of valid solutions, each solution being a list of integers representing the cities to be visited.

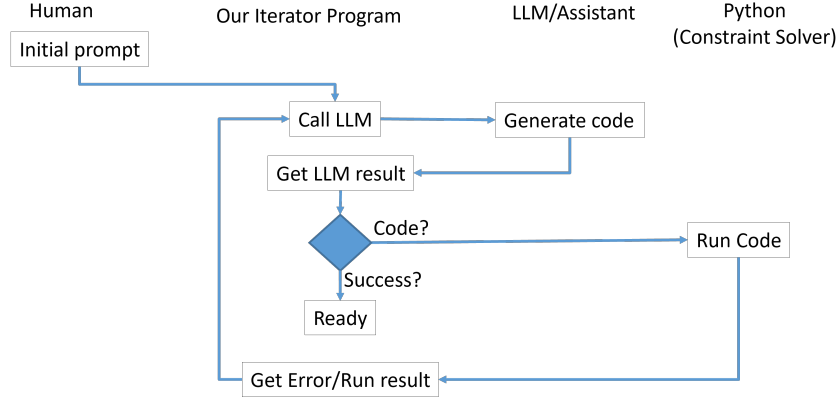
**Figure 1:** General prompt including an example for the LLM on how to call the verification method. The specific user prompt provides a problem instance. (Human created.)

Inspired by these advancements, we investigate in which way large language models can support knowledge acquisition and engineering for constraint satisfaction problems (CSPs). The research questions we aim to answer are the following:

1. Can LLMs generate reliable knowledge models?
2. What kind of prompts are effective?

In our earlier work [10], we introduced first ideas on applying test-driven principles to the generation of constraint satisfaction problems using LLMs. The current paper extends that work in two major ways: (i) we employ more recent LLM technology (GPT-4.1), which provides improved stability and reasoning capabilities compared to the models used previously, and (ii) we enhance the verification process, in particular for more demanding problems such as the traveling salesperson problem (TSP), where we introduce refined verification routines that guide the model towards handling optimality aspects. We use similar example problems in both papers.

In this paper, we propose a test-routine-guided approach for the generation of knowledge bases representing constraint satisfaction problems. In our approach, basic test routines act as a background knowledge oracle, guiding an LLM to automatically generate and validate generated CSPs. Based on



**Figure 2:** Architecture iterating through LLM-generated code. Verification is done by enforcing the LLM using the method `verify_solutions`, which is then executed with Python. For termination, a maximum number of iterations can be given (not shown).

a provided test routine that acts as a validation checkpoint, an LLM generates the knowledge base. This approach not only supports the automated generation of constraint models but also ensures the correctness of the knowledge base by leveraging test-driven validation.

The major contributions of this paper are the following:

1. We show how to integrate LLMs into knowledge acquisition processes for constraint-based systems.
2. Based on test-driven development, we are able to generate CSPs which – on a semantic level – conform completely with the underlying domain knowledge.
3. We show the applicability of the proposed approach with different commonly known CSPs.

The remainder of this paper is organized as follows. In Section 2, we review the state of the art in knowledge generation. In Section 3, we present our approach for test-routine-guided CSP generation. Section 4 details our verification method. Section 5 presents details of the implementation and the results of some solved CSPs. A discussion of the evaluation results of our generation approach is provided in Section 6. In Section 7, we conclude the paper by summarizing the research results and indicating different future research directions.

## 2. State of the Art in Knowledge Base Generation

Many real-world problems can be defined as constraint satisfaction problems (CSPs) requiring knowledge engineers to translate domain-specific constraints into corresponding formal representations. This modeling process is often complex and labor-intensive and triggers a need for enhanced knowledge acquisition support [11]. We now provide an overview of the existing state of the art in the automated generation of knowledge bases. These approaches range from the generation of knowledge bases from graphical descriptions, the learning and generation of knowledge bases from examples, and finally, the LLM-based knowledge base generation which is a discipline of specific relevance for the work presented in this paper [12].

*Generating Knowledge Bases from Graphical Descriptions.* The modeling of domain knowledge on the basis of graphical models has a long tradition in knowledge engineering. Examples thereof are the graphical representation of ontologies on the basis of the PROTEGE<sup>5</sup> environment or the representation of variability properties in terms of feature models in the FEATUREIDE environment.<sup>6</sup> Such feature models can then be translated into corresponding constraint-based or SAT-based representations [13].

<sup>5</sup><https://protege.stanford.edu/> (accessed: May 2025)

<sup>6</sup><https://featureide.de/> (accessed: May 2025)

**Listing 1:** Verification method for the TSP problem. Other verifications are removed. (Human created.)

```
1 def verify_solutions(  
2     solutions: List, problem: Literal["queen"] | Literal["tsp"] | Literal["simple"  
3 ]  
4 ) -> bool:  
5     # Code for other problems removed.  
6  
7     if problem == "tsp":  
8         try:  
9             # valids = [test_tsp.is_valid_route(sol) for sol in solutions]  
10            # if sum(valids) == 2 and len(solutions) == 2:  
11            #     print("All solutions are correct")  
12            # else:  
13            #     print(f"{sum(valids)} of the {len(valids)} solutions are correct  
14            #         . Expected 2 solutions")  
15  
16            evals = [test_tsp.eval_route(sol) for sol in solutions]  
17            cost = sum([s[0] for s in evals])  
18            ham = sum([s[1] for s in evals])  
19            start = sum([s[2] for s in evals])  
20            correct = sum([(s[0] and s[1] and s[2]) for s in evals])  
21            if correct == 2 and len(solutions) == 2:  
22                print("All solutions are correct")  
23                return  
24            else:  
25                print(f"\n{correct} solutions are correct, {len(solutions) -  
26                correct} are not correct. Expected 2 correct solutions:")  
27                print(f"{cost} solutions have minimal cost\n{ham} solutions are  
28                hamiltonian\n{start} solutions start with city 0.")  
29            return  
30        except:  
31            raise ValueError(f"invalid input: {solutions}")  
32    raise ValueError(f"unkown constraint problem: {problem}")
```

Although graphical knowledge representation help to reduce the knowledge acquisition bottleneck, the construction of graphical models is still a manual process [14].

*Learning Knowledge Bases.* Learning knowledge bases (KBs) from examples is a key research area in artificial intelligence which focuses on an automated acquisition of structured representations of domain knowledge. One related aspect is the learning of CSPs from examples. Learning CSPs involves inferring constraints based on positive and negative examples, often using constraint acquisition techniques that query an oracle or analyze example datasets to build consistent constraint models [15]. Such approaches have shown to be effective in domains such as scheduling, configuration, and automated diagnosis.

With the goal to optimize search performance, *learning search heuristics* for CSP solvers has become a relevant topic. The efficiency of CSP solvers largely depends on heuristics for variable and value selection during search. Machine learning methods have been developed to predict or adapt these heuristics by extracting informative features from problem instances thus leveraging existing solving experience from the past [16]. Portfolio-based solver frameworks, such as SATzilla, use learned models to dynamically select or combine heuristics and algorithms. With this, they significantly improve solver performance across heterogeneous CSP instances [17].

*LLM-based Knowledge Generation.* With the rise of large language models (LLMs), automated LLM-based knowledge generation has the potential to become a key approach in the context of knowledge

**Listing 2:** Generated constraint model for the TSP problem. The specific task is given as it was given in the prompt (Lines 5 to 11). LLM created the code. It took 5 iterations with 3 syntactic errors and 1 verification error (see chat in the Addendum). Part 1/2.

```

1  ‘‘python
2  from pychoco.model import Model
3  from test_tool import verify_solutions
4
5  C = 4
6  D = [
7      [0, 10, 15, 20],
8      [10, 0, 35, 25],
9      [15, 35, 0, 30],
10     [20, 25, 30, 0]
11 ]
12
13 max_cost = max(max(row) for row in D)
14
15 def get_arc_cost_expr(model, from_var, to_var, arc_cost_var):
16     # add table constraint for (from_var, to_var, arc_cost_var)
17     tuples = []
18     for from_city in range(C):
19         for to_city in range(C):
20             tuples.append([from_city, to_city, D[from_city][to_city]])
21     model.table([from_var, to_var, arc_cost_var], tuples).post()
22
23 model = Model()
24 tour = [model.intvar(0, C-1, f"tour_{i}") for i in range(C)]
25 model.all_different(tour).post()
26 model.arithm(tour[0], "=", 0).post()

```

engineering processes [12, 18, 19, 20, 21]. Suchanek and Luu [22] argue, as we do, for combining LLMs with data-driven and knowledge-based methods to leverage the strengths of both. A key strategy in our work is grounding LLM outputs through test cases that verify the intended semantics of generated CSPs. Nayak and Timmapathini [23] follow a similar path by using LLM to identify objects and relations in structured data. A major commonality with the work presented in this paper is the idea of applying test cases as background knowledge, which helps to assure the correctness of the generated knowledge bases. Our approach advances the state of the art by providing generalized test routines which help to assure global consistency.

Ahmed and Choudhury [24] explore LLMs for the generation of optimization problems, aiming to simplify this task for domain experts by reducing the need for deep mathematical expertise. Using datasets of problem definitions and solutions, they demonstrate how LLMs can translate textual descriptions into formal optimization models. Unlike their human-in-the-loop fine-tuning approach, we focus on automated feedback using both syntactic validation (e.g., checking for compile errors raising exceptions) and semantic validation via test cases.

LLMs have also been applied in strategic reasoning contexts, including economic simulations and game theory [25]. A related method is the "Program of Thoughts" approach [26], which uses an LLM and a Python interpreter to solve numerical problems, though it lacks discussion on verifying the resulting code. Logical reasoning applications of LLMs include both fine-tuning models for specific tasks [27] and automated prompt engineering [28]. Pan et al. [29], for instance, improve code generation by using errors from constraint and SAT solvers as prompt feedback. Our work builds on this idea by

incorporating test cases to further enhance CSP output on the semantic level. Other work, such as Acher et al. [30], has explored the usage of diverse prompts to generate configuration knowledge, including support for multiple programming languages. However, these efforts rely on manual execution without iterative correction mechanisms.

Finally, the automatic creation of CSPs (as proposed in this paper) can be seen as a specialized form of ontology construction, where concept hierarchies are generated using LLM prompts tailored to specific knowledge queries. For instance, Funk et al. [31] show how prompts like “What are the most important subcategories of category A?” can guide the creation of structured concept trees.

### 3. Approach for Test-Driven Generation of Constraint Satisfaction Problems

Main aspects of our methodology are the description of the input, the prompt, and the overall architecture. The verification methods is described in Section 4.

#### 3.1. Input Description

We start with a textual description of a problem at hand, for which a constraint model shall be generated.

- This textual description can be the name of a commonly known task such as the N-queen problem, Magic Square, Map Coloring, or the Traveling Salesperson Problem (TSP) [10]. Using commonly known means, here, a description of the problem exists in the Internet and, hence, was used when the LLM had been trained.
- Or, the textual description is a natural language description of restrictions (e.g. *There are five houses. The Englishman lives in the red house. ...*) that must be fulfilled such as given for a Zebra Puzzle<sup>7</sup>.
- Or, a general description of an abstract problem with a concrete task instance is given.

In this paper, we concentrate on names as input, thus, expecting a commonly known meaning of a constraint satisfaction problem.

In our current implementation, the generated models are expressed in Python using the PyChoco API [32, 33], which defines the formal language available for constraint modeling. As a result, the LLM must produce syntactically correct Python code that makes use of Choco’s modeling primitives. This choice ensures that solutions can be verified automatically but also means that the approach is bound to the expressive power and syntax of this solver. While we mention natural language descriptions and abstract problem definitions as possible inputs, in practice the current approach is most effective for well-known benchmark problems with concise descriptions (such as N-Queens or TSP). Larger-scale configuration problems are beyond the current scope. We therefore position our approach as particularly suitable for small to medium-sized CSPs or as a prototyping tool, while the scalability to industrial-size configuration tasks remains open for future work.

#### 3.2. Prompting

We provide a prompt for a given problem in the usual manner where we describe the general task of the LLM as a *system – prompt* (see Fig. 1 “system:”)<sup>8</sup>, and one example of the code to be generated (one shot learning). Important for the verification is here that the representation of the solution is specified in the prompt (here a Python list) and the demand to call the method `verify_solutions` with the problem (here the name) parameterized. Note that we do not mention a specific tool in the prompt that would be needed to let the LLM generate a tool call, we simply say: *write python code*.

<sup>7</sup>[https://en.wikipedia.org/wiki/Zebra\\_Puzzle](https://en.wikipedia.org/wiki/Zebra_Puzzle) (accessed: May 2025)

<sup>8</sup>For clarity, for each figure and listing, we indicate the originator of the code.

**Listing 3:** Generated constraint model for the TSP problem. LLM created the code and the iterator prints the answer of the verification method. Part 2/2.

```
1 arc_costs = [model.intvar(0, max_cost, f"arc_cost_{i}") for i in range(C)]
2 for i in range(C):
3     get_arc_cost_expr(model, tour[i], tour[(i+1)%C], arc_costs[i])
4
5 cost = model.intvar(0, 1000, "cost")
6 model.sum(arc_costs, "=", cost).post()
7
8 # First, find the minimal cost
9 solver = model.get_solver()
10 min_cost = None
11 for s in solver.find_all_solutions():
12     c = s.get_int_val(cost)
13     if min_cost is None or c < min_cost:
14         min_cost = c
15
16 # Now, collect all minimal cost tours
17 model = Model()
18 tour = [model.intvar(0, C-1, f"tour_{i}") for i in range(C)]
19 model.all_different(tour).post()
20 model.arithm(tour[0], "=", 0).post()
21 arc_costs = [model.intvar(0, max_cost, f"arc_cost_{i}") for i in range(C)]
22 for i in range(C):
23     get_arc_cost_expr(model, tour[i], tour[(i+1)%C], arc_costs[i])
24 cost = model.intvar(0, 1000, "cost")
25 model.sum(arc_costs, "=", cost).post()
26 model.arithm(cost, "=", min_cost).post()
27
28 solver = model.get_solver()
29 solutions = []
30 for s in solver.find_all_solutions():
31     t = [s.get_int_val(tour[i]) for i in range(C)]
32     solutions.append(t)
33
34 verify_solutions(solutions, "tsp")
35 ‘‘‘
```

---

user:

```
1 All solutions are correct
```

We do so because we would not rely on a specific keyword such as *Python\_REPL* which shall trigger LANGCHAIN (see Section 5.1).

A specific task is given through the definition of a concrete constraint satisfaction problem (see Fig. 1 "user:")

### 3.3. Architecture

Fig. 2 shows our iterative architecture. Only the initial prompt is given by a human, then our iterator program calls the LLM, which generates eventually the code (i.e., a constraint model written with the constraint solver PyChoco). If code was generated by the LLM, it is given to the Python executor, which



**Listing 4:** First, wrong version of the LLM-generated cost handling for the TSP model. Line 6 is wrong. The PyChoco library reports the error (Line 6 to 8).

```
1 # Add cost variable
2 cost = model.intvar(0, 1000, "cost")
3 # Express the cost of the tour (cycle)
4 model.sum(
5     [
6         model.element(D, tour[i], tour[(i+1)%C])
7         for i in range(C)
8     ],
9     "=", cost
10 ).post()
11
12 # Find the minimal cost
13 solver = model.get_solver()
14 min_cost = None
15 for s in solver.find_all_solutions():
16     c = s.get_int_val(cost)
17     if min_cost is None or c < min_cost:
18         min_cost = c
19
20 ##### Lines skipped #####
21
22 verify_solutions(solutions, "tsp")
23 ‘‘‘
```

---

user:

```
1 Traceback (most recent call last):
2   File "/tmp/tmpar_b5jq.py", line 22, in <module>
3     [
4     File "/tmp/tmpar_b5jq.py", line 23, in <listcomp>
5     model.element(D, tour[i], tour[(i+1)%C])
6     File "/home/chbaeh/miniconda3/envs/llmconst/lib/python3.10/site-packages/
7       pychoco/constraints/int_constraint_factory.py", line 245, in element
8       if len(table) == 0:
9 TypeError: object of type 'IntVar' has no len()
```

simply executes the code as usual with a Python interpreter. The constraint solver has to be imported through an import statement through the code which the LLM generates. The result of the interpreter, let it be a syntactic error indication (exceptions) (e.g., because the API of PyChoco is not used properly) or a semantic error, i.e., the verification method computes an error, is given back in the prompt to the LLM. Thus, the prompt, or better the context of the prompt, is increased with each iteration. If the LLM provides no code, the iteration stops.

Similar frameworks (sometime also called "reasoning frameworks") are LANGCHAIN [34], OpenAI Reasoning [35], Chain of Thought [36], ReAct [37], or those as in [38] based on the Model Context Protocol (MCP) [9]. However, we implemented our own loop (see Section 5.1).

## 4. Verifying Constraint Models

For verifying generated constraint models, we define tests or verification methods for the problem at hand, i.e., a problem-specific verification test that checks a solution for consistency or/and optimality.



```

user:
2 solutions are correct, 4 are not correct.
Expected 2 correct solutions:
2 solutions have minimal cost
6 solutions are hamiltonian
6 solutions start with city 0.

```

**Figure 3:** Verification responds that provides concrete hints for the LLM.

**Listing 5:** Central loop for iteratively (Line 2, `for`) calling the LLM (Line 5, `ask_openai`) and the tool (Line 10, `python_executer`), the context is increased (Lines 7 and 12, `append`). Line 13 relates to Line 20 in Listing 1, hence, checks if the solution was verified. (Human created.)

```

1 run_loop(messages: list[MessageType], n_iters: int = MAX_ITERS):
2     for i in range(n_iters):
3         print_message(messages[-1])
4         print(f"{i}: call llm")
5         ret_message = ask_openai(messages)
6         print_message(ret_message)
7         messages.append(ret_message)
8         code = extract_python_code(ret_message.get("content", ""))
9         if code is not None:
10            tool_output = python_executer.run(code)
11            tool_message = user_message(tool_output)
12            messages.append(tool_message)
13            if "All solutions are correct" in tool_output:
14                print_message(messages[-1])
15                return messages
16        else:
17            return messages
18    return messages

```

To ensure the accuracy of these tests, they are manually created by a human software engineer, as is usual for software implementation. However, perhaps with the support of source code generators such as Copilot<sup>9</sup>, however, with the explicit confirmation of the human engineer<sup>10</sup>. Hence, here, we explicitly exclude a pure generation approach to ensure correctness, avoiding confabulation of the LLM. As pointed out in Section 3.2 the call to the verification method is parameterized with the problem name and has to be included in the generated code by the LLM, which really was the case in all of our tests (see Section 5.2). The method `verify_solutions` (see Fig. 1) verifies each problem separately to verify the solutions created by the program generated by the LLM.

In any case, of course, the concrete task instance, as it has to be verified, has to be coded into the verification method. However, the solution, since it is also a known representation specified in the prompt (see Section 3.2), is checked for consistency with the restrictions that hold for each problem. For example, the positions of the created queens solutions must not threaten each other. Additionally to consistency, in the TSP model, not only the Hamiltonian path (each city is in the solution) is of interest but the solution should also be minimal. Here, in one chat (see the Addendum), the LLM generated first a not correct solution which was not minimal. Our verification method verbosely provides an explanation of the not correct results and includes the responses shown in Fig. 3 in the context, which leads to an improved version that adds the notion of cost to the chat for the first time (see Listing 4) -

<sup>9</sup><https://copilot.microsoft.com/> (accessed: May 2025)

<sup>10</sup>We strictly follow our regime P.b.H.W.b.I.T.E.b.H, i.e., Prompted by Human, Written by IT, Evaluated by Human.

the responses triggered the LLM to consider cost when solving the TSP. This means that for the TSP, the name "TSP" given in the initial prompt was not sufficient to ensure that an optimality criterion (the cost of the paths) is needed for solving the TSP, not only a consistency one. Hence, this is an example for changing or refining requirements during the chat and an example for a reasoning process. However, the first occurrence of cost handling in Listing 4 was wrong and later corrected in Listing 2.

The role of the user in this process is to provide problem-specific correctness criteria in the form of verification routines. These routines do not encode the complete CSP model, but rather provide a minimal set of semantic checks that any valid solution must satisfy (e.g., non-attacking queens in N-Queens, or route optimality in TSP). In practice, these checks are often simpler to formulate than an entire constraint model, since they can be expressed as direct computational tests over candidate solutions. The completeness of the verification depends on how precisely the user captures the essential constraints of the domain: if all relevant conditions are checked, then a generated model that passes verification can be regarded as correct. If only partial conditions are provided, the generated model will be correct only with respect to those conditions. Thus, the user's effort is shifted from fully modeling the problem to defining targeted, testable requirements that guide the LLM toward producing consistent CSPs.

Hence, in total, the verification method needs:

- concrete constraints of a constraint problem,
- a test method for checking these restrictions of the CSP in general,
- explanations as output if errors occur for guiding the LLM to a solution.

## 5. Implementation and Results

### 5.1. Implementation

The implementation utilizes Python, leverages ChatGPT-4.1 and is available in a repository at Zenodo<sup>11</sup>. As a constraint solver, we used PyChoco as a Python API for the constraint solver Choco [32, 33].

We do not include tool calls in the prompt, but only the demand to create Python code. Hence, we do not depend on the need that the LLM generates strings such as "Python\_REPL". Furthermore, we could simpler manipulate the output of the Python interpreter and, hence, compress the context for saving context tokens (not further discussed here). The Listing 5 shows the implementation of the core loop with prompt increase, call to LLM, extracting the code, and executing it. The verification is called implicitly in the LLM-generated code.

### 5.2. Result

As an example, Listing 2 and Listing 3 show the LLM-generated code for the TSP problem and the final verification responds. One can see how the method `verify_solutions` is included in the code to start verification. The code is a constraint model, because constraint variables with appropriate variable domains are defined (lines 24, 32), constraints are defined (here by *post* in lines 21, 25, 26, 33), and finally the constraint solver is called (line 36) to generate solutions (line 36) which are consistent with the posted constraints.

For each of the selected concrete problems N-queen, Magic Square, Map Coloring, and TSP, we could generate a constraint model and verify its correctness. For Map Coloring and Magic Square an only one-step iteration was needed, i.e., the LLM could directly generate the solution code which was syntactically correct, the API was used correctly, and the semantic could correctly being verified through our verification methods. The N-Queen solution ( $N = 8$ ) was sometimes found directly and, in other cases, some iterations were needed due to a misuse of the API (error messages from Python execution were included in the chat). Due to the indeterminism of the LLM, the iteration numbers

---

<sup>11</sup><https://doi.org/10.5281/zenodo.17132331>

Success	Iterations	Exceptions	Validation Errors
True	6	4	1
True	5	3	1
True	4	1	2
True	4	2	1
True	8	5	2
True	8	6	1
True	3	1	1
True	6	4	1
True	4	2	1
True	4	2	1
True	8	4	3
True	9	6	2
True	5	1	3
True	7	5	1
True	5	3	1
True	6	4	1
True	6	4	1
True	8	6	1
True	7	5	1
True	7	5	1

**Table 1**

TSP iterations of 20 runs with the same concrete task instance (see Fig. 1).

Success	Iterations	Exceptions	Validation Errors
True	1	0	0
True	2	1	0
True	2	1	0
True	2	1	0
True	2	1	0
True	1	0	0
True	11	1	9
True	8	7	0
True	19	1	17
True	9	8	0
True	2	1	0
True	1	0	0
True	4	3	0
True	1	0	0
True	3	2	0
True	2	1	0
True	8	7	0
True	1	0	0
True	7	6	0
True	1	0	0

**Table 2**

8-Queens iterations of 20 runs with the same concrete task instance (not shown).

vary. One example for the TSP needed five iterations with three API errors and one semantic error (see Listing 2).

Table 1 shows the result of 20 runs for generating a solution of the TSP. Exceptions are API errors or syntactic Python errors. The numbers of iterations and error differ, however, in any case a verifiable solution could be found. Similarly, Table 2 shows the runs for the N-Queens example. All chats are listed in our repository.

In a further test, we used the final generated TSP constraint model to apply it manually to larger problem instances. For small constraint sets, for example, up to 10 cities, the approach still works. However, beyond that, timeouts occur (after 60 seconds). There are three solutions, but these are found quickly because the LLM did not use PyChoco; instead, it relied on a purely combinatorial approach with Python. Importantly, in these three solutions the LLM never used the PyChoco API (such as `setObjective(Model.MINIMIZE, cost)`) to find optimal solutions; it only generated all the solutions and then filtered them afterwards. This approach is feasible only for very small problem instances.

## 6. Discussion

The iterative approach that we introduce in this paper has the following characteristics:

- A system prompt defining a general task and a user prompt for defining a concrete instance of the task.
- Demanding to use Python as a programming language and to use a specific constraint solver (here PyChoco).
- One-shot learning by giving one example for a code which shall be created.
- Defining that the code shall produce a certain representation (here a list of solutions) and, to call with this list as parameter a specific function for verifying the generated code.
- Defining that an iteration will take place.
- Manually coding a verification method for a specific problem.
- Implementing a simple loop that calls the LLM to generate code, calls Python to execute it, and successively appends the Python output to the context (prompt).
- Help the LLM through specific verification hints that refine the requirements (here, the problem definition) given in the initial prompt.
- For all problem instances a verifiable code could be generated.
- For this generation of constraint models, the iterative approach is important, only for simple tasks one iteration step was sufficient.
- No use of reasoning frameworks because of the simplicity of looping through tools (here constraint solver) that leads to full control over input and output.
- Using our current approach, we aim to facilitate the process of knowledge acquisition to achieve consensus on the accuracy of the knowledge base. Our automation not only supports the initial development of the knowledge base but also accommodates updates and modifications as the knowledge evolves. Additionally, knowledge engineers and domain experts can focus on designing and performing verification tests.

As new LLMs are constantly being released and their non-deterministic nature means that the numbers presented in the tables represent only a snapshot, future advancements in even more powerful models are likely to yield different results.

The verification method we present in this paper ensures complete (global) consistency and, in the case of the TSP, optimality of the resulting constraint model. This is feasible because the verification method has full access to the entire set of constraints that must be satisfied. In other cases, typically known from test-driven development of software artifacts, only partial aspects of the software can be formalized, tested, and verified. In CSPs, full access to the entire set of constraints is due to the formal character of the given constraints which can easily be implemented as verification tests. However, the formalization of the constraint model used to compute solutions is generated by the LLM and is often more challenging than simply writing verification tests.

A crucial point with solvers is that they implement specific efficient algorithms that are general and work on descriptive models. However, those algorithms have to be used by the API of the solver library. In our experiments, the LLM does not use these algorithms, especially *find\_all\_optimal\_solutions*

but generates an algorithm for the brute-force method. Hence, specific prompts for leveraging these APIs should be developed to make use of these efficient algorithms. A related aspect is the generation or better exploration of new efficient algorithms for constraint solving similar to exploring protein structures.

## 7. Conclusion

The paper shows that correct constraint models can be generated with the help of an LLM if one combines their verification with constraint solvers in an iterative loop. Thus, it demonstrates that the fundamental approach of combining the generative capabilities of LLMs with the precision of constraint solvers and programming language interpreters is both feasible and poised to transform the future development of constraint models. Our next steps will be to use textual descriptions of constraint problems instead of known names.

## Acknowledgments

This project was funded through the Federal Ministry for Economic Affairs and Climate Action (Bundesministerium für Wirtschaft und Klimaschutz), Germany, within the “Joint project: EnOB: IntelMOD - Intelligent modernization platform based on functional cost splitting; sub-project: Infrastructure of the modernization platform.” (“Verbundvorhaben: EnOB: IntelMOD - Intelligente Modernisierungsplattform auf Basis des Funktionalen Kostensplittings; Teilprojekt: Infrastruktur der Modernisierungsplattform.”)

## Declaration on Generative AI

During the preparation of this work, the authors used the LLM models ChatGPT, DeepSeek, and Claude exclusively for translation (German to English) and for literature and research searches. The LLM models were not employed to generate core content of the paper, i.e., they were not involved in analysis, methodology, results, or conclusions. Additionally, the AI-generated code discussed in the paper constitutes research data or experimental results and is presented as evidence supporting the research findings; it was not used as a tool to write the paper itself.

## References

- [1] A. Felfernig, L. Hotz, C. Bagley, J. Tiuhonen, Knowledge-based Configuration – From Research to Business Cases, Morgan Kaufmann, 2014.
- [2] Y. Huang, H. Liu, W. K. Ng, W. Lu, B. Song, X. Li, Automating knowledge acquisition for constraint-based product configuration, *Journal of Manufacturing Technology Management* 19 (2008) 744–754. doi:10.1108/17410380810888120.
- [3] T. E. Nordlander, E. C. Freuder, R. J. Wallace, Maintaining constraint-based applications, in: 4th International Conference on Knowledge Capture, K-CAP '07, ACM, 2007, pp. 79–86. doi:10.1145/1298406.1298422.
- [4] U. Junker, Configuration, in: F. Rossi, P. van Beek, T. Walsh (Eds.), *Handbook of Constraint Programming*, Elsevier, 2006, pp. 837–873.
- [5] R. Barták, M. A. Salido, Constraint satisfaction for planning and scheduling problems, *Constraints* 16 (2011) 223–227. URL: <https://doi.org/10.1007/s10601-011-9109-4>. doi:10.1007/s10601-011-9109-4.
- [6] I. Ozkaya, A. Carleton, J. Robert, D. Schmidt, Application of Large Language Models (LLMs) in Software Engineering: Overblown Hype or Disruptive Change?, Carnegie Mellon University, Software Engineering Institute’s Insights (blog), 2023. URL: <https://doi.org/10.58012/6n1p-pw64>, accessed: 2024-May-16.

- [7] K. Michailidis, D. Tsouros, T. Guns, Constraint Modelling with LLMs Using In-Context Learning, in: P. Shaw (Ed.), 30th International Conference on Principles and Practice of Constraint Programming (CP 2024), volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2024, pp. 20:1–20:27. doi:10.4230/LIPIcs.CP.2024.20.
- [8] F. Régim, E. D. Maria, A. Bonlarron, Combining constraint programming reasoning with large language model predictions, in: Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP 2024), volume 275 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 25:1–25:16. URL: <https://arxiv.org/abs/2407.13490>. doi:10.4230/LIPIcs.CP.2024.25.
- [9] P. Anthropic, Model Context Protocol (MCP), <https://modelcontextprotocol.io>, 2025. Accessed: 2025-04-20.
- [10] L. Hotz, C. Bähnisch, S. Lubos, A. Felfernig, A. Haag, J. Twiefel, Exploiting Large Language Models for the Automated Generation of Constraint Satisfaction Problems, volume 3812 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024, pp. 91–100.
- [11] B. O’Sullivan, Automated modelling and solving in constraint programming, in: 24th AAAI Conference on Artificial Intelligence, AAAI’10, AAAI Press, 2010, pp. 1493–1497.
- [12] B. P. Allen, L. Stork, P. Groth, Knowledge Engineering Using Large Language Models, Transactions on Graph Data and Knowledge (TGDK), Special Issue on Trends in Graph Data and Knowledge 1 (2023) 3:1–3:19. doi:10.4230/TGDK.1.1.3.
- [13] D. Benavides, S. Segura, A. Ruiz-Cortes, Automated analysis of feature models 20 years later: A literature review, *Information Systems* 35 (2010) 615–636.
- [14] A. Felfernig, G. Friedrich, D. Jannach, UML as domain specific language for the construction of knowledge-based configuration systems, *International Journal on Software Engineering and Knowledge Engineering* 10 (2000) 449–469.
- [15] C. Bessiere, F. Koriche, N. Lazaar, B. O’Sullivan, Constraint acquisition, *Artificial Intelligence* 244 (2017) 315–342. doi:10.1016/j.artint.2015.08.001.
- [16] L. Xu, F. Hutter, H. H. Hoos, K. Leyton-Brown, Satzilla: Portfolio-based algorithm selection for sat, *Journal of Artificial Intelligence Research* 32 (2008) 565–606.
- [17] K. Leyton-Brown, E. Nudelman, Y. Shoham, Learning the Empirical Hardness of Optimization Problems: The Case of Combinatorial Auctions, in: 8th International Conference on Principles and Practice of Constraint Programming, Springer-Verlag, Berlin, Heidelberg, 2002, pp. 556–572.
- [18] E. C. Freuder, Conversational modeling for constraint satisfaction, AAAI Conference on Artificial Intelligence 38 (2024) 22592–22597. URL: <https://ojs.aaai.org/index.php/AAAI/article/view/30268>. doi:10.1609/aaai.v38i20.30268.
- [19] F. Petroni, T. Rocktäschel, P. P. Lewis, A. Bakhtin, Y. Wu, A. Miller, S. Riedel, Language models as knowledge bases?, 2019. URL: <https://arxiv.org/pdf/1909.01066>.
- [20] Y. Ding, A. Poudel, Q. Zeng, T. Weninger, B. Veeramani, S. Bhattacharya, EntGPT: Linking Generative Large Language Models with Knowledge Bases, 2024. URL: <https://arxiv.org/abs/2402.06738>.
- [21] S. Razniewski, A. Yates, N. Kassner, G. Weikum, Language models as or for knowledge bases, 2021. URL: <https://arxiv.org/abs/2110.04888>.
- [22] F. Suchanek, Luu, Knowledge bases and language models: Complementing forces, in: 7th International Joint Conference on Rules and Reasoning, Springer-Verlag, Berlin, Heidelberg, 2023, pp. 3–15.
- [23] A. Nayak, H. P. Timmapathini, Llm2kb: Constructing knowledge bases using instruction tuned context aware large language models, 2023. URL: <https://arxiv.org/abs/2308.13207>.
- [24] T. Ahmed, S. Choudhury, LM4OPT: Unveiling the Potential of Large Language Models in Formulating Mathematical Optimization Problems, 2024. URL: <https://arxiv.org/abs/2403.01342>.
- [25] Y. Zhang, S. Mao, T. Ge, X. Wang, A. de Wynter, LLM as a mastermind: A survey of strategic reasoning with large language models, 2023. URL: <https://arxiv.org/abs/2404.01230v1>. arXiv:2404.01230.
- [26] W. Chen, X. Ma, X. Wang, W. W. Cohen, Program of thoughts prompting: Disentangling computa-

- tion from reasoning for numerical reasoning tasks, *Transactions on Machine Learning Research* (2023). URL: <https://openreview.net/forum?id=YfZ4ZPt8zd>.
- [27] P. Clark, O. Tafjord, K. Richardson, Transformers as soft reasoners over language, in: 29th International Joint Conference on Artificial Intelligence, IJCAI'20, 2021, pp. 3882–3890.
  - [28] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. H. Hsin Chi, D. Zhou, Self-consistency improves chain of thought reasoning in language models, *ArXiv* (2022). URL: <https://arxiv.org/abs/2203.11171>.
  - [29] L. Pan, A. Albalak, X. Wang, W. Y. Wang, Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning, 2023. *arXiv:2305.12295*.
  - [30] M. Acher, J. G. Duarte, J.-M. Jézéquel, On programming variability with large language model-based assistant, in: *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume A, SPLC '23*, Association for Computing Machinery, New York, NY, USA, 2023, p. 8–14. URL: <https://doi.org/10.1145/3579027.3608972>.
  - [31] M. Funk, S. Hosemann, J. C. Jung, C. Lutz, Towards ontology construction with language models, 2023. URL: <https://arxiv.org/abs/2309.09898>.
  - [32] C. Prud'homme, J.-G. Fages, Choco-solver: A Java library for constraint programming, *Journal of Open Source Software* 7 (2022) 4708. URL: <https://doi.org/10.21105/joss.04708>. doi:10.21105/joss.04708, <https://github.com/chocoteam/pychoco> (PyChoco, Python binding).
  - [33] Choco Team, PyChoco: Python binding for Choco-solver, <https://github.com/chocoteam/pychoco>, 2025.
  - [34] V. Mavroudis, LangChain v0.3, Technical Report, Preprints, 2024. URL: <https://doi.org/10.20944/preprints202411.0566.v1>. doi:10.20944/preprints202411.0566.v1, preprint, not peer-reviewed.
  - [35] OpenAI, Learning to Reason with LLMs, <https://openai.com/index/learning-to-reason-with-llms/>, 2024. Accessed: 2025-04-20.
  - [36] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, D. Zhou, Chain-of-thought prompting elicits reasoning in large language models, in: *Advances in Neural Information Processing Systems*, volume 35, Curran Associates, Inc., 2022, pp. 24824–24837. URL: <https://arxiv.org/abs/2201.11903>.
  - [37] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, Y. Cao, React: Synergizing reasoning and acting in language models, 2023. URL: <https://arxiv.org/abs/2210.03629>. *arXiv:2210.03629*.
  - [38] S. Szeider, MCP-Solver: Integrating language models with constraint programming systems, <https://arxiv.org/abs/2501.00539>, 2025. ArXiv preprint *arXiv:2501.00539*, accessed: 2025-05-20.



## Addendum

The figures 4 to 12 show one complete chat.

---

system:

You are an agent designed to solve constraint problems, you write python code to solve and verify constraint problems. Your code is run and verified by the user. If you get an error, debug your code and try again. You only use the constraint solver psychoco. You conclude your python code with a variable "solutions" and verify it with the method call "verify\_solutions(solutions, <problem\_name>", where <problem\_name> is given to you as the first word of the instructions. "solutions" should be a list of all solutions to the constraint problem. If verification fails, you correct your solution.

Use the following example as a template for solving constraint problems with psychoco:

User: ints10. find numbers one to 10.

Assistant:

```
'''python
from psychoco.model import Model
from test_tool import verify_solutions

model = Model()
x = model.intvar(1, 10, "x")

solver = model.get_solver()
solutions = [s.get_int_val(x) for s in solver.find_all_solutions()]
verify_solutions(solutions, "ints10")
'''
```

---

user:

tsp. solve the traveling salesman problem, use the following problem instance:

```
# Number of cities
C = 4

# Distance matrix
D = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
```

the solution should be a list of valid solutions, each solution being a list of integers representing the cities to be visited.

---

assistant:

```
'''python
from psychoco.model import Model
from test_tool import verify_solutions
```

**Figure 4:** Complete chat for generating a solution for the TSP problem. Part I.

```

C = 4
D = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

model = Model()

# The tour: a permutation of cities, starting at city 0 (fix first city)
tour = model.intvars([0] + [1, C-1], C, "tour") # tour[0] = 0

# All cities must be visited exactly once
model.all_different(tour).post()
model.arithm(tour[0], "=", 0).post()

# To break symmetries, fix the first city to 0
# (already done above)

solver = model.get_solver()

solutions = []
for s in solver.find_all_solutions():
    t = [s.get_int_val(tour[i]) for i in range(C)]
    # Make the tour a cycle by returning to 0
    if t[0] == 0 and len(set(t)) == C and all(0 <= city < C for city in t):
        solutions.append(t)

verify_solutions(solutions, "tsp")
'''

```

**Figure 5:** Complete chat for generating a solution for the TSP problem. Part II.

user:

Traceback (most recent call last):

```
File "/tmp/tmps5uh3m8t.py", line 15, in <module>
    tour = model.intvars([0] + [1, C-1], C, "tour") # tour[0] = 0
File "/home/chbaeh/miniconda3/envs/llmconst/lib/python3.10/site-packages/
    psychoco/variables/variable_factory.py", line 71, in intvars
    return [self.intvar(lb, ub, name) for i in range(0, size)]
TypeError: 'list' object cannot be interpreted as an integer
```

assistant:

```
'''python
from psychoco.model import Model
from test_tool import verify_solutions

C = 4
D = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

model = Model()

# The tour: a permutation of cities, starting at city 0 (fix first city)
tour = [model.intvar(0, C-1, f"tour_{i}") for i in range(C)]

# All cities must be visited exactly once
model.all_different(tour).post()
model.arithm(tour[0], "=", 0).post()

solver = model.get_solver()

solutions = []
for s in solver.find_all_solutions():
    t = [s.get_int_val(tour[i]) for i in range(C)]
    # Ensure it is a valid tour starting at 0
    if t[0] == 0 and len(set(t)) == C and all(0 <= city < C for city in t):
        solutions.append(t)

verify_solutions(solutions, "tsp")
'''
```

**Figure 6:** Complete chat for generating a solution for the TSP problem. Part III.

---

user:

2 solutions are correct, 4 are not correct. Expected 2 correct solutions:  
2 solutions have minimal cost  
6 solutions are hamiltonian  
6 solutions start with city 0.

---

assistant:

```
'''python
from pychoco.model import Model
from test_tool import verify_solutions

C = 4
D = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

model = Model()

tour = [model.intvar(0, C-1, f"tour_{i}") for i in range(C)]
model.all_different(tour).post()
model.arithm(tour[0], "=", 0).post()

# Add cost variable
cost = model.intvar(0, 1000, "cost")
# Express the cost of the tour (cycle)
model.sum(
    [
        model.element(D, tour[i], tour[(i+1)%C])
        for i in range(C)
    ],
    "=", cost
).post()
```

**Figure 7:** Complete chat for generating a solution for the TSP problem. Part IV.

```

# Find the minimal cost
solver = model.get_solver()
min_cost = None
for s in solver.find_all_solutions():
    c = s.get_int_val(cost)
    if min_cost is None or c < min_cost:
        min_cost = c

# Search again, collect only minimal cost tours
model = Model()

tour = [model.intvar(0, C-1, f"tour_{i}") for i in range(C)]
model.all_different(tour).post()
model.arithm(tour[0], "=", 0).post()
cost = model.intvar(0, 1000, "cost")
model.sum(
    [
        model.element(D, tour[i], tour[(i+1)%C])
        for i in range(C)
    ],
    "=", cost
).post()
model.arithm(cost, "=", min_cost).post()

solver = model.get_solver()
solutions = []
for s in solver.find_all_solutions():
    t = [s.get_int_val(tour[i]) for i in range(C)]
    solutions.append(t)

verify_solutions(solutions, "tsp")
'''

```

**Figure 8:** Complete chat for generating a solution for the TSP problem. Part V.

user:

Traceback (most recent call last):

```
File "/tmp/tmpar_b5jgq.py", line 22, in <module>
[
File "/tmp/tmpar_b5jgq.py", line 23, in <listcomp>
model.element(D, tour[i], tour[(i+1)%C])
File "/home/chbaeh/miniconda3/envs/llmconst/lib/python3.10/site-packages/
pychoco/constraints/int_constraint_factory.py", line 245, in element
if len(table) == 0:
TypeError: object of type 'IntVar' has no len()
```

assistant:

```
‘‘python
from pychoco.model import Model
from test_tool import verify_solutions

C = 4
D = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

model = Model()
tour = [model.intvar(0, C-1, f"tour_{i}") for i in range(C)]
model.all_different(tour).post()
model.arithm(tour[0], "=", 0).post()

# Create variables for each arc's cost
arc_costs = [model.intvar(0, max(max(row) for row in D), f"arc_cost_{i}") for i
              in range(C)]

for i in range(C):
    # arc from tour[i] to tour[(i+1)%C]
    # Use element1d to get D[from][to]
    from_var = tour[i]
    to_var = tour[(i+1)%C]
    for from_city in range(C):
        for to_city in range(C):
            # If from_var==from_city and to_var==to_city then arc_costs[i] == D[
            from_city][to_city]
            model.if_then(
                model.and_(
                    model.arithm(from_var, "=", from_city),
                    model.arithm(to_var, "=", to_city)
                ),
                model.arithm(arc_costs[i], "=", D[from_city][to_city])
            )

cost = model.intvar(0, 1000, "cost")
model.sum(arc_costs, "=", cost).post()
```

**Figure 9:** Complete chat for generating a solution for the TSP problem. Part VI.

```

# First, find the minimal cost
solver = model.get_solver()
min_cost = None
for s in solver.find_all_solutions():
    c = s.get_int_val(cost)
    if min_cost is None or c < min_cost:
        min_cost = c

# Now, collect all minimal cost tours
model = Model()
tour = [model.intvar(0, C-1, f"tour_{i}") for i in range(C)]
model.all_different(tour).post()
model.arithm(tour[0], "=", 0).post()
arc_costs = [model.intvar(0, max(max(row) for row in D), f"arc_cost_{i}") for i
              in range(C)]
for i in range(C):
    from_var = tour[i]
    to_var = tour[(i+1)%C]
    for from_city in range(C):
        for to_city in range(C):
            model.if_then(
                model.and_(
                    model.arithm(from_var, "=", from_city),
                    model.arithm(to_var, "=", to_city)
                ),
                model.arithm(arc_costs[i], "=", D[from_city][to_city])
            )
cost = model.intvar(0, 1000, "cost")
model.sum(arc_costs, "=", cost).post()
model.arithm(cost, "=", min_cost).post()

```

**Figure 10:** Complete chat for generating a solution for the TSP problem. Part VII.



```

solver = model.get_solver()
solutions = []
for s in solver.find_all_solutions():
    t = [s.get_int_val(tour[i]) for i in range(C)]
    solutions.append(t)

verify_solutions(solutions, "tsp")
'''

```

---

user:

```

Traceback (most recent call last):
  File "/tmp/tmp8jxdh1xk.py", line 29, in <module>
    model.if_then(
AttributeError: 'Model' object has no attribute 'if_then'

```

---

assistant:

```

'''python
from pychoco.model import Model
from test_tool import verify_solutions

C = 4
D = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

max_cost = max(max(row) for row in D)

def get_arc_cost_expr(model, from_var, to_var, arc_cost_var):
    # add table constraint for (from_var, to_var, arc_cost_var)
    tuples = []
    for from_city in range(C):
        for to_city in range(C):
            tuples.append([from_city, to_city, D[from_city][to_city]])
    model.table([from_var, to_var, arc_cost_var], tuples).post()

model = Model()
tour = [model.intvar(0, C-1, f"tour_{i}") for i in range(C)]
model.all_different(tour).post()
model.arithm(tour[0], "=", 0).post()

arc_costs = [model.intvar(0, max_cost, f"arc_cost_{i}") for i in range(C)]
for i in range(C):
    get_arc_cost_expr(model, tour[i], tour[(i+1)%C], arc_costs[i])

cost = model.intvar(0, 1000, "cost")
model.sum(arc_costs, "=", cost).post()

```

**Figure 11:** Complete chat for generating a solution for the TSP problem. Part VIII.

```

# First, find the minimal cost
solver = model.get_solver()
min_cost = None
for s in solver.find_all_solutions():
    c = s.get_int_val(cost)
    if min_cost is None or c < min_cost:
        min_cost = c

# Now, collect all minimal cost tours
model = Model()
tour = [model.intvar(0, C-1, f"tour_{i}") for i in range(C)]
model.all_different(tour).post()
model.arithm(tour[0], "=", 0).post()
arc_costs = [model.intvar(0, max_cost, f"arc_cost_{i}") for i in range(C)]
for i in range(C):
    get_arc_cost_expr(model, tour[i], tour[(i+1)%C], arc_costs[i])
cost = model.intvar(0, 1000, "cost")
model.sum(arc_costs, "=", cost).post()
model.arithm(cost, "=", min_cost).post()

solver = model.get_solver()
solutions = []
for s in solver.find_all_solutions():
    t = [s.get_int_val(tour[i]) for i in range(C)]
    solutions.append(t)

verify_solutions(solutions, "tsp")
'''

```

---

user:

All solutions are correct

**Figure 12:** Complete chat for generating a solution for the TSP problem. Part IX.