Large Language Models Can Solve Real-World Planning Rigorously with Formal Verification Tools

Anonymous ACL submission

Abstract

Despite their recent advancements, Large Language Models (LLMs) still struggle to directly generate correct plans for complex multi-constraint planning problems, even with self-verification and self-critique. For example, a U.S. domestic travel planning benchmark TravelPlanner was proposed in Xie et al. (2024), where the best LLM OpenAI o1-preview can only find travel plans that satisfy user requirements with a 10% success rate given all needed information. In this work, we tackle this difficult problem by proposing an LLM-based planning framework that formalizes and solves complex multi-constraint planning problems as constrained satisfiability problems, which are further consumed by sound and complete satisfiability solvers. We start with TravelPlanner as the primary use case and achieve a success rate of 93.9%. We demonstrate our framework's robustness by showing its effectiveness in diverse paraphrased prompts. More importantly, our framework has strong zero-shot generalizability: It can successfully handle unseen constraints in a completely unseen international travel dataset we created, and it can even generalize well to new domains such as routing and task allocation problems in a zero-shot manner. Moreover, when user input queries are infeasible, our framework can identify the unsatisfiable core, provide failure reasons, and offers personalized modification suggestions to users according to diverse human preferences. We show that our framework can modify and solve for an average of 81.6% and 91.7% unsatisfiable queries from two datasets and prove with ablations that all key components of our framework are effective and necessary.

1 Introduction

Recent work has demonstrated that large language models (LLMs) (Brown et al., 2020; Ouyang et al., 2022; Achiam et al., 2023), with abundant world knowledge, abilities to collect information via tools, and capabilities of reasoning, have significant potential in solving planning problems (Huang et al., 2022a; Ahn et al., 2022; Yao et al., 2022; Huang et al., 2022b; Song et al., 2023). However, modern LLMs are not well-suited for directly solving highly complex combinatorial optimization problems with multiple levels of constraints. This is because LLMs generate responses based on token probabilities derived from their training data and do not inherently possess the ability to perform rigorous logical or mathematical reasoning. To investigate the performance of LLMs on complex realistic multi-constraint problems, Xie et al. (2024) proposed a U.S. domestic travel planning benchmark, TravelPlanner, and showed that LLMs are not capable of handling this task and the best LLM at the time, GPT-4, only achieves a 0.6% success rate. We test the strongest model of today, OpenAI o1-preview (o1p), with TravelPlanner and observed a pass rate of 10.0% even with access to pre-collected information. LLM-Modulo Framework (Kambhampati et al.), a recent work that combines LLMs with external critics, verifiers, and humans, raises the pass rate to 20% with GPT-4-Turbo and 65% with o1-preview, which is the best performance on TravelPlanner as of now.

To tackle multi-constraint problems like travel planning, an alternative way is through constraint-based planning to formalize the problem as a constraint satisfaction problem (CSP) (Dechter, 2003; Lozano-Pérez and Kaelbling, 2014), including boolean satisfiability problem (SAT) (Kautz and Selman, 1999; Rintanen, 2012) and satisfiability modulo theory (SMT) (Barrett et al., 2010; De Moura and Bjørner, 2011; Dantam et al., 2016), and solve it with existing algorithm-based solvers (Dutertre and De Moura, 2006; De Moura and Bjørner, 2008; Barrett et al., 2011). However, algorithm-based solvers usually have steep learning curves. As natural language queries have no fixed format, planners need to extract key information

from input queries accurately to model the problem. Crucially, even if the extracted information is correct, users must still modify inputs and query the tools repeatedly if their inputs are unsatisfiable.

In short, LLM-based and algorithm-based planning methods have complementary strengths: LLMs excel at parsing human input and interactions but struggle to rigorously solve complex planning problems with multiple constraints. In contrast, algorithm-based solvers are sound and complete when solving multi-constraint satisfiability problems but are incapable of handling dynamic, general, and sometimes ambiguous natural language requirements. Can we design a framework that combines the merits of both paradigms and enables a strong, rigorous, and yet user-friendly planning experience for human users?

Motivated by this, in this paper, we propose a travel-planning framework that enables LLMs to process human queries and generate code to automatically utilize algorithm-based solvers, *e.g.*, the SMT solver, to formally formulate, solve, and reason over the planning problem. Specifically, we take travel planning as the primary use case and provide LLMs with instruction steps and corresponding codes for using SMT solver to solve the example travel planning problems. We find that, with only three examples in the prompt, the LLM can effectively learn the pattern and generalize to new input queries. Since a solver is called to solve the problem with all the constraints encoded, our method is guaranteed to generate a plan if it exists.

Notably, our method offers following features, including and beyond aforementioned advantages.

• **Superior Planning Success Rate.** We evaluate our framework over different LLMs, GPT-4, Claude 3 Opus, and Mistral-large, and show that our framework achieves the best final pass rates of 93.3% and 93.9% on TravelPlanner validation and test sets. Our framework significantly outperforms the best tool-use framework on TravelPlanner, LLM Modulo, which achieves pass rates of 20% and 65% with GPT-4-Turbo and o1-preview.

• **Feedback Interactions on Unsolvable Cases.** If the input query is not satisfiable, our framework utilizes SMT solvers to identify the exact unsatisfiable constraints, analyzes the unsatisfiable reasons, and proposes suggestions to modify the query until it becomes satisfiable. In addition to a fully autonomous mode of offering default modification suggestions by itself, LLM can even interactively communicate with humans to incorporate their unique preferences. Our experiments show that our feedback mechanism can turn an average of 81.6% and 91.7% insolvable problems from two datasets into solvable ones under different user reaction patterns to our model's suggestions.

• **Strong Zero-Shot Generalizability to Unseen Constraints and Tasks.** Although the proposed planning system is demonstrated with only travel planning examples with a fixed set of constraints, we found that it is readily generalizable to unseen constraints and even unseen planning tasks. Remarkably, we further introduce four completely new domains that involve combinatorial tasks: Block Picking, Task Allocation, Travelling Salesman Problem, and Warehouse. On all new scenarios, our framework achieves an average of 89.0% optimal rates for four new domains in a zero-shot manner, as discussed in Section 5.2.3.

• **High Prompt Robustness.** To address concerns that the strong planning capabilities of our system might result from extensive prompt engineering, we conduct experiments to evaluate its robustness to varied paraphrased prompts. Our results show that the system effectively handles a wide range of prompt formulations, thus is not overly reliant on specific prompt designs. This highlights the flexibility and robustness of our method, establishing it as a powerful and adaptable planning paradigm.

2 Related Work

LLM Planning. LLMs have shown significant intelligence in reasoning (Wei et al., 2022; Kojima et al., 2022; Yao et al., 2022) and tool-use (Qin et al., 2023; Schick et al., 2024), offering the potential of promising planning capability. Previous works tackle planning problems with various ways (Huang et al., 2024): 1) decomposing the task into sub-tasks (Wei et al., 2022; Yao et al., 2022; Shen et al., 2024); 2) generating multiple plans and selecting the optimal one (Wang et al., 2022; Yao et al., 2024; Zhao et al., 2024; Besta et al., 2024; Hao et al., 2023); 3) reflecting on experiences and refining plan based on feedback (Shinn et al., 2024; Madaan et al., 2024; Chen et al., 2023b); 4) formalizing tasks and aiding with external planner (Liu et al., 2023; Guan et al., 2023; Chen et al., 2023a). While these planning algorithms have shown promising results, their planning scenarios are limited to simple tasks with a limited number of constraints. Xie et al. (2024) proposes a realistic complex travel planning benchmark and tests on

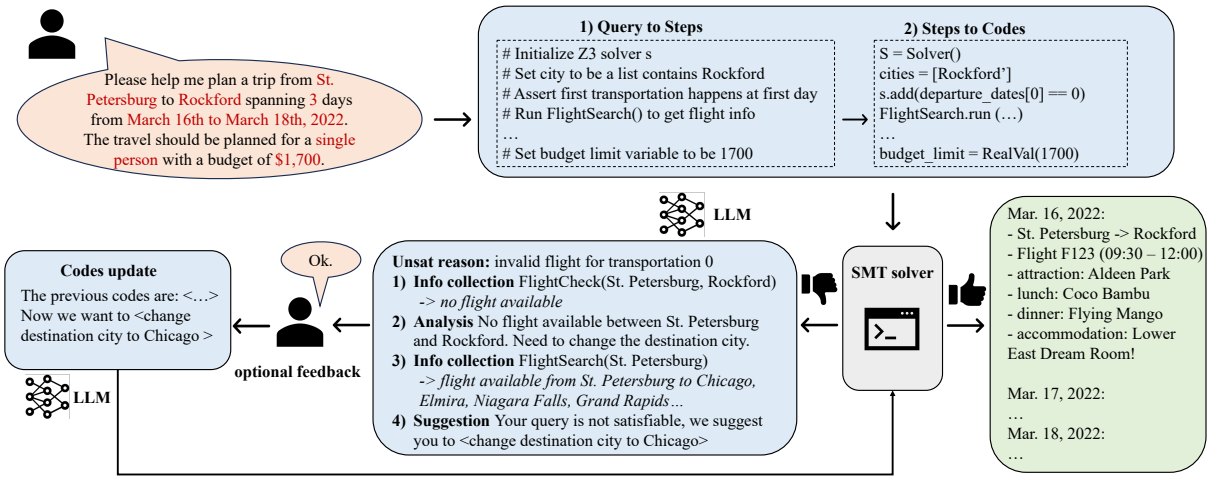


Figure 1: An overview of the framework. The blue region represents LLM. Given a natural language query, LLM 1) generates steps to formulate it as an SMT problem, 2) generates corresponding codes that encode the problem and call the solver. If the solver is not able to find the solution, LLM receives unsatisfiable reasons from the solver, collects information, analyzes the current situation, and offers suggestions to modify the query interactively. LLM then updates the code based on suggestions and calls the solver again to find a feasible plan.

various LLM planning methods to show that LLMs are not capable of handling multi-constraint tasks.

Algorithm-based Planning. Another way to tackle travel planning is through algorithm-based planning such as heuristic search (Hoffmann and Nebel, 2001; Helmert, 2006; Vidal, 2014) and constraint-based methods (Kautz and Selman, 1999; Rintanen, 2012, 2014; Lozano-Pérez and Kaelbling, 2014; Dantam et al., 2016). However, these methods can not generalize to diverse natural language inputs and may not guarantee to find the plan. Our framework enables LLM to utilize constraint-based planning methods by formalizing diverse human queries into an SMT problem and solving with sound and complete SMT solvers.

LLM Tool-use. Tool-using allows LLMs to utilize powerful external tools to increase reliability. Recent works explore how LLMs could utilize external tools such as search engines, operating environments, and code generators (Press et al., 2022; Yao et al., 2022; Schick et al., 2024; Liang et al., 2023; Singh et al., 2023; Peng et al., 2023; Song et al., 2023; Huang et al., 2022b) to provide feedback or extra information. In our framework, LLMs generate codes to formulate the travel planning problem as an SMT problem and call the SMT solver. This overcomes LLM's failure to consider all constraints by encoding and solving all constraints rigorously.

LLM Prompt Design. Prompt design is critical for LLM-based agents to improve the performance of black-box LLMs. While many works automatically evolve prompts through iterations (Wang et al., 2023; Fernando et al., 2023; Chen et al., 2024), in-context learning that includes example input-output pairs in prompts is a cheaper, widely-adopted, and reliable way (Brown et al., 2020; Wei et al., 2022; Liang et al., 2023). Although designing in-context examples requires task-specific efforts, most existing methods proposing to solve complex planning problems require different forms of task-specific efforts (Liu et al., 2023; Li et al., 2023; Gundawar et al., 2024). While our work requires prompt design, these designs are offline and not needed for end users. More importantly, we have shown that the performance of our framework is not sensitive to the specific wording of prompts (Section 5.2.1). Meanwhile, our framework generalizes and achieves great performance for other constraints and even in other tasks without the specific design of prompts (Sections 5.2.2 and 5.2.3).

3 Approach

3.1 Problem Formulation

In the framework design, we primarily focus on the specific travel planning task with a set of predefined constraints. However, we will show that the same prompts and workflow can readily be applied to new constraints and other planning scenarios.

Our travel planning problem is formulated as follows. Given a natural language description of humans' constraints $\mathcal{C}$ of a travel plan, the system should output a plan that satisfies $\mathcal{C}$. The travel starts from city o, travels k destination cities, and returns to o. The travel spans n days. The travel takes $k+1$ transportation methods for $k+1$ travels from

city to city. The travelers visit x attractions, dine in y restaurants, and live in accommodations for $n-1$ nights. By default, we set $x = n$, $y = 3n$. However, this is not a fixed requirement. Users could specify their unique requirements by adding descriptions in prompts, for example, *"Number of attractions to visit per day is 2"*. Table 5 summarizes the constraints $\mathcal{C}$ for two datasets we used. The output plan should satisfy $\mathcal{C}$ and specify the city to visit, transportation method, attraction, restaurant, and accommodation for each day. See Appendix C for example input query and output plan.

3.2 Framework Overview

As shown in Fig. 1, if the planning problem comes with valid solutions, our framework solves the problem in four steps. First, an LLM is prompted to parse the user request and output a set of steps to convert the user descriptions to a formal planning problem. Second, the LLM is prompted to further convert the steps to code. Third, the framework calls an external formal solver, in our case the SMT solver, to execute the code, whose output is then parsed by the LLM into natural language outputs.

However, there are cases where the user queries do not bear valid solutions, *e.g.*, the users request to stay in five-star hotels but the specified budget is too low, where the aforementioned steps would fail to generate a valid plan. In these cases, the LLM is prompted to reason about the situations to give suggestions to modify the constraints, *e.g.*, increasing the budget. If users accept the suggestions, the modified planning problem is formed and sent to the solver; otherwise, new suggestions will be made until an agreement is reached or timeout.

Section 3.3 details the three steps to solve satisfiable planning problems; Section 3.4 describes our approach to repairing unsatisfiable plans iteratively. All the prompts are listed in Appendix G.

3.3 Satisfiable Plan Solving

3.3.1 Query-Step Generation

Query-Step generation involves transforming natural language queries into a sequence of executable steps, expressed in natural language, to formulate the constraints. For example, to specify the *"travel spanning 3 days"* constraint in the query in Fig. 1, an example sequence of steps is:

```
1. Set 't_dates' variables for 2 transportation between cities
2. Assert first transportation happens at first day (day 0),
   and last happens at last day (day 2)
```

For the query *"travel 2 destination cities in 5 days"*, an example sequence of steps is:

Example in prompt:

```
# Assert first transportation happens at first day (day 0),
# last happens at last day (day 2)
s.add(And(t_dates[0] == 0, t_dates[1] == 2))
```

Input step:

```
# Assert first transportation happens at first day (day 0),
# last happens at last day (day 4),
# and second could happen at any day in between
```

Response code:

```
s.add(And(t_dates[0] == 0, t_dates[2] == 4))
s.add(And(t_dates[1] > t_dates[0], t_dates[1] < t_dates[2]))
```

Figure 2: Step to Code translation example.

```
1. Set 't_dates' variables for 3 transportation between cities
2. Assert first transportation happens at first day (day 0),
   last happens at last day (day 4),
   and second could happen at any day in between
```

We teach the LLM to perform such generation by providing three human-crafted examples, each containing a natural language query and the corresponding steps. Since the steps to formulate different constraints are different, the steps in each example are broken into nine sections, each focusing on the constraints about one particular subject, such as destinations cities, departure dates, transportation methods, *etc.* (See Appendix G for the detailed prompt). It is important to note that although in-context examples only concern a limited number of constraints, the LLM is able to learn generic patterns and generalize to unseen constraints.

Optionally, converting natural language queries into fixed format JSON descriptions before generating the encoding steps could help LLMs summarize the key information and further improve the performance. However, prompting LLMs to generate JSON descriptions requires explanations of the needed fields in JSON, which could expose more information that helps LLMs to better understand the problem. For fairness, we do not generate JSON files when compared with other methods with natural language inputs. We include the result of our framework with this extra step in Appendix D.1.

3.3.2 Step-Code Generation

Step-code generation involves converting each of the steps generated in the previous stage into Python code, which would call the relevant APIs that collect information, such as `CitySearch`, `FlightSearch`, `AttractionSearch` *etc.*, as well as the SMT solver to execute the steps. Inspired by Liang et al. (2023), we teach the LLM to generate the code using examples as demonstrations, as shown in Fig. 1 (part 3). Each example contains a single step drawn from the query-code generation examples (Section 3.3.1), and the corresponding human-written code. To ensure comprehensive-

ness, our step-code examples cover almost all the steps mentioned in the query-step examples, less some duplicated or highly similar steps. Since the examples already contain sufficient demonstrations of all the API and SMT solver calls, the LLM can learn to use these APIs and the solver without separate API documentation. Fig. 2 shows how the LLM generalizes to new instruction steps to write corresponding codes given these examples.

3.3.3 SMT Solver

After generating codes, our framework executes these codes to encode the problem and call the SMT solver. Since the SMT solver is sound and complete, it guarantees to find a solution if there exists one. Thus, if the constraints are satisfiable, the solver generates a formally verified plan. If the constraints are not satisfiable, the solver outputs the unsatisfiable reasons and LLM could, based on its commonsense and reasoning capabilities, analyze the reasons, actively collect more information, and provide humans with suggestions to modify the constraints. We extract the unsatisfiable reasons with Z3 solver's `get_unsat_core` function. When the framework proves the constraints to be unsatisfiable, it proceeds to interactive plan repair with the unsatisfiable reasons.

3.4 Interactive Plan Repair

When a proposed query is not satisfiable, LLM's reasoning capability and commonsense knowledge to analyze the current situation and offer suggestions become vital. Furthermore, these capabilities enable an interactive setting, in which humans can agree, disagree, or provide feedback to LLM's proposed suggestions. LLM can deliver personalized plans built upon different human preferences.

Inspired by ReAct (Yao et al., 2022), in our framework, LLM could either take an action to collect information based on unsatisfiable reasons, analyze the current situation based on collected information, or provide suggestions. We equip LLM with information collection APIs and descriptions of their usage. As shown in Fig. 1, the unsatisfiable reason is *"invalid flight for transportation 0"*. With the reason, the LLM first collects flight information with FlightCheck API. Realizing no flight is available between St. Petersburg and Rockford, LLM analyzes and decides to change the destination city. It then runs FlightSearch API to search for all available destinations and chooses one of them. LLM offers this as a suggestion to the user. The user can optionally provide feedback, including yes, no, any natural language preference, or even modifications users proposed. If the user disagrees with the suggestion or provides their preferences, the framework starts another iteration and proposes new suggestions. If the users do not provide feedback, agree with the suggestion, or propose their own modification, the framework continues by inputting this modification, together with original codes, to an LLM and prompting it to modify the codes. By running the modified codes, the framework generates a plan if the modified constraints are satisfiable. Otherwise, the framework will gather the unsatisfiable reasons and start another round.

4 Dataset

To access our framework's ability to 1) generalize to unseen constraints and 2) interactive plan repair for unsatisfiable queries, we propose a dataset, UnsatChristmas, that introduces new constraints not included in TravelPlanner and contains 39 unsatisfiable queries under this setting. The queries in UnsatChristmas aim to create an international travel plan for Christmas week in 2023. We set cities in our dataset to be the top ten worldwide city destinations in 2019[1] and obtain attraction information from Metabase[2]. We utilize Google Flights[3] to collect flight information from 12-24-2023 to 12-30-2023 for these ten cities. Compared with TravelPlanner, we omit detailed information on transportation methods, restaurants, and accommodations but introduce detailed constraints regarding flights and attractions. As shown in Table 5, UnsatChristmas allows users to specify preferences for 1) non-stop flights, 2) the list of airlines, and 3) the list of attraction categories. We collect 39 unsatisfiable queries with 4 possible reasons: non-stop flight rule not satisfied, flight airline requirement not satisfied, attraction category requirement not satisfied, budget not enough. Out of the 39 queries, 12 fail due to one reason, 18 fail due to two reasons, 8 fail due to three reasons, and 1 fails due to four reasons. There are 13 queries with a single destination city, 13 with two, and 13 with three. In addition, to test the interactive plan repair performance, we also modify 12 queries from the training set of TravelPlanner to be unsatisfiable.

[1] https://go.euromonitor.com/white-paper-travel-2019-100-cities.html

[2] https://www.metabase.com/blog/data-guide-to-travel

[3] https://www.google.com/travel/flights

5 Experimental Results

We examine our framework on both TravelPlanner and UnsatChristmas. We use GPT-4 (Achiam et al., 2023) with temperature 0 by default, and we also compare with Claude 3 Opus-20240229 (cla) and Mistral-Large (Mis) with temperature 0 for satisfiable plan solving evaluation. We use Z3 SMT solver (De Moura and Bjørner, 2008). Since the solution space is prohibitive considering the combinatorial choices and a few queries have very limited feasible plans, we limit the SMT solver's maximum runtime for each query to 30 minutes. Please refer to Appendix B for cost and runtime analysis.

5.1 Satisfiable Plan Solving Evaluation

We examine how well our framework can create travel plans for satisfiable natural language queries on the TravelPlanner benchmark. We design our example instruction steps and corresponding codes using three queries from TravelPlanner's training set and tune the prompt with other queries in the training set. We evaluate our method on both the validation (180 queries) and the test set (1000 queries).

Evaluation Metric We adopt evaluation metrics from Xie et al. (2024) and mainly look at the Final Pass Rate, which represents whether LLMs pass all constraints. Please refer to Appendix D.2 for detailed descriptions of other evaluation metrics.

Baselines We compare our framework with the three strongest models using different strategies from Xie et al. (2024). Greedy Search uses a traditional search algorithm and heuristically optimizes for total cost. TwoStage (GPT-4), a two-stage tool-use framework, collects information with ReAct (Yao et al., 2022) and then gives plans. Direct (GPT-4), a sole-planning framework, has access to all necessary information and gives plans without tool-calling needs. We also include the result of Direct (o1-preview), the strongest reasoning model so far, for the validation set. Due to the long runtime of o1-preview, we do not evaluate Two-Stage or our framework with o1-preview. To verify the effectiveness of our framework in varied LLMs, we also evaluate with Claude 3 Opus and Mistral-Large. We tune the prompt with the training set and include the prompt differences in Appendix G.1.3.

Results and Analysis Table 1 shows the performance comparison. From the results, all LLM planning methods, TwoStage (GPT-4), Direct (GPT-4), and Direct (o1-preview), struggle to take all constraints into consideration with a final pass rate of 0.6%, 4.4%, and 10.0%. Without formal specification, Greedy Search fails to pass any of the tasks. Ours (Claude-3), with the capability of formally encoding the problem as an SMT problem, achieves the highest final pass rate of 93.3% and 93.9% for validation and test set. This demonstrates our framework's robustness in solving satisfiable queries. In addition, Ours (GPT-4) could reach comparable results as Ours (Claude-3). Although the pass rate for Ours (Mistral-Large) drops 26.6% and 26.1% compared to Ours (Claude-3), it still significantly outperforms baselines, and 92.3% and 97.0% of its delivered plans are correct plans. See Appendix G.1.4 for major failure cases of Ours (Mistral-Large). These results demonstrate the adaptability of our framework to various LLMs.

In addition, since inputs of baselines are natural language only, to ensure fairness, we do not include the additional step to translate natural language inputs into JSON representations. However, we show that including this step could even further improve the performance by helping LLMs extract and summarize key information before generating steps. We test on all three LLMs and achieve 98.9%, 98.3%, and 84.4% pass rates with on average 9.4% improvements on the validation set. We also test GPT-4 on the test set, which achieves 97.0%. Please refer to Appendix D.1 for details.

Flight is the only transportation method:

```
# Set transportation method (flight) variable for 3 transportations between cities
# Assert all 3 transportations between cities are flight
```

All taken flights are non-stop if "non-stop":

```
# For each 'flight_index', get specific flight rule and airlines info, assert 'non-stop' exist for all flights
```

All taken flights are within the required airlines list ['United', 'Emirates']:

```
# For each 'flight_index', get specific flight rule and airlines info, assert 'United' or 'Emirates' exist for all flights
```

All specified attraction in ["Garden", "Museum"] must be visited:

```
# Set 'attraction_category' variables for each category required
# For each attraction_category type in ["Garden", " Museum "], iterate through all attractions to check if it is satisfied
```

Figure 3: Example of how JSON-Step prompt generalizes to unseen constraints. Yellow: unseen constraint types. Green: corresponding generated steps.

5.2 Generalization Capability Analysis

5.2.1 New Prompts

Since both Query-Step and Step-Code prompts require careful design, we test the compatibility of our framework to handle diverse new prompts. We use GPT-4o (gpt) to paraphrase the entire Query-Step and Step-Code prompts, including the examples used in prompts, for every query in TravelPlan-

Table 1: Performance comparison of satisfiable queries for 180 queries in the validation set and 1000 queries in test set. The results of Greedy Search, TwoStage, and Direct with GPT-4 are from Xie et al. (2024)

Method	Delivery Rate	Commonsense Pass Rate		Hard Constraint Pass Rate		Final Pass Rate
		Micro	Macro	Micro	Macro	
***Validation** (#180)*						
Greedy Search	**100**	74.4	0	60.8	37.8	0
TwoStage (GPT-4)	89.4	61.1	2.8	15.2	10.6	0.6
Direct (GPT-4)	**100**	80.4	17.2	47.1	22.2	4.4
Direct (o1-preview)	**100**	79.6	15.0	41.9	37.8	10.0
Ours (Mistral-Large)	72.2	72.0	70.6	63.3	66.7	66.7
Ours (Claude-3)	96.1	**96.0**	**95.6**	94.8	93.3	**93.3**
Ours (GPT-4)	95.0	95.0	95.0	**95.7**	**98.9**	**93.3**
***Test** (#1000)*						
Greedy Search	**100**	72.0	0	52.4	31.8	0
TwoStage (GPT-4)	93.1	63.3	2.0	10.5	5.5	0.6
Direct (GPT-4)	**100**	80.6	15.2	44.3	23.1	4.4
Ours (Mistral-Large)	69.9	69.8	69.4	63.0	67.8	67.8
Ours (Claude-3)	95.4	**95.2**	**94.3**	**93.5**	**93.9**	**93.9**
Ours (GPT-4)	91.5	91.4	91.1	91.3	90.2	90.2

Table 2: Performance of zero-shot generalization to four other combinatorial optimization tasks.

	Block Picking		Task Allocation		TSP		Warehouse	
Method	Delivery	Optimal	Delivery	Optimal	Delivery	Optimal	Delivery	Optimal
TwoStage(GPT-4o)	80	4	84	0	100	0	72	0
Ours(GPT-4o)	100	92	92	92	100	100	84	72

Table 3: Performance of interactive plan repair for unsatisfiable queries on 39 queries from UnsatChristmas.

Method	Always Agree	Budget	Non-stop	Airline	Attraction Category	Destination Cities	Average
No Reason	74.4	61.5	69.2	53.8	69.2	53.8	63.7
No Feedback	N/A	59.0	79.5	61.5	79.5	74.4	70.8
No Solver	25.6	20.5	28.2	20.5	23.1	33.3	25.2
Ours	89.7	59.0	84.6	64.1	89.7	84.6	78.6
Ours-20	92.3	61.5	87.2	66.7	89.7	92.3	81.6

Table 4: Performance of interactive plan repair for unsatisfiable queries on 12 modified queries from TravelPlanner.

Method	Always Agree	Budget	Destination Cities	Transportation Methods	House Type	Average
No Reason	75	83.3	91.7	83.3	66.7	80
No Feedback	N/A	50	91.7	66.7	75	70.9
No Solver	16.7	16.7	50	25	16.7	25.0
Ours	91.7	75	100	83.3	75	85.0
Ours-20	100	83.3	100	91.7	83.3	91.7

ner's validation set and repeat for all 180 queries. We use the paraphrased prompts to test our framework, which achieves a final pass rate of 86.7% for GPT-4. This showcases the outstanding performance of our framework is not sensitive to the wording of prompts and does not heavily rely on the prompt design. Please refer to Appendix G.4 for example paraphrased prompts.

5.2.2 New Travel Plan Constraints

Since the travel planning problem involves various constraints of different types, our example instruction steps may not be comprehensive enough to cover all possible constraints. Here we examine our framework's robustness by testing whether it could generalize to the constraint types not shown in prompt examples in a zero-shot manner. As shown in Table 5, UnsatChristmas has different constraints than TravelPlanner. We show that by adding several lines of constraint description in the JSON-Step prompt, LLM could generate steps for new constraints without the need to add new examples. Figure 3 shows how our framework encodes unseen constraints in UnsatChristmas. Please see

Appendix G.3 for the added constraint description and see Appendix H for full generated steps.

5.2.3 New Multi-Constraint Planning Tasks

To show the capability of our framework to generalize to other domains, we conduct experiments in four new tasks: **Block Picking**, **Task Allocation**, **Travelling Salesman Problem (TSP)**, **Warehouse**. For each task, we create 25 different scenarios. See Appendix F.1 for detailed descriptions. For both Query-Step and Step-Code generation, we include one example from travel planning and a few lines of new problem description (Appendix F.2) in the prompt to test the zero-shot generalization capability. We implement the TwoStage tool-use framework as baselines. We use GPT-4o (gpt) as the LLM to account for long code generation. We use both delivery rate and optimal rate as the evaluation metrics. Results in Table 2 show that LLMs themselves fail to directly solve multi-constraint problems with large solution spaces, but our framework, with its knowledge of encoding and solving travel planning problems as SMT problems, could be adapted to other multi-constraint problems in a zero-shot manner with good optimal rate: 92%, 92%, 100%, and 72% respectively. Please refer to Appendix F.3 for failure cases analysis.

5.3 Interactive Plan Repair Evaluation

We examine our framework's interactive plan repair capability on both the modified queries from TravelPlanner and UnsatChristmas.

Evaluation Metric We evaluate success rate: if our framework successfully modifies the query and delivers a feasible plan within 10 iterations.

Implementation Details Queries of UnsatChristmas have four unsatisfactory modes: 1) not enough budget, 2) no required non-stop flight, 3) no required airline, 4) no required attraction category. We test our framework with simulated users with different preferences. One mimic user agrees to all suggestions proposed by LLM, and five mimic users have hard constraints for budget, non-stop flight, airline, attraction category, and destination cities, respectively. They refuse suggestions that change their hard constraint and provide feedback indicating they will not change this information.

Modified queries of TravelPlanner have three unsatisfactory modes: 1) not enough budget, 2) no required transportation method, and 3) no required house type. Mimic users have hard constraints for budget, destination cities, transportation methods, and house type.

Ablation Studies Key components in our framework include 1) LLM receives unsatisfiable reasons from the solver; 2) LLM collects information based on the reasons, analyzes, and offers suggestions; 3) LLM receives human preferences regarding offered suggestions and modifies codes; 4) SMT solver gives satisfiability verification. We perform ablation studies to examine these key components. We compare: 1) **No Reason**: asking LLM to resolve unsatisfiable queries without providing unsatisfiable reasons; 2) **No Feedback**: asking the human to only provide binary *"agree"* or *"disagree"* feedback without explaining why; 3) **No Solver**: removing the SMT solver. The LLM directly gives a list of suggestions in one iteration because no solver is provided to verify the updated query; 4) **Ours**: our approach (Section 3.4); 5) **Ours-20**: our approach with maximum 20 iterations.

Results and Analysis Table 3 and 4 show the interactive plan repair performance. Our framework successfully addresses an average of 78.6% and 85.0% diverse human preferences across all types of mimic humans. **Ours-20** raises the success rate to 81.6% and 91.7%, showing the potential of increasing iteration limits to achieve better results. We include figures of iterations versus performance in Appendix E. For queries from both datasets, **Ours** significantly outperforms **No Solver** by an average of 53.4% and 60.0% across all mimic humans. This suggests that LLM's capability to utilize the SMT solver to verify the modified query largely benefits the interactive plan repair process. **Ours** also outperforms **No Reason** by an average of 14.9% and 5.0% and outperforms **No Feedback** by an average of 7.8% and 14.1%. These results validate the effectiveness of our key components.

6 Conclusion

In this work, we propose a framework that enables LLMs to utilize an SMT solver to formally formulate and solve complex realistic planning problems as constrained satisfiability problems. Our framework generalizes to natural language query inputs, almost guarantees to deliver plans for satisfiable queries with a pass rate of 93.9%, and provides personalized suggestions to modify unsatisfiable queries. We prove our framework can handle diverse paraphrased prompts. We also show that our framework can generalize to unseen constraint types and new domains in a zero-shot manner.

7 Limitation

The limitations and potential risks of the work are as follows:

Prompt Designing We need a careful design of instruction steps and corresponding codes to encode the problem. It is time-consuming to formulate the problem from scratch. However, as discussed in Section 2 **LLM Prompt Design**, most of the existing works that could achieve strong performance on complex planning problems rely on different forms of task-specific efforts. In addition, the potential of our framework to generalize to the unseen constraints and unseen tasks eases the future efforts needed to incorporate more constraints into the framework and to solve more different multi-constraint problems. We also show our framework compatibility to handle diverse paraphrased prompts. Moreover, as the designers of the framework, we design offline prompts to enable full model autonomy for end users. Thus, after these prompts are designed, the effort needed for any end user to utilize our framework is a simple natural language query. With our framework, end users can utilize powerful solvers to solve their problems without having any knowledge about the solvers.

Solver Runtime For a massive database with more destination city choices, various constraint types, and queries that only have a few feasible plans, our framework could take a long runtime to find the plan. To relieve this limitation, a potential way is to introduce some heuristics and prioritize a portion of choices to be verified first.

Risky Data Since all information sources of our framework is from the database we use, it currently does not have the capability to distinguish unsafe or incorrect information. One potential risk of our framework is that it could generate risky plans based on unsafe information from the database.

References

Au large. https://mistral.ai/news/mistral-large/. Accessed: 2024-02-26.

Hello gpt-4o. https://openai.com/index/hello-gpt-4o/. Accessed: 2024-05-13.

Introducing openai o1-preview. https://openai.com/index/introducing-openai-o1-preview/. Accessed: 2024-09-12.

Introducing the next generation of claude. https://www.anthropic.com/news/claude-3-family. Accessed: 2024-03-04.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. 2022. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*.

Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. cvc4. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 171–177. Springer.

Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690.

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.

Yongchao Chen, Jacob Arkin, Yilun Hao, Yang Zhang, Nicholas Roy, and Chuchu Fan. 2024. Prompt optimization in multi-step tasks (promst): Integrating human feedback and preference alignment. *arXiv preprint arXiv:2402.08702*.

Yongchao Chen, Jacob Arkin, Yang Zhang, Nicholas Roy, and Chuchu Fan. 2023a. Autotamp: Autoregressive task and motion planning with llms as translators and checkers. *arXiv preprint arXiv:2306.06531*.

Yongchao Chen, Jacob Arkin, Yang Zhang, Nicholas Roy, and Chuchu Fan. 2023b. Scalable multi-robot collaboration with large language models: Centralized or decentralized systems? *arXiv preprint arXiv:2309.15943*.

Neil T Dantam, Zachary K Kingston, Swarat Chaudhuri, and Lydia E Kavraki. 2016. Incremental task and motion planning: A constraint-based approach. In *Robotics: Science and systems*, volume 12, page 00052. Ann Arbor, MI, USA.

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.

Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77.

Rina Dechter. 2003. *Constraint processing*. Morgan Kaufmann.

Bruno Dutertre and Leonardo De Moura. 2006. A fast linear-arithmetic solver for dpll (t). In *International Conference on Computer Aided Verification*, pages 81–94. Springer.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2023. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. 2023. Leveraging pretrained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, 36:79081–79094.

Atharva Gundawar, Mudit Verma, Lin Guan, Karthik Valmeekam, Siddhant Bhambri, and Subbarao Kambhampati. 2024. Robust planning with llm-modulo framework: Case study in travel planning. *arXiv preprint arXiv:2405.20625*.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*.

Malte Helmert. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246.

Jörg Hoffmann and Bernhard Nebel. 2001. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022a. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pages 9118–9147. PMLR.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. 2022b. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*.

Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. 2024. Understanding the planning of llm agents: A survey. *arXiv preprint arXiv:2402.02716*.

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Paul Saldyt, and Anil B Murthy. Position: Llms can't plan, but can help planning in llm-modulo frameworks. In *Forty-first International Conference on Machine Learning*.

Henry Kautz and Bart Selman. 1999. Unifying sat-based and graph-based planning. In *IJCAI*, volume 99, pages 318–325.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Beibin Li, Konstantina Mellou, Bo Zhang, Jeevan Pathuri, and Ishai Menache. 2023. Large language models for supply chain optimization. *arXiv preprint arXiv:2307.03875*.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500. IEEE.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. 2023. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*.

Tomás Lozano-Pérez and Leslie Pack Kaelbling. 2014. A constraint-based method for solving sequential manipulation planning problems. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3684–3691. IEEE.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

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

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. 2022. Measuring and narrowing the compositionality gap in language models. *arXiv preprint arXiv:2210.03350*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Jussi Rintanen. 2012. Planning as satisfiability: Heuristics. *Artificial intelligence*, 193:45–86.

Jussi Rintanen. 2014. Madagascar: Scalable planning with sat. *Proceedings of the 8th International Planning Competition (IPC-2014)*, 21:1–5.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2024. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. 2023. Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11523–11530. IEEE.

Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. 2023. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2998–3009.

Vincent Vidal. 2014. Yahsp3 and yahsp3-mt in the 8th international planning competition. *Proceedings of the 8th International Planning Competition (IPC-2014)*, pages 64–65.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric P Xing, and Zhiting Hu. 2023. Promptagent: Strategic planning with language models enables expert-level prompt optimization. *arXiv preprint arXiv:2310.16427*.

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

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

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. 2024. Travelplanner: A benchmark for real-world planning with language agents. *arXiv preprint arXiv:2402.01622*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Zirui Zhao, Wee Sun Lee, and David Hsu. 2024. Large language models as commonsense knowledge for large-scale task planning. *Advances in Neural Information Processing Systems*, 36.

Large Language Models Can Solve Real-World Planning Rigorously with Formal Verification Tools

1 Introduction **1**

2 Related Work **2**

3 Approach **3**
- 3.1 Problem Formulation . . . 3
- 3.2 Framework Overview . . . 4
- 3.3 Satisfiable Plan Solving . . . 4
  - 3.3.1 Query-Step Generation . . . 4
  - 3.3.2 Step-Code Generation . . . 4
  - 3.3.3 SMT Solver . . . 5
- 3.4 Interactive Plan Repair . . . 5

4 Dataset **5**

5 Experimental Results **6**
- 5.1 Satisfiable Plan Solving Evaluation . . . 6
- 5.2 Generalization Capability Analysis . . . 6
  - 5.2.1 New Prompts . . . 6
  - 5.2.2 New Travel Plan Constraints . . . 7
  - 5.2.3 New Multi-Constraint Planning Tasks . . . 8
- 5.3 Interactive Plan Repair Evaluation . . . 8

6 Conclusion **8**

7 Limitation **9**

A Description of constraints **13**

B Runtime and cost analysis **14**
- B.1 Satisfiable Plan Solving . . . 14
- B.2 Interactive Plan Repair . . . 14

C Example input queries and output plans **15**

D Satisfiable Plan Solving Evaluation Details **17**
- D.1 Result of Ours+JSON on TravelPlanner . . . 17
- D.2 Description of other evaluation metrics . . . 17

E Interactive Plan Repair: Iteration versus Performance **18**

F New Multi-constraint Tasks details **19**
- F.1 Task Setup . . . 19
- F.2 Added task description in prompt . . . 19
- F.3 Failure case analysis . . . 20

G Prompts **22**
- G.1 Prompts for Satisfiable Plan Solving . . . 22
- G.2 Prompts for Interactive Plan Repair . . . 33
- G.3 Prompts for Generalization Evaluation . . . 39
- G.4 Paraphrased Prompt Examples . . . 40

H Example input query and output step for generalization evaluation **47**

A Description of constraints

Constraint	Description
Destination cities	Destination cities should not be repeated
Transportation dates	First transportation happens at first day, last transportation happens at last day, and others happens in between non-repeatedly
Transportation methods	Every transportation uses flight, self-driving, or taxi Self-driving is not valid during the trip if taxi or flight is used No flight if *"no flight"* is mentioned, and no self-driving if *"no self-driving"* is mentioned
Flight	No flight if flights unavailable between two cities on certain dates All taken flights are non-stop if *"non-stop"* is mentioned All taken flights' airlines are within the required airlines list
Driving	No driving if driving routes unavailable between two cities
Restaurant	Restaurant choices should not be repeated Restaurant for day must be located within that day's city All specified cuisine types must be visited
Attraction	Attraction choices should not be repeated Attraction for day must be located within that day's city All specified attraction types must be visited
Accommodation	Accommodation for day must be located within that day's city All specified accommodations must satisfy specified Room Rule All specified accommodations must satisfy specified Room Type The number of consecutive days spent in an accommodation must meet the accommodation's minimum number of nights' stay.
Budget	Total spend is within specified budget

Table 5: Descriptions of constraints for two datasets. Constraints in teal are the constraints only in TravelPlanner. Constraints in brown are the constraints only in our dataset. Constraints in black are common constraints.

B Runtime and cost analysis

Since TravelPlanner's database has 65 states, 312 cities, 3827361 flights, 17603 driving information, 5303 attractions, 9552 restaurants, and 5064 accommodations, the solution space is extremely large considering the combinatorial choices. In addition, a few queries are challenging in that they have few feasible plans. We limit SMT solver's maximum runtime for each query to 30 minutes. However, we show below that solver is not time consuming for most of the cases.
In this section, we include the detail runtime and cost analysis of both satisfiable plan solving and interactive plan repair of our framework.

B.1 Satisfiable Plan Solving

For the satisfiable plan solving part, we recorded the runtime and cost for 180 queries in TravelPlanner's validation set. Over the 180 queries, the average cost is $0.74 per query using GPT-4. Over the 179 queries with delivered plans, the average time spent for different stages in our framework are shown in Table 6. The average total time spent for all stages is 245.66 seconds (4.09 minutes) per query. The Step-Code generation contains multiple LLM calls for various types of constraints, thus takes most of the time.

LLM NL-JSON	LLM JSON-Step	LLM Step-Code	SMT Solver	Total
5.45	35.16	166.66	38.39	245.66

Table 6: Runtime (seconds) of each stage of our framework for satisfiable plan solving.

Out of the 180 queries, there is one query with no delivered plan since its runtime exceeds 30 minutes. For queries with heavy computational costs, introducing some heuristics that prioritize a portion of all possible solutions could help to reduce the computational overhead of SMT solvers. Our framework introduces a simple heuristic: for queries that ask to visit multiple cities in a state, we will prioritize the cities with available transportations between the origin. This heuristic helps to reduce the runtime, especially for a big state with 20 cities. In addition to this simple heuristic, some other heuristics may help, which we plan to explore more in the future: pre-calculate estimated money spent and prioritize the cheaper solutions, prioritize the cities with a larger number of transportation methods/ restaurants/ accommodations, etc.

B.2 Interactive Plan Repair

For the interactive plan solving part, we recorded the runtime and cost for queries in UnsatChristmas for mimic-human with hard budget constraints. Over the 23 (out of 39) successful queries, the average cost is $0.65 per iteration using GPT-4. The average time spent for different stages in our framework are shown in Table 7. The average total time spent for both stages is 33.68 seconds per iteration. Note that for mimic-human with hard budget constraints, the average number of iterations that successfully modify the queries is 2.22 per query.

LLM interactive suggestion	Code Update	Total
10.35	23.33	33.68

Table 7: Runtime (seconds per iteration) of each stage of our framework for interactive plan repair.

C Example input queries and output plans

In an query, the user can specify 1) length of travel (3, 5, or 7 days), 2) the destination city or state (for 5/7 days travel, the destination cities would be 2/3 cities from a state), 3) travel dates, 4) budget, 5) preferences regarding transportation methods, 6) preferences regarding restaurant cuisine types, 7) preferences regarding accommodation type and rules.
We list an example input query and the corresponding output plans.

Input query:
Can you create a 5-day travel itinerary for a group of 3, departing from Atlanta and visiting 2 cities in Minnesota from March 3rd to March 7th, 2022? We have a budget of $7,900. We require accommodations that allow parties and should ideally be entire rooms. Although we don't plan to self-drive, we would like the flexibility to host parties.
Corresponding output plan:
{
"days": 1,
"current_city": "from Atlanta to Minneapolis(Minnesota)",
"transportation": "Taxi, from Atlanta to Minneapolis(Minnesota), duration: 16 hours 26 mins, distance: 1,821 km, cost: 1821",
"breakfast": "-",
"attraction": "Minneapolis Sculpture Garden, Minneapolis(Minnesota)",
"lunch": "-",
"dinner": "Bawa Chicken, Minneapolis(Minnesota)",
"accommodation": "Light-filled Apartment in great area, close 2 all!, Minneapolis(Minnesota)"
},
{
"days": 2,
"current_city": "from Minneapolis(Minnesota) to St. Cloud(Minnesota)",
"transportation": "Taxi, from Minneapolis(Minnesota) to St. Cloud(Minnesota), duration: 1 hour 6 mins, distance: 105 km, cost: 105",
"breakfast": "Friends Restaurant, St. Cloud(Minnesota)",
"attraction": "Blacklight Adventures, St. Cloud(Minnesota)",
"lunch": "Madras Cafe, St. Cloud(Minnesota)",
"dinner": "Aggarwal's Sweets Paradise, St. Cloud(Minnesota)",
"accommodation": "The Gramercy East, St. Cloud(Minnesota)"
},
{
"days": 3,
"current_city": "St. Cloud(Minnesota)",
"transportation": "-",
"breakfast": "Mocha, St. Cloud(Minnesota)",
"attraction": "AirMaxx Trampoline Park & Warrior Course, St. Cloud(Minnesota)",
"lunch": "House of Commons, St. Cloud(Minnesota)",
"dinner": "Cakewalk - Park Plaza, St. Cloud(Minnesota)",
"accommodation": "The Gramercy East, St. Cloud(Minnesota)"
},
{
"days": 4,
"current_city": "St. Cloud(Minnesota)",
"transportation": "-",

"breakfast": "Pa Pa Ya, St. Cloud(Minnesota)",
"attraction": "Beaver Island Trail South, St. Cloud(Minnesota)",
"lunch": "Grecian Key Restaurant, St. Cloud(Minnesota)",
"dinner": "Food Care, St. Cloud(Minnesota)",
"accommodation": "The Gramercy East, St. Cloud(Minnesota)"
},
{
"days": 5,
"current_city": "from St. Cloud(Minnesota) to Atlanta",
"transportation": "Taxi, from St. Cloud(Minnesota) to Atlanta, duration: 17 hours 19 mins, distance: 1,919 km, cost: 1919",
"breakfast": "Annapurna Sweets, St. Cloud(Minnesota)",
"attraction": "-",
"lunch": "Republic of Chicken, St. Cloud(Minnesota)",
"dinner": "-",
"accommodation": "-"
}

D Satisfiable Plan Solving Evaluation Details

D.1 Result of Ours+JSON on TravelPlanner

Here we include the Ours+JSON framework and satisfiable Plan Solving result of Ours+JSON on dataset TravelPlanner. We test the same LLMs (GPT-4, Claude-3-Opus, and Mistral-Large) on the validation set and test GPT-4 on the test set of TravelPlanner. Our framework achieves final pass rates of 98.9%, 98.3%, and 84.4% respectively on validation set, and 97.0% for GPT-4 on test set. From the result, the performance is further improved by 5.6%, 5.0%, and 17.7% for GPT-4, Claude-3, and Mistral on validation set, and is further improved by 6.8% for GPT-4 on test set. Since translating the JSON allows the LLMs to first extract and summarize the key information from the natural langauge query into a fix-formatted clear representation, it helps to understand and represent the problem better.

Table 8: Result comparison of Ours and Ours+JSON with three LLMs on TravelPlanner

	Valid (Ours)	Valid (Ours+JSON)	Test (Ours)	Test (Ours+JSON)
GPT-4	93.3	98.9	90.2	97.0
Claude-3	93.3	98.3	93.9	N/A
Mistral-Large	66.7	84.4	67.8	N/A

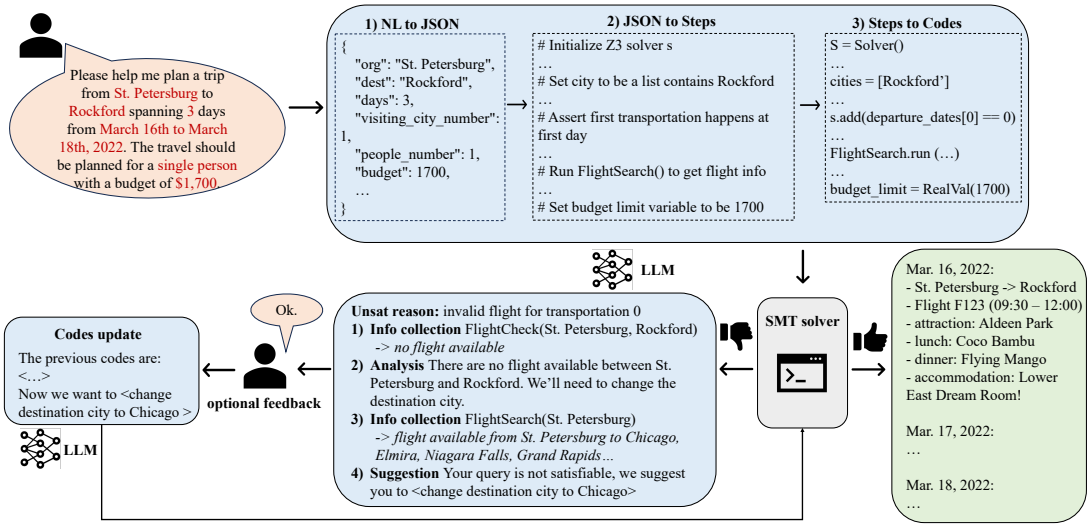


Figure 4: An overview of the framework with JSON.

D.2 Description of other evaluation metrics

Delivery Rate measures whether a final plan is generated within a limited time.

Commonsense constraints defined in (Xie et al., 2024) include: all information in the plan is within closed sandbox, the plan is complete without any left out part, all activities should be conducted in current city, travel route is reasonable, restaurant and attractions should not be repeated, transportation is reasonable (no self-driving if taxi or flight is taken during the travel), the nunmber of consecutive days spent in a specific accommodation must meet its required minimum number of nights' stay.

Hard constraints include: the total spend of the trip is within budget, the specified room rule does not exist ("No parties", "No smoking", "No children under 10", "No pets", and "No visitors"), the specified room type exists ("Entire Room", "Private Room", "Shared Room", and "No Shared Room"), the specified cuisine types are fulfilled during the trip ("Chinese", "American", "Italian", "Mexican", "Indian", "Mediterranean", and "French"), the specified transportation method is satisfied ("No flight" and "No self-driving".).

For Commonsense Constraint Pass Rate and Hard Constraint Pass Rate, two evaluation modes, micro and macro, are used to test the agent's capability to follow single constraint and follow constraints holistically. Micro calculates the ratio of passed constraints to the total number of constraints, while Macro calculates the ratio of plans that pass all commonsense or hard constraints among all tested plans.

E Interactive Plan Repair: Iteration versus Performance

Figure 5 shows the performance (success rate %) of interactive plan repair over different numbers of iterations for both datasets.

For the 39 queries in UnsatChristmas, 63.7% of the queries could be successfully modified to be satisfiable within 3 iterations, 74.8% within 5 iterations, 78.6% within 10 iterations, and 81.6% within 20 iterations. The performance increases quickly during the first 5 iterations, and the framework solves a limited number of more difficult queries with more iterations.

Similarly, for the 12 modified queries in TravelPlanner, 65.0% of the queries could be successfully modified to be satisfiable within 3 iterations, 73.3% within 5 iterations, 85.0% within 10 iterations, and 91.7% within 20 iterations.

The results suggest that we do not need extensive iterations to fully capture a major portion of the human queries.

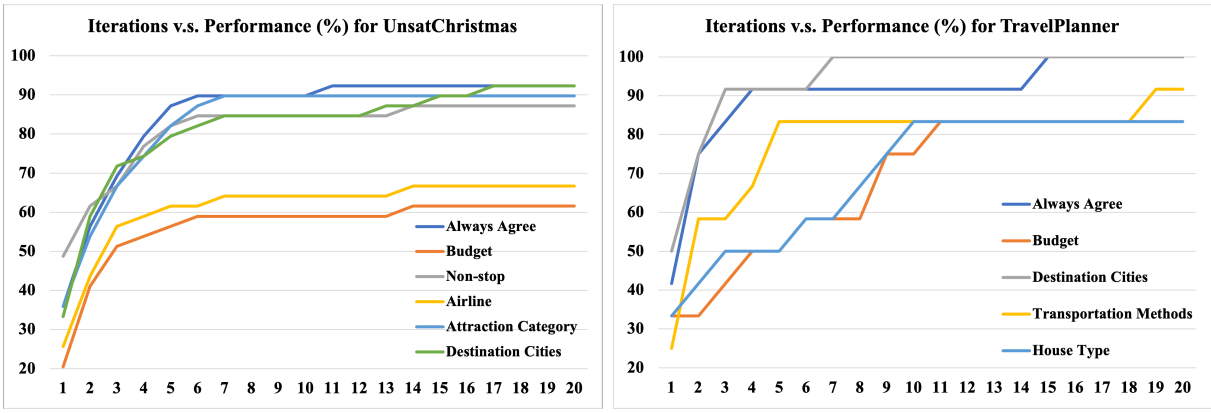
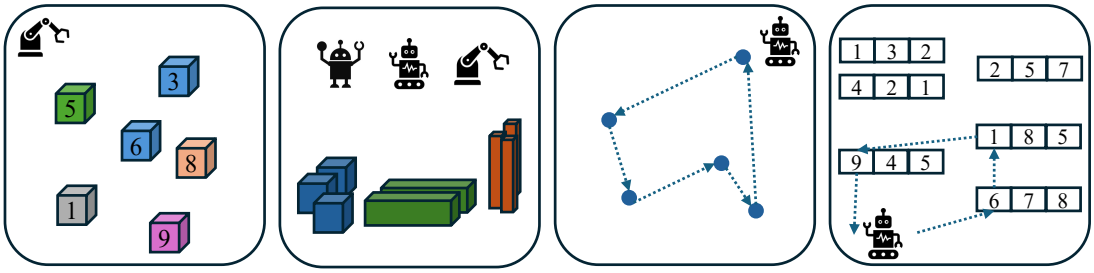


Figure 5: Performance (success rate %) of interactive plan repair over different numbers of iterations for two datasets

F New Multi-constraint Tasks details

F.1 Task Setup



Block Picking Task Allocation TSP Warehouse

Figure 6: Four new multi-constraint tasks.

F.1.1 Block Picking

There are blocks of different colors and scores in the scene. The goal is to select required number of unique blocks with required color, while maximizing the score. All possible block colors are red, yellow, black, pink, and blue. For 25 different scenarios, we set the total number of blocks to be a random number between 50 to 200, each with a random score between 1 to 20 and a random color. For the query, we will randomly choose 1 to 3 colors from all colors to be the required color, and 10 to be the required number of blocks to pick.

F.1.2 Task Allocation

Given a list of three tasks A, B and C, and three heterogeneous robots that are skilled at different tasks, the goal is to find the way to assign different tasks to different robots and finish the tasks with minimized finish time. The three robots could work in parallel, but the finish time counts the time when the last robot stops working. For 25 different scenarios, we set number of task A, B and C to be random numbers between 10 to 100. For each robot, we set its work time to finish each task to be a random number between 10 to 100.

F.1.3 TSP

Given a list of ten cities, the goal is to visit each city exactly once with minimized distance travelled. For 25 different scenarios, we set the coordinates of each city to be a random tuple between 0 and 1.

F.1.4 Warehouse

The robot has a task list of length N that needs to be finished one by one. In the warehouse, there are 50 stations, where the robots can visit stations to finish different tasks. The robot starts at station 0, travel n stations to finish n tasks, and then travel back to station 0. The robot needs to finish n tasks while minimizing the total distance travelled. For 25 different scenarios, we set the total task number to be 50, and the total station number to be 50, and each station can be used to accomplish 3 random tasks. We set the number of tasks the robot need to finish to be a random number from 3 to 10, and the tasks to be random numbers within 50. We set the coordinates of each station to be a random tuple between 0 and 1.

F.2 Added task description in prompt

Block Picking
Now, you are given a JSON constraint of a block stacking problem.
There are blocks of different colors and scores in the scene. You need to select "block_number" non-repeat blocks with color in "color" list, while maximizing the score.
You have access to a BlockSearch API.
BlockSearch.run(color:list) gives 1.all possible block ids of color in "color" list and 2.corresponding

score info. You should assert chosen blocks index does not exceed block id list length.
BlockSearch.get_info(score_info, block_index) gives the score of certain block.

Task Allocation
Now, you are given a JSON constraint of a task allocation problem.
Given a list of tasks and three heterogeneous robots that are skilled at different tasks, the goal is to find the way to assign different number of tasks to different robots and finish the tasks with minimized finish time. The three robots could work in parallel, but the finish time counts the time when the last robot stops working.
You have access to a TimeSearch() API.
TimeSearch.run() searches robots' accomplishing time info.
TimeSearch.get_info(time_info, robot: str, task: str) gives the accomplishing time of certain block for certain task. An example of robot and task string is: 'robotA', 'taskA'.
Note that for each robot R and each task T, the number of task T robot R is allocated needs to be non-negative and within the total number of T.
You have access to a Max(variable_list) function that outputs the max of a list of variables.

TSP
Now, you are given a JSON constraint of a travelling salesman problem (TSP) problem.
Given 10 cities, you need to non-repeatly visit each city exactly once with minimized distance travelled.
You have access to a DistanceSearch() API.
DistanceSearch.run() takes no argument and gives the distance info between cities, and Distance-Serarch.get_info(distance_info, city_1, city_2) gives the distance(a real number) between two cities.
You should explicitly assert city index does not exceed total number of cities.
You do not need to count the distance to go back to the first city, so the total number of distances you need to consider is 9.

Warehouse
Now, you are given a JSON constraint of a warehouse robot routing problem.
The robot are given a task list "task_id" of length n and needs to finish them one by one. In the warehouse, there are "total_station_number" stations, where the robots can visit stations to finish different tasks.
The robot needs to finish n tasks while minimizing the total distance travelled. When calculating total travel distance, make sure to include 1.the distance to travel from origin (0) to first station; 2.distance between n stations (so this is n-1 values); 3.distance to travel from last station back to origin(0).
You have access to a StationSearch() API.
StationSearch.run_task(tasks) takes a list of tasks that the robot needs to accomplish, and gives a list of stations_id_list. For each task, you should assert robot choose one station id from corresponding stations_list, which is all possible stations.
StationSearch.run_distance() takes no argument and gives the distance info between stations, and StationSearch.get_info(distance_info, station_1_id, station_2_id) takes gives the distance (a real number) between two stations.

F.3 Failure case analysis

For Block Picking, Task Allocation, and Warehouse, there are failure cases.

Block Picking For Block Picking, LLM fails to give the optimal plan for 2 out of 25 delivered

plans. The failure reasons are same for these two plans. In the block picking task, all picked blocks need to be distinct. Thus, in the codes LLM writes, it needs to explicitly check all block indexes it chooses are different. In these two plans, the LLM fails to take this into account, thus repeatedly choose same blocks with high scores to maximize the score.

Task Allocation For Task Allocation, LLM fails to deliver the plan for 2 out of 25 scenarios. The failure reasons are same. The LLMs are provided a Max(variable_list) function, which takes in a list of variables and return the max. However, in the codes written by LLM, they fail to input a list of variables, but input the variables themselves one by one. This gives runtime errors thus fails to deliver the plan.

Warehouse For Warehouse, LLM fails to deliver the plan for 4 out of 25 scenarios, and fails to give the optimal plan for 3 out of the 21 delivered plan.
Note that Warehouse is a more challenging task in that it needs to select stations to visit while calculate the minimum travel distance between them at the same time. Thus, the codes are more complex to write. The delivery failure reasons for the 4 scenarios are same. Since the task requires the robot to travel from origin, visit n stations, and back to origin at the end. LLM could choose to set n station variables to represent n stations needs to visit or set n+2 station variables and make the first and last one to equal to 0. However, in the failure cases, the LLM set n variables to represent n stations, but at the same time assert first and last one to equal to 0. This brings conflicts because the it could assert one variable to equal to two values at the same time.
The non-optimal reasons for the 3 delivered plan are same: since the StationSearch.run_task(tasks) outputs a station_id_list, the contents are IDs of stations. LLM needs to create station variables and assert it to equal to *value* from the ID lists. However, in the codes, LLM assert the station variables to equal to *index* from zero to length of station_id_list. This makes it to calculate incorrect distance thus outputting non-optimal solutions.

G Prompts

G.1 Prompts for Satisfiable Plan Solving

G.1.1 Query-Step prompt

The instruction prompt for natural language to steps translation prompt is provided as follows:

You are given a natural language query that contains important constraints to satisfy for a travel plan problem. These constraints could include departure city, destination state or city, total travel days and dates, and some spcecial requirements regarding accommodation, restaurant, and transportation.
Your job is to give a detailed step by step instuction to encode this constraint as code.
Here are some example steps for different constraint:
——EXAMPLE 1——
Natural Language query:
Can you create a 5-day travel itinerary for a group of 3, departing from Atlanta and visiting 2 cities in Minnesota from March 3rd to March 7th, 2022? We have a budget of $7,900. We require accommodations that allow parties and should ideally be entire rooms. Although we don't plan to self-drive, we would like the flexibility to host parties.
Steps:
Destination cities
Run CitySearch to get all possible destination cities in Minnesota State for dates ["2022-03-16", "2022-03-17", "2022-03-18"] from origin 'Atlanta', remove origin 'Atlanta' if it is in list
Loop through cities for 2 destination cities
Initialize Z3 solver s
Set 'city' variable to be indexes of 2 destination cities
If city_0_index and city_1_index are not same, assert 2 'city' variables equal to city index

Departure dates
Set 'departure_dates' variables for 3 transportations between cities
Assert first transportation happens at first day (day 0), last transportation happens at last day (day 4), and second transportation could happen at any day in between

Transportation methods
Set transportation method (flight, self-driving, taxi) variable for 3 transportations between cities
Assert only one of flight, self-driving, or taxi is used for 3 transportations between cities, self-driving is not valid if taxi or flight is used for any transportation
Assert all 3 transportations between cities are not self-driving

Flight information
Run FlightSearch to get flight info for Atlanta as origin, list of cities, city_0 and city_1, and dates ["2022-03-16", "2022-03-17", "2022-03-18"]
Get specific flight price info with Atlanta as origin and final destination, specific city variable, and departure date for 3 transportations
Set 'flight_index' variable for 3 transportations
Assert 3 'flight_index' variables are within valid range if taking flight, assert flight index to be -1 if not taking flight
Calculate flight price for 3 people for 3 transportations based on flight index variable
Get specific flight arrival time info with Atlanta as origin and final destination, specific city, and departure date for 3 transportations
Calculate flight arrival time for 3 transportations based on flight index variable

Driving information
Run DistanceSearch to get driving info for Atlanta as origin and city_0 and city_1
Get specific driving distance info with Atlanta as origin and final destination, specific city, and departure date for 3 transportations
Assert driving info is not empty if driving
Calculate self-driving and taxi price for 3 people and 3 transportations based on driving distance
Get driving arrival time with Atlanta as origin and final destination, specific city, and departure date for 3 transportations

Restaurant information
Get arrivals and city list for each day based on 3 transportations, 5 total travel day, and departure dates variables
Run RestaurantSearch to get restaurant price info and cuisine info for city_0 and city_1
Set 'restaurant_in_which_city' variables for 15 (3 meals per day, 5 days) meals
For each 'restaurant_in_which_city' variable, assert it to be either current city or next city based on transportation arrivals time
Set 'restaurant_index' variables for 15 (3 meals per day, 5 days) meals
For each 'restaurant_index', get specific price info based on 'restaurant_in_which_city' variable, assert index are within valid range, assert restaurants in same city are not repeated, and calculate restaurant price for 3 people
Calculate restaurant price based on restaurant index

Attraction information
Run AttractionSearch to get attraction info for city_0 and city_1
Set 'attraction_in_which_city' variables for 5 (1 per day) attractions
For each 'attraction_in_which_city' variable, assert it to be either current city or next city based on transportation arrivals time
Set 'attraction_index' variables for 5 (1 per day) attractions
For each 'attraction_index', get specific length info based on attraction in which city variable, assert index are within valid range, and attrations in same city are not repeated

Accommodation information
Run AccommodationSearch to get accommodation info and accommodation constraints for city_0 and city_1
Set 'accommodation_index' variables for 2 (1 per city) accommodations
For each 'accommodation_index', get specific price info based on accommodation in which city variable, assert 'accommodation_index' variable are within valid range, calculate number of room need for 3 people and accommodation price
For each city, get accommodation minimum night info and assert it to be less than the days stay in this city
For each 'accommodation_index', get specific room type and house rules info, assert 'Entire home/apt' exist for all accommodations, assert 'No parties' does not exist for all accommodations

Budget
Set budget limit variable to be 7900
Add 3 transportation price to spent, according to whether transportation method is flight, self-driving, or taxi
Add restaurant price to spent
Add accommodation price to spent
Assert current spent is within budget

——EXAMPLE 2——
Natural Language query:
Can you help with generating a 7-day travel plan for a party of 5? We're setting off from Indianapolis and planning to explore 3 cities in Colorado from March 11th to March 17th, 2022. We have a budget of $15,100 for this trip. We'll be bringing our pets, so pet-friendly accommodations are a must. It's important for us to stay in accommodations that permit children under the age of 10. We're also hoping to find places that offer Mexican, Italian, Mediterranean, and Indian cuisines. No shared rooms for accommodations would be ideal. We do not have preferences for transportation.
Steps:
Destination cities
Run CitySearch to get all possible destination cities in Colorado State for dates ['2022-03-11', '2022-03-12', '2022-03-13', '2022-03-14', '2022-03-15', '2022-03-16', '2022-03-17'] from origin 'Indianapolis', remove origin 'Indianapolis' if it is in list
Loop through cities for 3 destination cities
Initialize Z3 solver s
Set 'city' variable to be indexes of 3 destination cities
If city_0_index, city_1_index, city_2_index are not same, assert 3 'city' variables equal to city index

Departure dates
Set 'departure_dates' variables for 4 transportations between cities
Assert first transportation happens at first day (day 0), last transportation happens at last day (day 6), second and third transportation happen in between but not at the same day

Transportation methods
Set transportation method (flight, self-driving, taxi) variable for 4 transportations between cities
Assert only one of flight, self-driving, or taxi is used for 4 transportations between cities, self-driving is not valid if taxi or flight is used for any transportation

Flight information
Run FlightSearch to get flight info for Indianapolis as origin, list of cities, city_0, city_1 and city_2, and dates ['2022-03-11', '2022-03-12', '2022-03-13', '2022-03-14', '2022-03-15', '2022-03-16', '2022-03-17']
Get specific flight price info with Indianapolis as origin and final destination, specific city variable, and departure date for 4 transportations
Set 'flight_index' variable for 4 transportations
Assert 4 'flight_index' variables are within valid range if taking flight, assert flight index to be -1 if not taking flight
Calculate flight price for 5 people for 4 transportations based on flight index variable
Get specific flight arrival time info with Indianapolis as origin and final destination, specific city, and departure date for 4 transportations
Calculate flight arrival time for 4 transportations based on flight index variable

Driving information
Run DistanceSearch to get driving info for Indianapolis as origin and city_0, city_1 and city_2
Get specific driving distance info with Indianapolis as origin and final destination, specific city, and departure date for 4 transportations
Assert driving info is not empty if driving
Calculate self-driving and taxi price for 5 people and 4 transportations based on driving distance
Get driving arrival time with Indianapolis as origin and final destination, specific city, and

departure date for 4 transportations

Restaurant information
Get arrivals and city list for each day based on 4 transportations, 7 total travel day, and departure dates variables
Run RestaurantSearch to get restaurant price info and cuisine info for city_0, city_1 and city_2
Set 'restaurant_in_which_city' variables for 21 (3 meals per day, 7 days) meals
For each 'restaurant_in_which_city' variable, assert it to be either current city or next city based on transportation arrivals time
Set 'restaurant_index' variables for 21 (3 meals per day, 7 days) meals
For each 'restaurant_index', get specific price info based on 'restaurant_in_which_city' variable, assert index are within valid range, assert restaurants in same city are not repeated, and calculate restaurant price for 5 people
Set 'cuisine_type' variables for each cuisine type ['Mexican', 'Italian', 'Mediterranean', 'Indian'] required
For each cuisine type, iterate through all restaurant to check if it is satisfied

Attraction information
Run AttractionSearch to get attraction info for city_0, city_1 and city_2
Set 'attraction_in_which_city' variables for 7 (1 per day) attractions
For each 'attraction_in_which_city' variable, assert it to be either current city or next city based on transportation arrivals time
Set 'attraction_index' variables for 7 (1 per day) attractions
For each 'attraction_index', get specific length info based on attraction in which city variable, assert index are within valid range, and attrations in same city are not repeated

Accommodation information
Run AccommodationSearch to get accommodation info and accommodation constraints for city_0, city_1 and city_2
Set 'accommodation_index' variables for 3 (1 per city) accommodations
For each 'accommodation_index', get specific price info based on accommodation in which city variable, assert 'accommodation_index' variable are within valid range, calculate number of room need for 5 people and accommodation price
For each city, get accommodation minimum night info and assert it to be less than the days stay in this city
For each 'accommodation_index', get specific room type and house rules info, assert 'Shared room' does not exist for all accommodations, assert 'No pets' does not exist for all accommodations, assert 'No children under 10' does not exist for all accommodations

Budget
Set budget limit variable to be 15100
Add 4 transportation price to spent, according to whether transportation method is flight, self-driving, or taxi
Add restaurant price to spent
Add accommodation price to spent
Assert current spent is within budget

——EXAMPLES 3——
Natural Language query:
Please create a 3-day travel itinerary for 2 people beginning in Fort Lauderdale and ending in

Milwaukee from the 8th to the 10th of March, 2022. Our travel budget is set at $1,100. We'd love to experience both American and Chinese cuisines during our journey. We'd love to live in private rooms but we don't want to take flights.
Steps:
Destination cities
Set cities to be a list includes Milwaukee only
Loop through cities for 1 destination cities
Initialize Z3 solver s
Set 'city' variable to be indexes of 1 destination cities
Assert 'city' variable equal to city index

Departure dates
Set 'departure_dates' variables for 2 transportations between cities
Assert first transportation happens at first day (day 0), last transportation happens at last day (day 2)

Transportation methods
Set transportation method (flight, self-driving, taxi) variable for 2 transportations between cities
Assert only one of flight, self-driving, or taxi is used for 2 transportations between cities, self-driving is not valid if taxi or flight is used for any transportation
Assert all 2 transportations between cities are not flight

Flight information
Run FlightSearch to get flight info for Fort Lauderdale as origin, list of cities, city_0, and dates ["2022-03-08", "2022-03-09", "2022-03-10"]
Get specific flight price info with Fort Lauderdale as origin and final destination, specific city, and departure date for 2 transportations
Set 'flight_index' variable for 2 transportations
Assert 2 'flight_index' variables are within valid range if taking flight, assert flight index to be -1 if not taking flight
Calculate flight price for 2 people for 2 transportations based on flight index variable
Get specific flight arrival time info with Fort Lauderdale as origin and final destination, specific city, and departure date for 2 transportations
Calculate flight arrival time for 2 transportations based on flight index variable

Driving information
Run DistanceSearch to get driving info for Fort Lauderdale as origin and city_0
Get specific driving distance info with Fort Lauderdale as origin and final destination, specific city, and departure date for 2 transportations
Assert driving info is not empty if driving
Calculate self-driving and taxi price for 2 people and 2 transportations based on driving distance
Get driving arrival time with Fort Lauderdale as origin and final destination, specific city, and departure date for 2 transportations

Restaurant information
Get arrivals and city list for each day based on 2 transportations, 3 total travel day, and departure dates variables
Run RestaurantSearch to get restaurant price info and cuisine info for city_0
Set 'restaurant_in_which_city' variables for 9 (3 meals per day, 3 days) meals
For each 'restaurant_in_which_city' variable, assert it to be either current city or next city based on transportation arrivals time

Set 'restaurant_index' variables for 9 (3 meals per day, 3 days) meals
For each 'restaurant_index', get specific price info based on 'restaurant_in_which_city' variable, assert index are within valid range, assert restaurants in same city are not repeated, and calculate restaurant price for 2 people
Set 'cuisine_type' variables for each cuisine type ["American", "Chinese"] required
For each cuisine type, iterate through all restaurant to check if it is satisfied

Attraction information
Run AttractionSearch to get attraction info for city_0
Set 'attraction_in_which_city' variables for 3 (1 per day) attractions
For each 'attraction_in_which_city' variable, assert it to be either current city or next city based on transportation arrivals time
Set 'attraction_index' variables for 3 (1 per day) attractions
For each 'attraction_index', get specific length info based on attraction in which city variable, assert index are within valid range, and attrations in same city are not repeated

Accommodation information
Run AccommodationSearch to get accommodation info and accommodation constraints for city_0
Set 'accommodation_index' variables for 1 (1 per city) accommodations
For each 'accommodation_index', get specific price info based on accommodation in which city variable, assert 'accommodation_index' variable are within valid range, calculate number of room need for 2 people and accommodation price
For each city, get accommodation minimum night info and assert it to be less than the days stay in this city
For each 'accommodation_index', get specific room type info, assert 'Private room' exist for all accommodations

Budget
Set budget limit variable to be 1100
Add 2 transportation price to spent, according to whether transportation method is flight, self-driving, or taxi
Add restaurant price to spent
Add accommodation price to spent
Assert current spent is within budget
——EXAMPLES END——
Based on the examples above, give the steps for following Natural Language query. Follow the original step structures.
Note to keep the format in examples and start each line containing steps with '#'
Natural Language query:

G.1.2 Step-Code prompt

The step to code example prompt for each constraint type is provided as follows:
Destination cities:

```
# Python script for testing satisfiability of the destination cities constraint of a travel plan problem.

# Run CitySearch to get all possible destination cities in Minnesota State from origin 'Atlanta', remove origin 'Atlanta' if it is
# in list
cities = CitySearch.run('Minnesota', 'Atlanta', query_json['date'])
if 'Atlanta' in cities:
        cities.remove('Atlanta')
# Set cities to be a list includes Milwaukee only
cities = ['Milwaukee']
# Loop through cities for 2 destination cities
```

```
for city_0_index, city_0 in enumerate(cities):
    for city_1_index, city_1 in enumerate(cities):
        # Initialize Z3 solver s
        s = Optimize()
        # Set 'city' variable to be indexes of 2 destination cities
        variables['city'] = [Int('city_' + str(i)) for i in range(2)]
        # If city_0_index and city_1_index are not same, assert 2 'city' variables equal to city index
        if city_0_index != city_1_index:
        s.assert_and_track(variables['city'][0] == city_0_index,  'visit city in cities list')
        s.assert_and_track(variables['city'][1] == city_1_index,  'visit city in cities list')
# Loop through cities for 1 destination cities
for city_0_index, city_0 in enumerate(cities):
    # Initialize Z3 solver s
    s = Optimize()
    # Set 'city' variable to be indexes of 1 destination cities
    variables['city'] = [Int('city_' + str(i)) for i in range(1)]
    # Assert 'city' variable equal to city index
    s.assert_and_track(variables['city'][0] == city_0_index,  'visit city in cities list')

# Based on the examples above, in which the lines start with '#' is the instuction, where the line/lines below it before the
# next '#' is the corresponding code.
# For this below instruction, write corresponding code and respond instruction with code only. Start with ########## Destination
# cities response########## and end with ########## Destination cities response ends##########.
```

Departure Dates:

```
# Python script for testing satisfiability of the departure dates constraint of a travel plan problem.

# Set 'departure_dates' variables for 3 transportations between cities
variables['departure_dates'] = [Int('departure_dates_transportation_' + str(i)) for i in range(3)]
# Assert first transportation happens at first day (day 0), last transportation happens at last day (day 6), second and third
# transportation happen in between but not at the same day
s.assert_and_track(variables['departure_dates'][0] == 0,  'travel start date')
s.assert_and_track(And(variables['departure_dates'][1] > 0, variables['departure_dates'][1] < variables['departure_dates'][2]),
↪  'valid travel date')
s.assert_and_track(And(variables['departure_dates'][2] > variables['departure_dates'][1], variables['departure_dates'][1] < 6),
↪  'valid travel date')
s.assert_and_track(variables['departure_dates'][3] == 6,  'travel end date')
# Assert first transportation happens at first day (day 0), last transportation happens at last day (day 2)
s.assert_and_track(variables['departure_dates'][0] == 0,  'travel start date')
s.assert_and_track(variables['departure_dates'][2] == 2,  'travel end date')

# Based on the examples above, in which the lines start with '#' is the instuction, where the line/lines below it before the next
# '#' is the corresponding code.
# For this below instruction, write corresponding code and respond instruction with code only. Start with ########## Departure
# dates response########## and end with ########## Departure dates response ends##########.
```

Transportation Methods:

```
# Python script for testing satisfiability of the transportation methods constraint of a travel plan problem.

# Set transportation method variable (flight, self-driving, taxi) for 3 transportations between cities
variables['flight'] = [Bool('flight_travel_' + str(i)) for i in range(3)]
variables['self-driving'] = [Bool('self-driving_travel_' + str(i)) for i in range(3)]
variables['taxi'] = [Bool('taxi_travel_' + str(i)) for i in range(3)]
# Assert only one of flight, self-driving, or taxi is used for 3 transportations between cities, self-driving is not valid if taxi
# or flight is used for any transportation
s.assert_and_track(Or(variables['flight'][0], variables['self-driving'][0], variables['taxi'][0]),  'either flight, self-driving,
↪  or taxi for first transportation')
s.assert_and_track(Or(variables['flight'][1], variables['self-driving'][1], variables['taxi'][1]),  'either flight, self-driving,
↪  or taxi for second transportation')
s.assert_and_track(Or(variables['flight'][2], variables['self-driving'][2], variables['taxi'][2]),  'either flight, self-driving,
↪  or taxi for third transportation')
s.assert_and_track(Not(Or(And(variables['flight'][0], variables['self-driving'][0]), And(variables['flight'][0],
↪  variables['taxi'][0]), And(variables['taxi'][0], variables['self-driving'][0]))),  'flight, self-driving, and taxi not
↪  simutaneously for first transportation')
s.assert_and_track(Not(Or(And(variables['flight'][1], variables['self-driving'][1]), And(variables['flight'][1],
↪  variables['taxi'][1]), And(variables['taxi'][1], variables['self-driving'][1]))),  'flight, self-driving, and taxi not
↪  simutaneously for second transportation')
s.assert_and_track(Not(Or(And(variables['flight'][2], variables['self-driving'][2]), And(variables['flight'][2],
↪  variables['taxi'][2]), And(variables['taxi'][2], variables['self-driving'][2]))),  'flight, self-driving, and taxi not
↪  simutaneously for third transportation')
s.assert_and_track(Implies(Or(variables['flight'][0], variables['flight'][1], variables['flight'][2]),
↪  Not(Or(variables['self-driving'][0], variables['self-driving'][1], variables['self-driving'][2]))), 'no self-driving if taken
↪  flight for any transportation')
s.assert_and_track(Implies(Or(variables['taxi'][0], variables['taxi'][1], variables['taxi'][2]),
↪  Not(Or(variables['self-driving'][0], variables['self-driving'][1], variables['self-driving'][2]))), 'no self-driving if taken
↪  taxi for any transportation')
# Assert all 3 transportations between cities are not self-driving
s.assert_and_track(Not(variables['self-driving'][0]), 'no self-driving for first transportation')
s.assert_and_track(Not(variables['self-driving'][1]), 'no self-driving for second transportation')
s.assert_and_track(Not(variables['self-driving'][2]), 'no self-driving for third transportation')
```

```
# Based on the examples above, in which the lines start with '#' is the instuction, where the line/lines below it before the next
# '#' is the corresponding code.
# For this below instruction, write corresponding code and respond instruction with code only. Start with ########## Transportation
# response########## and end with ########## Transportation response ends##########.
```

Flight Information:

```
# Python script for testing satisfiability of the flight constraint constraint of a travel plan problem.

# Run FlightSearch to get flight info for Atlanta as origin, list of cities, city_0 and city_1, and dates
flight_info = FlightSearch.run_for_all_cities_and_dates('Atlanta', cities, [city_0, city_1], query_json['date'])
# Get specific flight price info with Atlanta as origin and final destination, specific city variable, and departure date for 3
# transportations
flight_0_price_list, flight_0_price_list_length = FlightSearch.get_info(flight_info, 'Atlanta', variables['city'][0],
↪  variables['departure_dates'][0], 'Price')
flight_1_price_list, flight_1_price_list_length = FlightSearch.get_info(flight_info, variables['city'][0], variables['city'][1],
↪  variables['departure_dates'][1], 'Price')
flight_2_price_list, flight_2_price_list_length = FlightSearch.get_info(flight_info, variables['city'][1], 'Atlanta',
↪  variables['departure_dates'][2], 'Price')
# Set 'flight_index' variable for 3 transportations
variables['flight_index'] = [Int('flight_{}_index'.format(i)) for i in range(3)]
# Assert 3 'flight_index' variables are within valid range if taking flight, assert flight index to be -1 if not taking flight
s.assert_and_track(Implies(variables['flight'][0], And(variables['flight_index'][0] >= 0,variables['flight_index'][0] <
↪  flight_0_price_list_length)), 'valid flight index for flight 0')
s.assert_and_track(Implies(variables['flight'][1], And(variables['flight_index'][1] >= 0,variables['flight_index'][1] <
↪  flight_1_price_list_length)), 'valid flight index for flight 1')
s.assert_and_track(Implies(variables['flight'][2], And(variables['flight_index'][2] >= 0,variables['flight_index'][2] <
↪  flight_2_price_list_length)), 'valid flight index for flight 2')
s.assert_and_track(Implies(Not(variables['flight'][0]), variables['flight_index'][0] == -1), 'valid flight index for flight 0')
s.assert_and_track(Implies(Not(variables['flight'][1]), variables['flight_index'][1] == -1), 'valid flight index for flight 1')
s.assert_and_track(Implies(Not(variables['flight'][2]), variables['flight_index'][2] == -1), 'valid flight index for flight 2')
# Calculate flight price for 2 people for 3 transportations based on flight index variable
flight_0_price = 2 * FlightSearch.get_info_for_index(flight_0_price_list, variables['flight_index'][0])
flight_1_price = 2 * FlightSearch.get_info_for_index(flight_1_price_list, variables['flight_index'][1])
flight_2_price = 2 * FlightSearch.get_info_for_index(flight_2_price_list, variables['flight_index'][2])
# Get specific flight arrival time info with Atlanta as origin and final destination, specific city, and departure date for 3
# transportations
flight_0_arrtime_list, _ = FlightSearch.get_info(flight_info, 'Atlanta', variables['city'][0], variables['departure_dates'][0],
↪  'ArrTime')
flight_1_arrtime_list, _ = FlightSearch.get_info(flight_info, variables['city'][0], variables['city'][1],
↪  variables['departure_dates'][1], 'ArrTime')
flight_2_arrtime_list, _ = FlightSearch.get_info(flight_info, variables['city'][1], 'Atlanta', variables['departure_dates'][2],
↪  'ArrTime')
# Calculate flight arrival time for 3 transportations based on flight index variable
flight_0_arrtime = FlightSearch.get_info_for_index(flight_0_arrtime_list, variables['flight_index'][0])
flight_1_arrtime = FlightSearch.get_info_for_index(flight_1_arrtime_list, variables['flight_index'][1])
flight_2_arrtime = FlightSearch.get_info_for_index(flight_2_arrtime_list, variables['flight_index'][2])

# Based on the examples above, in which the lines start with '#' is the instuction, where the line/lines below it before the next
# '#' is the corresponding code.
# For this below instruction, write corresponding code and respond instruction with code only. Start with ########## Flight
# response########## and end with ########## Flight response ends##########.
```

Driving Information:

```
# Python script for testing satisfiability of the driving constraint of a travel plan problem.

# Run DistanceSearch to get driving info for Atlanta as origin and city_0 and city_1
driving_info = DistanceSearch.run_for_all_cities('Atlanta', cities, [city_0, city_1])
# Get specific driving distance info with Atlanta as origin and final destination, specific city, and departure date for 3
# transportations
driving_0_distance, driving_0_length = DistanceSearch.get_info(driving_info, 'Atlanta', variables['city'][0], 'Distance')
driving_1_distance, driving_1_length = DistanceSearch.get_info(driving_info, variables['city'][0], variables['city'][1],
↪  'Distance')
driving_2_distance, driving_2_length = DistanceSearch.get_info(driving_info, variables['city'][1],'Atlanta', 'Distance')
# Assert driving info is not empty if driving
s.assert_and_track(Implies(Or(variables['self-driving'][0], variables['taxi'][0]), driving_0_length > 0), 'driving is possible for
↪  transportation 0')
s.assert_and_track(Implies(Or(variables['self-driving'][1], variables['taxi'][1]), driving_1_length > 0), 'driving is possible for
↪  transportation 1')
s.assert_and_track(Implies(Or(variables['self-driving'][2], variables['taxi'][2]), driving_2_length > 0), 'driving is possible for
↪  transportation 2')
# Calculate self-driving and taxi price for 3 people and 3 transportations based on driving distance
self_driving_0_price = 0.05 * driving_0_distance * math.ceil(3 / 5)
self_driving_1_price = 0.05 * driving_1_distance * math.ceil(3 / 5)
self_driving_2_price = 0.05 * driving_2_distance * math.ceil(3 / 5)
taxi_0_price = driving_0_distance * math.ceil(3 / 4)
taxi_1_price = driving_1_distance * math.ceil(3 / 4)
taxi_2_price = driving_2_distance * math.ceil(3 / 4)
# Get driving arrival time with Atlanta as origin and final destination, specific city, and departure date for 3 transportations
driving_0_arrtime, _ = DistanceSearch.get_info(driving_info, 'Atlanta', variables['city'][0], 'Duration')
driving_1_arrtime, _ = DistanceSearch.get_info(driving_info, variables['city'][0], variables['city'][1], 'Duration')
```

```
driving_2_arrtime, _ = DistanceSearch.get_info(driving_info, variables['city'][1], 'Atlanta', 'Duration')

# Based on the examples above, in which the lines start with '#' is the instuction, where the line/lines below it before the next
# '#' is the corresponding code.
# Follow the variable names in examples.
# For this below instruction, write corresponding code and respond instruction with code only. Start with ########## Driving
# response########## and end with ########## Driving response ends##########.
```

Restaurant Information:

```
# Python script for testing satisfiability of the restaurant constraint of a travel plan problem.

# Get arrivals and city list for each day based on 3 transportations, 5 total travel day, and departure dates variables
transportation_0_arrtime = If(variables['flight'][0], flight_0_arrtime, driving_0_arrtime)
transportation_1_arrtime = If(variables['flight'][1], flight_1_arrtime, driving_1_arrtime)
transportation_2_arrtime = If(variables['flight'][2], flight_2_arrtime, driving_2_arrtime)
arrives = get_arrivals_list([transportation_0_arrtime, transportation_1_arrtime, transportation_2_arrtime], 5,
↪  variables['departure_dates'])
city_list = get_city_list(variables['city'], 5, variables['departure_dates'])
# Run RestaurantSearch to get restaurant price info and cuisine info for city_0 and city_1
restaurant_price, restaurant_cuisines = RestaurantSearch.run_for_all_cities(cities, [city_0, city_1])
# Run RestaurantSearch to get restaurant price info and cuisine info for city_0
restaurant_price, restaurant_cuisines = RestaurantSearch.run_for_all_cities(cities, [city_0])
# Set 'restaurant_in_which_city' variables for 15 (3 meals per day, 5 days) meals
variables['restaurant_in_which_city'] = [Int('restaurant_' + str(i)) for i in range(3*5)]
# For each 'restaurant_in_which_city' variable, assert it to be either current city or next city based on transportation arrivals
# time
for i, variable in enumerate(variables['restaurant_in_which_city']):
    date_index = i // 3
    meal_index = i % 3
    if meal_index == 0: # breakfast
        s.assert_and_track(Or(variable == city_list[date_index], variable == city_list[date_index+1]),  'eat in which city b')
        s.assert_and_track(Implies(arrives[date_index]> 10, variable == city_list[date_index]),'eat in which city b')
        s.assert_and_track(Implies(arrives[date_index]< 5, variable == city_list[date_index+1]),'eat in which city b')
    if meal_index == 1: # lunch
        s.assert_and_track(Or(variable == city_list[date_index], variable == city_list[date_index+1]),  'eat in which city l')
        s.assert_and_track(Implies(arrives[date_index]> 15, variable == city_list[date_index]),'eat in which city l')
        s.assert_and_track(Implies(arrives[date_index]< 10, variable == city_list[date_index+1]),'eat in which city l')
    if meal_index == 2: # dinner
        s.assert_and_track(Or(variable == city_list[date_index], variable == city_list[date_index+1]),  'eat in which city d')
        s.assert_and_track(Implies(arrives[date_index]> 22, variable == city_list[date_index]),'eat in which city d')
        s.assert_and_track(Implies(arrives[date_index]< 17, variable == city_list[date_index+1]),'eat in which city d')
# Set 'restaurant_index' variables for 15 (3 meals per day, 5 days) meals
variables['restaurant_index'] = [Int('restaurant_{}_index'.format(i)) for i in range(3*5)]
# For each 'restaurant_index', get specific price info based on 'restaurant_in_which_city' variable, assert index are within valid
# range, assert restaurants in same city are not repeated, and calculate restaurant price for 2 people
all_restaurant_price = 0
for i, variable in enumerate(variables['restaurant_index']):
    restaurant_price_list, restaurant_list_length = RestaurantSearch.get_info(restaurant_price,
    ↪  variables['restaurant_in_which_city'][i], 'Price')
    s.assert_and_track(Implies(variables['restaurant_in_which_city'][i] != -1, And(variable >= 0, variable <
    ↪  restaurant_list_length)), 'valid restaurant index')
    s.assert_and_track(Implies(variables['restaurant_in_which_city'][i] == -1, variable == -1), 'valid restaurant index')
    for j in range(i-1, -1, -1):
        s.assert_and_track(Implies(And(variables['restaurant_in_which_city'][i] != -1, variables['restaurant_in_which_city'][i] ==
        ↪  variables['restaurant_in_which_city'][j]), variable != variables['restaurant_index'][j]), 'non repeating restaurant
        ↪  index')
    Calculate restaurant price based on restaurant index
    all_restaurant_price += 2 * If(variables['restaurant_in_which_city'][i] != -1,
    ↪  RestaurantSearch.get_info_for_index(restaurant_price_list, variable), 0)
# Set 'cuisine_type' variables for each cuisine type required
variables['cuisines_type'] = [Int('cuisines_' + i) for i in query_json['local_constraint']['cuisine']]
# For each cuisine type, iterate through all restaurant to check if it is satisfied
for index, cuisine in enumerate(query_json['local_constraint']['cuisine']):
    count = 0
    for i, variable in enumerate(variables['restaurant_index']):
        restaurant_cuisines_list, _ = RestaurantSearch.get_info(restaurant_cuisines, variables['restaurant_in_which_city'][i],
        ↪  'Cuisines')
        count += If(RestaurantSearch.check_exists(cuisine, restaurant_cuisines_list, variable), 1, 0)
    s.assert_and_track(variables['cuisines_type'][index] == count,  cuisine + 'type restaurant')
    s.assert_and_track(variables['cuisines_type'][index] > 0,  cuisine + 'type restaurant is visited')

# Based on the examples above, in which the lines start with '#' is the instuction, where the line/lines below it before the next
# '#' is the corresponding code.
# For this below instruction, write corresponding code and respond instruction with code only. Start with ########## Restaurant
# response########## and end with ########## Restaurant response ends##########.
```

Attraction Information:

```
# Python script for testing satisfiability of the attraction constraint of a travel plan problem.

# Run AttractionSearch to get attraction info for city_0 and city_1
attraction_info = AttractionSearch.run_for_all_cities(cities, [city_0, city_1])
```

```
# Run AttractionSearch to get attraction info for city_0
attraction_info = AttractionSearch.run_for_all_cities(cities, [city_0])
# Set 'attraction_in_which_city' variables for 5 (1 per day) attractions
variables['attraction_in_which_city'] = [Int('attraction_' + str(i)) for i in range(1*5)]
# For each 'attraction_in_which_city' variable, assert it to be either current city or next city based on transportation arrivals
# time
for i, variable in enumerate(variables['attraction_in_which_city']):
    s.assert_and_track(variable == If(arrives[i]> 18, city_list[i], city_list[i+1]),  'attraction in which city')
# Set 'attraction_index' variables for 5 (1 per day) attractions
variables['attraction_index'] = [Int('attraction_{}_index'.format(i)) for i in range(1*5)]
# For each 'attraction_index', get specific length info based on attraction in which city variable, assert index are within valid
# range, and attrations in same city are not repeated
for i, variable in enumerate(variables['attraction_index']):
    attraction_list_length = AttractionSearch.get_info(attraction_info, variables['attraction_in_which_city'][i])
    s.assert_and_track(Implies(variables['attraction_in_which_city'][i] != -1, And(variable >= 0, variable <
    ↪  attraction_list_length)), 'valid attraction index')
    s.assert_and_track(Implies(variables['attraction_in_which_city'][i] == -1, variable == -1), 'valid attraction index')
    for j in range(i-1, -1, -1):
        s.assert_and_track(Implies(And(variables['attraction_in_which_city'][i] != -1, variables['attraction_in_which_city'][i] ==
        ↪  variables['attraction_in_which_city'][j]), variable != variables['attraction_index'][j]), 'non repeating attraction
        ↪  index')

# Based on the examples above, in which the lines start with '#' is the instuction, where the line/lines below it before the next
# '#' is the corresponding code.
# For this below instruction, write corresponding code and respond instruction with code only. Start with ########## Attraction
# response########## and end with ########## Attraction response ends##########.
```

Accommodation Information:

```
# Python script for testing satisfiability of the accommodation constraint of a travel plan problem.

# Run AccommodationSearch to get accommodation info and accommodation constraints for city_0 and city_1
accommodation_info, accommodation_constraints = AccommodationSearch.run_for_all_cities(cities, [city_0, city_1])
# Run AccommodationSearch to get accommodation info and accommodation constraints for city_0
accommodation_info, accommodation_constraints = AccommodationSearch.run_for_all_cities(cities, [city_0])
# Set 'accommodation_index' variables for 2 (1 per city) accommodations
variables['accommodation_index'] = [Int('accommodation_{}_index'.format(i)) for i in range(2)]
# For each 'accommodation_index', get specific price info based on accommodation in which city variable, assert
# 'accommodation_index' variable are within valid range, calculate number of room need for 2 people and accommodation price
all_accommodation_price = 0
for i, variable in enumerate(variables['accommodation_index']):
    accommodation_price_list, accommodation_list_length = AccommodationSearch.get_info(accommodation_info, variables['city'][i],
    ↪  'Price')
    s.assert_and_track(And(variable >= 0, variable < accommodation_list_length), 'valid accomodation index')
    accommodation_maximum_occupancy_list, _ = AccommodationSearch.get_info(accommodation_info, variables['city'][i],
    ↪  'Maximum_occupancy')
    num_room = convert_to_int(RealVal(2) / AccommodationSearch.get_info_for_index(accommodation_maximum_occupancy_list, variable))
    all_accommodation_price += (variables['departure_dates'][i+1] - variables['departure_dates'][i]) * num_room *
    ↪  AccommodationSearch.get_info_for_index(accommodation_price_list, variable)
# For each city, get accommodation minimum night info and assert it to be less than the days stay in this city
for index, city in enumerate(variables['city']):
    accommodation_minimum_nights_list, _ = AccommodationSearch.get_info(accommodation_info, city, 'Minimum_nights')
    minimum_night = AccommodationSearch.get_info_for_index(accommodation_minimum_nights_list,
    ↪  variables['accommodation_index'][index])
    s.assert_and_track(minimum_night <= variables['departure_dates'][index+1]- variables['departure_dates'][index], 'minimum
    ↪  nights satisfied')
# For each 'accommodation_index', get specific room type and house rules info, assert 'Entire home/apt' exist for all
# accommodations, assert 'No parties' does not exist for all accommodations
for i, variable in enumerate(variables['accommodation_index']):
        accommodation_room_types_list, _ = AccommodationSearch.get_info(accommodation_constraints, variables['city'][i],
        ↪  'Room_types')
        accommodation_house_rules_list, _ = AccommodationSearch.get_info(accommodation_constraints, variables['city'][i],
        ↪  'House_rules')
        s.assert_and_track(AccommodationSearch.check_exists('Entire home/apt', accommodation_room_types_list, variable) == True,
        ↪  'Entire home/apt' + 'types accomadation visited')
        s.assert_and_track(AccommodationSearch.check_exists('No parties', accommodation_house_rules_list, variable) == False,  'No
        ↪  parties' + 'rules accomadation not visited')

# Based on the examples above, in which the lines start with '#' is the instuction, where the line/lines below it before the next
# '#' is the corresponding code.
# For this below instruction, write corresponding code and respond instruction with code only. Start with ########## Accommodation
# response########## and end with ########## Accommodation response ends##########.
```

Budget:

```
# Python script for testing satisfiability of the budget constraint of a travel plan problem.

# Set budget limit variable to be 7900
variables['budget_limit'] = RealVal(7900)
# Add 3 transportation price to spent, according to whether transportation method is flight, self-driving, or taxi
spent = 0
```

```
spent += If(variables['flight'][0], flight_0_price, If(variables['self-driving'][0], self_driving_0_price,
↪  If(variables['taxi'][0], taxi_0_price, 10000)))
spent += If(variables['flight'][1], flight_1_price, If(variables['self-driving'][1], self_driving_1_price,
↪  If(variables['taxi'][1], taxi_1_price, 10000)))
spent += If(variables['flight'][2], flight_2_price, If(variables['self-driving'][2], self_driving_2_price,
↪  If(variables['taxi'][2], taxi_2_price, 10000)))
# Add restaurant price to spent
spent += all_restaurant_price
# Add accommodation price to spent
spent += all_accommodation_price
# Assert current spent is within budget
s.assert_and_track(spent <= variables['budget_limit'], 'budget enough')

# Based on the examples above, in which the lines start with '#' is the instuction, where the line/lines below it before the next
# '#' is the corresponding code.
# For this below instruction, write corresponding code and respond instruction with code only. Start with ########## Budget
# response########## and end with ########## Budget response ends##########.
```

G.1.3 Prompt difference of GPT-4, Claude 3, and Mistral-Large

With the prompt we have for GPT-4 as the starting point, we adjust the prompts (add more explainations or examples) for Claude-3 and Mistral-Large using the training set in TravelPlanner.

Claude-3 almost has the same prompt as GPT-4, except for the JSON-Step prompt. Since in training set, a failure case for Claude-3 is it is not able to handle the "house rule" properly. When the JSON specifies "house rule" to be "children under 10" it means the travellers have children under 10 and would like to stay in accommodations without "No children under 10" rule. While Claude-3 sometimes is not able to give "No children under 10" in the step, instead, it gives steps with "children under 10 not allowed". To enable it to handle this, we add one sentence explanation **"if house rule 'xxx' is mentioned, then 'No xxx' should not exist for all accomadations."** in JSON-Step prompt.

Compared to Claude-3, Mistral-Large needs more prompt adjustment:

- We add **"You can only assign null to local constraints if it is needed. Other fields must have values."** to NL-JSON prompt because Mistral-Large sometimes misses some information in JSON translation.
- Claude-3 uses the same JSON-Step prompt as Mistral-Large.
- We add a 3-city loop-through-cities example in Destination Cities Step-Code prompt; We add a 2-city travel-date-assertion example in Departure Dates prompt; We add a 1-city transportation-method assertion-example to Transportation Methods; We add instructions that ask LLM to not use for-loops and name variable with "i" as when it tries to iteratively create or access variables with i it fails to write the correct code.

From the amount of changes we need to make, we can observe that Mistral-Large in general produces more code generation errors compared to GPT-4 and Claude-3, thus needing more examples and explanations provided in prompts.

G.1.4 Failure cases of Mistral-Large

Although we tune our prompt with training set, there are still some failure cases that do not appear in training set and thus negatively affect Mistral-Large's delivery rate.
The major failure mode is: "room type" takes the value "no shared room". This room type is special in that when other room types such as "private room" is specified, the generated instruction steps should be "private room exists for all accommodations". However, only when "no shared room" is mentioned, the steps should be "shared room does not exist for all accommodations." Since "no shared room" does not appear in training set or examples, and Mistral-Large is not able to generalize to it, it fails by producing "not shard room exists for all accommodations", thus fails to search for home with type "not shared room". This is the major failure mode and is responsible for 7.8% of failed delivered plans (15.0% in total). Other than this, the failures are induced by runtime issue or some occasional code generation errors.

G.2 Prompts for Interactive Plan Repair

G.2.1 Suggestion prompt

The instruction prompt that guides LLM to collect information, analyze current situation, and offer suggestions for unsatisfiable queries is provided as follows:
Suggestion prompt for UnsatChristmas:

> As a travel planner, you have some constraints in JSON format to satisfy for a round trip travel plan problem.
> The trip spans "date", goes from "org", travels "dest" cities in a row in between, and then goes back to "org"
> For example, if "org" is city_0, and "dest" is [city_1, city_2], then the flights could be [city_0->city_1,city_1->city_2,city_2->city_0] or [city_0->city_2,city_2-> city_1,city_1->city_0]
> In addition, "local_constraint" contains three possible constraints. "flight rule" specifies whether "nonstop" is required or not. "airlines" specifies a list of a airlines user can accept. Possible options of "attraction_category" specifies a list of categories of attractions want to visit. If the field value is null in JSON, this specific hard constraint is not included.
> The specified "flight rule" needs to be satisfied by all flights. All flights need to be one of the accepted "airlines". All categories in "attraction_category" needs to be satisfied, and a category could be satisfied if it exists for one attraction.
> After anaylzed these constraints, you found they are not satisfiable under current setting. You will be giving unsatisfiable reasons.
> Collect information based on the reasons or, based on the information you collect, analyze current situation or give a suggested modification to the constraints.
> Info collecting can take 4 different actions:
> (1) FlightSearch[Departure City, Destination City]: Description: A flight information retrieval tool.
> Parameters:
> Departure City: The city you'll be flying out from.
> Destination City: The city you aim to reach.
> Example: FlightSearch[New York City, London] would fetch flights from New York City to London.
>
> (2) AirlineSearch[Airline]:
> Description: Find flights of input airline.
> Parameter: Airline - The airline name you want to take.
> Example: AirlineSearch[United] would return all flights of United airline.
>
> (3) AttractionSearch[City]:
> Description: Find attractions categories in a city of your choice.
> Parameter: City - The name of the city where you're seeking attractions.
> Example: AttractionSearch[London] would return attraction categories in London.
>
> (4) CategorySearch[Category]:
> Description: Find cities contain attractions of input category.
> Parameter: Category - The attraction category where you want to visit.
> Example: CategorySearch[Park] would return all cities where attractions of category Park exist.
>
> You need to take an action analyze current situation and plan your future steps after each FlightSearch, AirlineSearch, AttractionSearch, or CategorySearch.
> Example: Analyze[your analysis of current situation and plan for future]

You can suggest to remove the non-stop constraint, suggest to change required airlines, suggest to change destination cities(but keep number of destination cities unchanged), suggest to change attraction categories, or suggest to raise budget. Do not give other suggestions that change other fields in JSON input, such as origin, number of visit cities, etc.
Please give a reasonable suggestion to modify the constraint only when you think you've collected enough information and the suggestion has high chance to be satisfiable. For example, if destination city does not have required attraction category, you should suggest to change destination city if info shows the new city has the required category
Please try to keep original constraint and make minimal change to original constraint only when it is necessary.
Examples:
Suggest[raise budget to 5000]
Suggest[change destination cities to be Istanbul and Macau]
Suggest[remove the non-stop constraint]
Suggest[change airlines to be United, Air France, or JetBlue]
Suggest[change attraction categories to be Garden and Museum]

A list of possible cities is ['Bangkok', 'Dubai', 'Hong Kong', 'Istanbul', 'Kuala Lumpur', 'London', 'Macau', 'New York City', 'Paris', 'Singapore']
Now, based on the input query, unsatisfiable reasons, and collected information, please give the next action(only one action) you want to take only with no explainations, you need to give a suggestion within 15 iterations:
Input query: {query}
Unsatisfiable reasons: {reasons}
Collected information: {info}

Suggestion prompt for TravelPlanner:

As a travel planner, you have some constraints in JSON format to satisfy for a round trip travel plan problem.
The trip spans "date", goes from "org", travels "dest" city, and then goes back to "org" For example, if "org" is city_0, and "dest" is city_1, then the transportations would be [city_0->city_1,city_1->city_0]
In addition, "local_constraint" contains four possible constraints. Possible options of "house rule" includes ["parties", "smoking", "children under 10", "pets", "visitors"]. Possible options of "cuisine" includes ["Chinese", "American", "Italian", "Mexican", "Indian", "Mediterranean", "French"]. Possible options of "house type" includes ["entire room", "private room", "shared room", "not shared room"]. Possible options of "transportation" includes ["no flight", "no self-driving"]. If the field value is null in JSON, this specific hard constraint is not included.
The specified "house rule" and "house type" needs to be satisfied by all accommodations. The specified "transportation" needs to be satisfied by all transportations. All cuisines in "cuisine" needs to be satisfied, and a cuisine could be satisfied if it exists for one restaurant.
After anaylzed these constraints, you found they are not satisfiable under current setting. You will be giving unsatisfiable reasons.
Collect information based on the reasons or, based on the information you collect, analyze current situation or give a suggested modification to the constraints.
Info collecting can take 6 different actions:
(1) DrivingCheck[Departure City, Destination City]: Description: A driving information checking tool that checks if driving is feasible.
Parameters:

Departure City: The city you'll be driving out from.
Destination City: The city you aim to reach.
Example: DrivingCheck[Grand Forks, Minneapolis] would check if driving is feasible from Grand Forks to Minneapolis.

(2) DrivingSearch[Departure City]:
Description: A driving information retrieval tool that returns all reachable cities.
Parameters:
Departure City: The city you'll be driving out from.
Example: DrivingSearch[Grand Forks] would return all reachable cities from Grand Forks through driving.

(3) FlightCheck[Departure City, Destination City, Date]:
Description: A flight information checking tool that checks if flight is feasible.
Parameters:
Departure City: The city you'll be flying out from.
Destination City: The city you aim to reach.
Date: The date you take the flight.
Example: FlightCheck[Grand Forks, Minneapolis, 2022-06-05] would check if flight is feasible from Grand Forks to Minneapolis on 2022-06-05.

(4) FlightSearch[Departure City, Date]: Description: A flight information retrieval tool that returns all reachable cities.
Parameters:
Departure City: The city you'll be flying out from.
Date: The date you take the flight.
Example: FlightSearch[Grand Forks, 2022-06-05] would return all reachable cities from Grand Forks through flight on 2022-06-05.

(5) AccommodationSearch[City]:
Description: Find accommodations types in a city of your choice.
Parameter: City - The name of the city where you're seeking accommodations.
Example: AccommodationSearch[Grand Forks] would return accommodation categories in Grand Forks.

(6) TypeSearch[Type]:
Description: Find cities contain accommodations of input type.
Parameter: Type - The accommodation type where you want to visit.
Example: TypeSearch[entire room] would return all cities where entire room type accommodation exist.

You need to take an action analyze current situation and plan your future steps after each DrivingCheck, DrivingSearch, FlightCheck, FlightSearch, AccommodationSearch, or TypeSearch.
Example: Analyze[your analysis of current situation and plan for future]

You can suggest to remove the "house type" constraint, suggest to remove the "transportation" constraint, suggest to change destination cities(but keep number of destination cities unchanged), or suggest to raise budget. Do not give other suggestions that change other fields in JSON input, such as origin, number of visit cities, etc.
Please give a reasonable suggestion to modify the constraint only when you think you've collected

enough information and the suggestion has high chance to be satisfiable. For example, if destination city does not have required accomadation type, you should suggest to change destination city if info shows the new city has the required type.
Please try to keep original constraint and make minimal change to original constraint only when it is necessary.
Examples:
Suggest[raise budget to 5000]
Suggest[change destination cities to be Minneapolis]
Suggest[remove the house type constraint]
Suggest[remove the flight/no flight/ no self-driving assertion for transportations]

Now, based on the input query, unsatisfiable reasons, and collected information, please give the next action(only one action) you want to take only with no explainations, you need to give a suggestion within 15 iterations:
Input query: {query}
Unsatisfiable reasons: {reasons}
Collected information: {info}

G.2.2 Suggestion-No Reason prompt

The Suggestion-No Reason prompt is basically modified from the Suggestion prompt by removing all descriptions about reasons.

G.2.3 Suggestion-No Sovler prompt

The instruction suggestion prompt that remove the iterative solver calling and directly guide LLM to offer a list of suggestions is provided as follows:
Suggestion-No Sovler prompt for UnsatChristmas:

As a travel planner, you have some constraints in JSON format to satisfy for a round trip travel plan problem.
The trip spans "date", goes from "org", travels "dest" cities in a row in between, and then goes back to "org" For example, if "org" is city_0, and "dest" is [city_1, city_2], then the flights could be [city_0->city_1,city_1->city_2,city_2->city_0] or [city_0->city_2,city_2-> city_1,city_1->city_0]
In addition, "local_constraint" contains three possible constraints. "flight rule" specifies whether "non-stop" is required or not. "airlines" specifies a list of a airlines user can accept. Possible options of "attraction_category" specifies a list of categories of attractions want to visit. If the field value is null in JSON, this specific hard constraint is not included.
The specified "flight rule" needs to be satisfied by all flights. All flights need to be one of the accepted "airlines". All categories in "attraction_category" needs to be satisfied, and a category could be satisfied if it exists for one attraction.
After anaylzed these constraints, you found they are not satisfiable under current setting.
Collect information or, based on the information you collect, analyze current situation or give a suggested modification to the constraints.
Info collecting can take 4 different actions:
(1) FlightSearch[Departure City, Destination City]:
Description: A flight information retrieval tool. Parameters:
Departure City: The city you'll be flying out from.
Destination City: The city you aim to reach.
Example: FlightSearch[New York City, London] would fetch flights from New York City to London.

(2) AirlineSearch[Airline]:
Description: Find flights of input airline.

Parameter: Airline - The airline name you want to take.
Example: AirlineSearch[United] would return all flights of United airline.

(3) AttractionSearch[City]:
Description: Find attractions categories in a city of your choice.
Parameter: City - The name of the city where you're seeking attractions.
Example: AttractionSearch[London] would return attraction categories in London.

(4) CategorySearch[Category]:
Description: Find cities contain attractions of input category.
Parameter: Category - The attraction category where you want to visit.
Example: CategorySearch[Park] would return all cities where attractions of category Park exist.

You need to take an action analyze current situation and plan your future steps after each FlightSearch, AirlineSearch, AttractionSearch, or CategorySearch.
Example: Analyze[your analysis of current situation and plan for future]

You can suggest to remove the non-stop constraint, suggest to change required airlines, suggest to change destination cities(but keep number of destination cities unchanged), suggest to change attraction categories, or suggest to raise budget. Do not give other suggestions that change other fields in JSON input, such as origin, number of visit cities, etc.
Please give reasonable suggestions to modify the constraint only when you think you've collected enough information and the suggestion has high chance to be satisfiable. For example, if destination city does not have required attraction category, you should suggest to change destination city if info shows the new city has the required category
Please try to keep original constraint and make minimal change to original constraint only when it is necessary.
You can give one or more suggestions if you think one is not enough. Please separate the suggestions with ;
Examples:
Suggest[raise budget to 5000]
Suggest[change destination cities to be Istanbul and Macau]
Suggest[remove the non-stop constraint]
Suggest[change airlines to be United, Air France, or JetBlue]
Suggest[change attraction categories to be Garden and Museum]
Suggest[raise budget to 3000; change destination cities to be London]

A list of possible cities is ['Bangkok', 'Dubai', 'Hong Kong', 'Istanbul', 'Kuala Lumpur', 'London', 'Macau', 'New York City', 'Paris', 'Singapore'] Now, based on the input query and collected information, please give the next action(only one action) you want to take only with no explainations, you need to give suggestions within 15 iterations:
Input query: {query}
Collected information: {info}

Suggestion-No Sovler prompt for UnsatChristmas:

As a travel planner, you have some constraints in JSON format to satisfy for a round trip travel plan problem.
The trip spans "date", goes from "org", travels "dest" city, and then goes back to "org" For example, if "org" is city_0, and "dest" is city_1, then the transportations would be [city_0->city_1,city_1-

>city_0]
In addition, "local_constraint" contains four possible constraints. Possible options of "house rule" includes ["parties", "smoking", "children under 10", "pets", "visitors"]. Possible options of "cuisine" includes ["Chinese", "American", "Italian", "Mexican", "Indian", "Mediterranean", "French"]. Possible options of "house type" includes ["entire room", "private room", "shared room", "not shared room"]. Possible options of "transportation" includes ["no flight", "no self-driving"]. If the field value is null in JSON, this specific hard constraint is not included. The specified "house rule" and "house type" needs to be satisfied by all accommodations. The specified "transportation" needs to be satisfied by all transportations. All cuisines in "cuisine" needs to be satisfied, and a cuisine could be satisfied if it exists for one restaurant.
After anaylzed these constraints, you found they are not satisfiable under current setting.
Collect information or, based on the information you collect, analyze current situation or give a suggested modification to the constraints.
Info collecting can take 6 different actions:
(1) DrivingCheck[Departure City, Destination City]: Description: A driving information checking tool that checks if driving is feasible.
Parameters:
Departure City: The city you'll be driving out from.
Destination City: The city you aim to reach.
Example: DrivingCheck[Grand Forks, Minneapolis]
would check if driving is feasible from Grand Forks to Minneapolis.

(2) DrivingSearch[Departure City]:
Description: A driving information retrieval tool that returns all reachable cities.
Parameters:
Departure City: The city you'll be driving out from.
Example: DrivingSearch[Grand Forks] would return all reachable cities from Grand Forks through driving.

(3) FlightCheck[Departure City, Destination City, Date]:
Description: A flight information checking tool that checks if flight is feasible.
Parameters:
Departure City: The city you'll be flying out from.
Destination City: The city you aim to reach. Date: The date you take the flight.
Example: FlightCheck[Grand Forks, Minneapolis, 2022-06-05] would check if flight is feasible from Grand Forks to Minneapolis on 2022-06-05.

(4) FlightSearch[Departure City, Date]:
Description: A flight information retrieval tool that returns all reachable cities.
Parameters:
Departure City: The city you'll be flying out from.
Date: The date you take the flight.
Example: FlightSearch[Grand Forks, 2022-06-05] would return all reachable cities from Grand Forks through flight on 2022-06-05.

(5) AccommodationSearch[City]:
Description: Find accommodations types in a city of your choice.
Parameter: City - The name of the city where you're seeking accommodations.
Example: AccommodationSearch[Grand Forks] would return accommodation categories in Grand Forks.

(6) TypeSearch[Type]:
Description: Find cities contain accommodations of input type.
Parameter: Type - The accommodation type where you want to visit.
Example: TypeSearch[entire room] would return all cities where entire room type accommodation exist.
You need to take an action analyze current situation and plan your future steps after each DrivingCheck, DrivingSearch, FlightCheck, FlightSearch, AccommodationSearch, or TypeSearch.
Example: Analyze[your analysis of current situation and plan for future]

You can suggest to remove the "house type" constraint, suggest to remove the "transportation" constraint, suggest to change destination cities(but keep number of destination cities unchanged), or suggest to raise budget. Do not give other suggestions that change other fields in JSON input, such as origin, number of visit cities, etc.
Please give a reasonable suggestion to modify the constraint only when you think you've collected enough information and the suggestion has high chance to be satisfiable. For example, if destination city does not have required accomadation type, you should suggest to change destination city if info shows the new city has the required type.
Please try to keep original constraint and make minimal change to original constraint only when it is necessary.
You can give one or more suggestions if you think one is not enough. Please separate the suggestions with ;
Examples:
Suggest[raise budget to 5000]
Suggest[change destination cities to be Minneapolis]
Suggest[remove the house type constraint]
Suggest[remove the flight/no flight/ no self-driving assertion for transportations]
Suggest[raise budget to 2000, change destination cities to be Chicago]

Now, based on the input query and collected information, please give the next action(only one action) you want to take only with no explainations, you need to give a suggestion within 15 iterations:
Input query: {query}
Collected information: {info}

G.2.4 Code modify prompt

As a travel planner, you have some python codes that tests the satisfiability of a travel plan problem. While now some of the constraints are changed, your task is to change the python codes according to the changed constraints.
Only change the part of code that needs to be modified, and do not add any new parts.
Please respond with codes only, and be sure to include full codes instead of lines of updated codes.
Start with ########## response ########## and end with ########## response ends ##########.

Original Codes: {codes}
Modified Constraints: {constraints}

G.3 Prompts for Generalization Evaluation

To test the capability of our framework to generalize to unseen constraint types, we add this task description and append at the end of Step-Code prompt:

Based on the examples above, can you give the steps for following JSON constraint with different fields. In the input JSON, "org" denotes the departure city. "dest" denotes the destination city/cities. "days" denotes the total number of travel days. When "days" equals 5 or 7. "date" includes the specific date to visit.
In addition, "local_constraint" contains three possible constraints. "flight rule" specifies the whether "non-stop" is required or not. "airlines" specifies a list of a airlines you can accept. Possible options of "attraction_category" specifies a list of categories of attractions want to visit. "transportation" is always 'flight'. If the field value is null in JSON, this specific hard constraint is not included.
The specified "flight rule" needs to be satisfied by all flights. All flights need to be one of the accepted "airlines". All categories in "attraction_category" needs to be satisfied, and a category could be satisfied if it exists for one attraction.
We do not consider restaurant information or accommodation information, instead, we fix each restaurant price to be 30 per person and accommodation to be 100 per person.

G.4 Paraphrased Prompt Examples

Here's an example paraphrased prompt for Query-Step:

You are provided with a natural language query that outlines specific constraints for a travel itinerary problem. These constraints may include details such as the departure city, destination state or city, total travel days and dates, and particular requirements related to accommodation, dining, and transportation.
Your task is to provide a comprehensive, step-by-step guide to translate these constraints into code. Below are some example steps for various constraints:
——EXAMPLE 1——
Natural Language query:
Can you create a 5-day travel itinerary for a group of 3, departing from Atlanta and visiting 2 cities in Minnesota from March 3rd to March 7th, 2022? We have a budget of $7,900. We require accommodations that allow parties and should ideally be entire rooms. Although we don't plan to self-drive, we would like the flexibility to host parties.
Steps:
Destination cities
Execute CitySearch to identify all potential destination cities in Minnesota State for the dates ["2022-03-16", "2022-03-17", "2022-03-18"] from the starting point 'Atlanta', and exclude 'Atlanta' if it appears in the list
Iterate over cities to select 2 destination cities
Initialize Z3 solver s
Define 'city' variable to represent indexes of 2 destination cities
Ensure city_0_index and city_1_index are distinct, and assert 2 'city' variables equal to city index

Departure dates
Define 'departure_dates' variables for 3 transportations between cities
Assert that the first transportation occurs on the first day (day 0), the last transportation on the last day (day 4), and the second transportation can occur on any day in between

Transportation methods
Define transportation method (flight, self-driving, taxi) variable for 3 transportations between cities
Assert that only one of flight, self-driving, or taxi is used for 3 transportations between cities, and self-driving is invalid if taxi or flight is used for any transportation
Assert that all 3 transportations between cities are not self-driving

Flight information
Execute FlightSearch to obtain flight info for Atlanta as the origin, list of cities, city_0 and city_1, and dates ["2022-03-16", "2022-03-17", "2022-03-18"]
Retrieve specific flight price info with Atlanta as the origin and final destination, specific city variable, and departure date for 3 transportations
Define 'flight_index' variable for 3 transportations
Assert that 3 'flight_index' variables are within a valid range if taking a flight, and set flight index to -1 if not taking a flight
Calculate flight price for 3 people for 3 transportations based on flight index variable
Retrieve specific flight arrival time info with Atlanta as the origin and final destination, specific city, and departure date for 3 transportations
Calculate flight arrival time for 3 transportations based on flight index variable

Driving information
Execute DistanceSearch to obtain driving info for Atlanta as the origin and city_0 and city_1
Retrieve specific driving distance info with Atlanta as the origin and final destination, specific city, and departure date for 3 transportations
Assert that driving info is not empty if driving
Calculate self-driving and taxi price for 3 people and 3 transportations based on driving distance
Retrieve driving arrival time with Atlanta as the origin and final destination, specific city, and departure date for 3 transportations

Restaurant information
Obtain arrivals and city list for each day based on 3 transportations, 5 total travel days, and departure dates variables
Execute RestaurantSearch to obtain restaurant price info and cuisine info for city_0 and city_1
Define 'restaurant_in_which_city' variables for 15 (3 meals per day, 5 days) meals
For each 'restaurant_in_which_city' variable, assert it to be either the current city or the next city based on transportation arrivals time
Define 'restaurant_index' variables for 15 (3 meals per day, 5 days) meals
For each 'restaurant_index', retrieve specific price info based on 'restaurant_in_which_city' variable, assert indexes are within a valid range, assert restaurants in the same city are not repeated, and calculate restaurant price for 3 people
Calculate restaurant price based on restaurant index

Attraction information
Execute AttractionSearch to obtain attraction info for city_0 and city_1
Define 'attraction_in_which_city' variables for 5 (1 per day) attractions
For each 'attraction_in_which_city' variable, assert it to be either the current city or the next city based on transportation arrivals time
Define 'attraction_index' variables for 5 (1 per day) attractions
For each 'attraction_index', retrieve specific length info based on attraction in which city variable, assert indexes are within a valid range, and attractions in the same city are not repeated

Accommodation information
Execute AccommodationSearch to obtain accommodation info and accommodation constraints for city_0 and city_1
Define 'accommodation_index' variables for 2 (1 per city) accommodations
For each 'accommodation_index', retrieve specific price info based on accommodation in which city variable, assert 'accommodation_index' variable is within a valid range, calculate the number

of rooms needed for 3 people and accommodation price
For each city, obtain accommodation minimum night info and assert it to be less than the days stayed in this city
For each 'accommodation_index', retrieve specific room type and house rules info, assert 'Entire home/apt' exists for all accommodations, assert 'No parties' does not exist for all accommodations

Budget
Set budget limit variable to be 7900
Add 3 transportation prices to spent, according to whether the transportation method is flight, self-driving, or taxi
Add restaurant price to spent
Add accommodation price to spent
Assert current spent is within budget

——EXAMPLE 2——
Natural Language query:
Can you help with generating a 7-day travel plan for a party of 5? We're setting off from Indianapolis and planning to explore 3 cities in Colorado from March 11th to March 17th, 2022. We have a budget of $15,100 for this trip. We'll be bringing our pets, so pet-friendly accommodations are a must. It's important for us to stay in accommodations that permit children under the age of 10. We're also hoping to find places that offer Mexican, Italian, Mediterranean, and Indian cuisines. No shared rooms for accommodations would be ideal. We do not have preferences for transportation.
Steps:
Destination cities
Execute CitySearch to identify all potential destination cities in Colorado State for the dates ['2022-03-11', '2022-03-12', '2022-03-13', '2022-03-14', '2022-03-15', '2022-03-16', '2022-03-17'] from the starting point 'Indianapolis', and exclude 'Indianapolis' if it appears in the list
Iterate over cities to select 3 destination cities
Initialize Z3 solver s
Define 'city' variable to represent indexes of 3 destination cities
Ensure city_0_index, city_1_index, city_2_index are distinct, and assert 3 'city' variables equal to city index

Departure dates
Define 'departure_dates' variables for 4 transportations between cities
Assert that the first transportation occurs on the first day (day 0), the last transportation on the last day (day 6), and the second and third transportations occur in between but not on the same day

Transportation methods
Define transportation method (flight, self-driving, taxi) variable for 4 transportations between cities
Assert that only one of flight, self-driving, or taxi is used for 4 transportations between cities, and self-driving is invalid if taxi or flight is used for any transportation

Flight information
Execute FlightSearch to obtain flight info for Indianapolis as the origin, list of cities, city_0, city_1, and city_2, and dates ['2022-03-11', '2022-03-12', '2022-03-13', '2022-03-14', '2022-03-15', '2022-03-16', '2022-03-17']
Retrieve specific flight price info with Indianapolis as the origin and final destination, specific

city variable, and departure date for 4 transportations
Define 'flight_index' variable for 4 transportations
Assert that 4 'flight_index' variables are within a valid range if taking a flight, and set flight index to -1 if not taking a flight
Calculate flight price for 5 people for 4 transportations based on flight index variable
Retrieve specific flight arrival time info with Indianapolis as the origin and final destination, specific city, and departure date for 4 transportations
Calculate flight arrival time for 4 transportations based on flight index variable

Driving information
Execute DistanceSearch to obtain driving info for Indianapolis as the origin and city_0, city_1, and city_2
Retrieve specific driving distance info with Indianapolis as the origin and final destination, specific city, and departure date for 4 transportations
Assert that driving info is not empty if driving
Calculate self-driving and taxi price for 5 people and 4 transportations based on driving distance
Retrieve driving arrival time with Indianapolis as the origin and final destination, specific city, and departure date for 4 transportations

Restaurant information
Obtain arrivals and city list for each day based on 4 transportations, 7 total travel days, and departure dates variables
Execute RestaurantSearch to obtain restaurant price info and cuisine info for city_0, city_1, and city_2
Define 'restaurant_in_which_city' variables for 21 (3 meals per day, 7 days) meals
For each 'restaurant_in_which_city' variable, assert it to be either the current city or the next city based on transportation arrivals time
Define 'restaurant_index' variables for 21 (3 meals per day, 7 days) meals
For each 'restaurant_index', retrieve specific price info based on 'restaurant_in_which_city' variable, assert indexes are within a valid range, assert restaurants in the same city are not repeated, and calculate restaurant price for 5 people
Define 'cuisine_type' variables for each cuisine type ['Mexican', 'Italian', 'Mediterranean', 'Indian'] required
For each cuisine type, iterate through all restaurants to check if it is satisfied

Attraction information
Execute AttractionSearch to obtain attraction info for city_0, city_1, and city_2
Define 'attraction_in_which_city' variables for 7 (1 per day) attractions
For each 'attraction_in_which_city' variable, assert it to be either the current city or the next city based on transportation arrivals time
Define 'attraction_index' variables for 7 (1 per day) attractions
For each 'attraction_index', retrieve specific length info based on attraction in which city variable, assert indexes are within a valid range, and attractions in the same city are not repeated

Accommodation information
Execute AccommodationSearch to obtain accommodation info and accommodation constraints for city_0, city_1, and city_2
Define 'accommodation_index' variables for 3 (1 per city) accommodations
For each 'accommodation_index', retrieve specific price info based on accommodation in which city variable, assert 'accommodation_index' variable is within a valid range, calculate the number

of rooms needed for 5 people and accommodation price
For each city, obtain accommodation minimum night info and assert it to be less than the days stayed in this city
For each 'accommodation_index', retrieve specific room type and house rules info, assert 'Shared room' does not exist for all accommodations, assert 'No pets' does not exist for all accommodations, assert 'No children under 10' does not exist for all accommodations

Budget
Set budget limit variable to be 15100
Add 4 transportation prices to spent, according to whether the transportation method is flight, self-driving, or taxi
Add restaurant price to spent
Add accommodation price to spent
Assert current spent is within budget

——EXAMPLE 3——
Natural Language query:
Please create a 3-day travel itinerary for 2 people beginning in Fort Lauderdale and ending in Milwaukee from the 8th to the 10th of March, 2022. Our travel budget is set at $1,100. We'd love to experience both American and Chinese cuisines during our journey. We'd love to live in private rooms but we don't want to take flights.
Steps:
Destination cities
Set cities to be a list that includes Milwaukee only
Iterate over cities for 1 destination city
Initialize Z3 solver s
Define 'city' variable to represent indexes of 1 destination city
Assert 'city' variable equals city index

Departure dates
Define 'departure_dates' variables for 2 transportations between cities
Assert that the first transportation occurs on the first day (day 0), and the last transportation on the last day (day 2)

Transportation methods
Define transportation method (flight, self-driving, taxi) variable for 2 transportations between cities
Assert that only one of flight, self-driving, or taxi is used for 2 transportations between cities, and self-driving is invalid if taxi or flight is used for any transportation
Assert that all 2 transportations between cities are not flights

Flight information
Execute FlightSearch to obtain flight info for Fort Lauderdale as the origin, list of cities, city_0, and dates ["2022-03-08", "2022-03-09", "2022-03-10"]
Retrieve specific flight price info with Fort Lauderdale as the origin and final destination, specific city, and departure date for 2 transportations
Define 'flight_index' variable for 2 transportations
Assert that 2 'flight_index' variables are within a valid range if taking a flight, and set flight index to -1 if not taking a flight
Calculate flight price for 2 people for 2 transportations based on flight index variable

Retrieve specific flight arrival time info with Fort Lauderdale as the origin and final destination, specific city, and departure date for 2 transportations
Calculate flight arrival time for 2 transportations based on flight index variable

Driving information
Execute DistanceSearch to obtain driving info for Fort Lauderdale as the origin and city_0
Retrieve specific driving distance info with Fort Lauderdale as the origin and final destination, specific city, and departure date for 2 transportations
Assert that driving info is not empty if driving
Calculate self-driving and taxi price for 2 people and 2 transportations based on driving distance
Retrieve driving arrival time with Fort Lauderdale as the origin and final destination, specific city, and departure date for 2 transportations

Restaurant information
Obtain arrivals and city list for each day based on 2 transportations, 3 total travel days, and departure dates variables
Execute RestaurantSearch to obtain restaurant price info and cuisine info for city_0
Define 'restaurant_in_which_city' variables for 9 (3 meals per day, 3 days) meals
For each 'restaurant_in_which_city' variable, assert it to be either the current city or the next city based on transportation arrivals time
Define 'restaurant_index' variables for 9 (3 meals per day, 3 days) meals
For each 'restaurant_index', retrieve specific price info based on 'restaurant_in_which_city' variable, assert indexes are within a valid range, assert restaurants in the same city are not repeated, and calculate restaurant price for 2 people
Define 'cuisine_type' variables for each cuisine type ["American", "Chinese"] required
For each cuisine type, iterate through all restaurants to check if it is satisfied

Attraction information
Execute AttractionSearch to obtain attraction info for city_0
Define 'attraction_in_which_city' variables for 3 (1 per day) attractions
For each 'attraction_in_which_city' variable, assert it to be either the current city or the next city based on transportation arrivals time
Define 'attraction_index' variables for 3 (1 per day) attractions
For each 'attraction_index', retrieve specific length info based on attraction in which city variable, assert indexes are within a valid range, and attractions in the same city are not repeated

Accommodation information
Execute AccommodationSearch to obtain accommodation info and accommodation constraints for city_0
Define 'accommodation_index' variables for 1 (1 per city) accommodation
For each 'accommodation_index', retrieve specific price info based on accommodation in which city variable, assert 'accommodation_index' variable is within a valid range, calculate the number of rooms needed for 2 people and accommodation price
For each city, obtain accommodation minimum night info and assert it to be less than the days stayed in this city
For each 'accommodation_index', retrieve specific room type info, assert 'Private room' exists for all accommodations

Budget
Set budget limit variable to be 1100

Add 2 transportation prices to spent, according to whether the transportation method is flight, self-driving, or taxi
Add restaurant price to spent
Add accommodation price to spent
Assert current spent is within budget

——EXAMPLES END——

Using the examples provided, outline the steps for the following Natural Language query. Adhere to the original step structures. Begin each line containing steps with '#'.
Natural Language query:

Here's an example paraphrased prompt for Step-Code (Destination City constraint type):

```
# Python script to evaluate the satisfiability of constraints related to destination cities in a travel plan problem.

# Execute CitySearch to retrieve all potential destination cities in Minnesota for the dates ["2022-03-16", "2022-03-17",
↪  "2022-03-18"], starting from 'Atlanta'. Exclude 'Atlanta' from the list if present.
cities = CitySearch.run('Minnesota', 'Atlanta', ["2022-03-16", "2022-03-17", "2022-03-18"])
if 'Atlanta' in cities:
    cities.remove('Atlanta')
# Assign cities to a list containing only 'Milwaukee'
cities = ['Milwaukee']
# Iterate over cities to select 2 destination cities
for city_0_index, city_0 in enumerate(cities):
    for city_1_index, city_1 in enumerate(cities):
        # Create a Z3 solver instance named s
        s = Optimize()
        # Define 'city' variable as indices of 2 destination cities
        variables['city'] = [Int('city_' + str(i)) for i in range(2)]
        # If city_0_index and city_1_index differ, assert that the 2 'city' variables match the city indices
        if city_0_index != city_1_index:
            s.assert_and_track(variables['city'][0] == city_0_index, 'visit city in cities list')
            s.assert_and_track(variables['city'][1] == city_1_index, 'visit city in cities list')
# Iterate over cities to select 3 destination cities
for city_0_index, city_0 in enumerate(cities):
    for city_1_index, city_1 in enumerate(cities):
        for city_2_index, city_2 in enumerate(cities):
            # Create a Z3 solver instance named s
            s = Optimize()
            # Define 'city' variable as indices of 3 destination cities
            variables['city'] = [Int('city_' + str(i)) for i in range(3)]
           # If city_0_index, city_1_index, and city_2_index are distinct, assert that the 3 'city' variables match the city indices
            if city_0_index != city_1_index and city_1_index != city_2_index and city_0_index != city_2_index:
                s.assert_and_track(variables['city'][0] == city_0_index, 'visit city in cities list')
                s.assert_and_track(variables['city'][1] == city_1_index, 'visit city in cities list')
                s.assert_and_track(variables['city'][2] == city_2_index, 'visit city in cities list')
# Iterate over cities to select 1 destination city
for city_0_index, city_0 in enumerate(cities):
    # Create a Z3 solver instance named s
    s = Optimize()
    # Define 'city' variable as the index of 1 destination city
    variables['city'] = [Int('city_' + str(i)) for i in range(1)]
    # Assert that the 'city' variable matches the city index
    s.assert_and_track(variables['city'][0] == city_0_index, 'visit city in cities list')

# From the examples provided, lines beginning with '#' are instructions, and the lines following them until the next '#' are the
↪  corresponding code.
# Adhere to the code structure and variable names used in the examples.
# For the instruction below, write the corresponding code and respond with the code only. Begin with ########## Destination cities
↪  response########## and conclude with ########## Destination cities response ends##########.
```

H Example input query and output step for generalization evaluation

We include the full example of Query-Step generation for unseen constraint types here:

Input JSON query {
"org": "Hong Kong",
"dest": ["New York City", "Bangkok"],
"days": 5,
"visiting_city_number": 2,
"date": ["2023-12-22", "2023-12-23", "2023-12-24", "2023-12-25", "2023-12-26"],
"people_number": 3,
"local_constraint": {
"flight rule": "non-stop",
"airlines": ["United", "Emirates"],
"attraction_category": ["Garden", "Historical Landmarks"],
"transportation": "flight"
},
"budget": 5000
}
Corresponding output step # Destination cities #
Set cities to be a list includes 'New York City' and 'Bangkok'
Loop through cities for 2 destination cities
Initialize Z3 solver s
Set 'city' variable to be indexes of 2 destination cities
If city_0_index and city_1_index are not same, assert 2 'city' variables equal to city index

Departure dates
Set 'departure_dates' variables for 3 transportations between cities
Assert first transportation happens at first day (day 0), last transportation happens at last day (day 4), and second transportation could happen at any day in between

Transportation methods
Set transportation method (flight) variable for 3 transportations between cities
Assert all 3 transportations between cities are flight

Flight information
Run FlightSearch to get flight info for Hong Kong as origin, city_0 and city_1, and dates
Get specific flight price info with Hong Kong as origin and final destination, specific city variable, and departure date for 3 transportations
Set 'flight_index' variable for 3 transportations
Assert 3 'flight_index' variables are within valid range if taking flight, assert flight index to be -1 if not taking flight
Calculate flight price for 3 people for 3 transportations based on flight index variable
Get specific flight arrival time info with Hong Kong as origin and final destination, specific city, and departure date for 3 transportations
Calculate flight arrival time for 3 transportations based on flight index variable
For each 'flight_index', get specific flight rule and airlines info, assert 'non-stop' exist for all flights, assert 'United' or 'Emirates' exist for all flights

Attraction information

```
# Get arrivals and city list for each day based on 3 transportations, 5 total travel day, and departure dates variables
# Run AttractionSearch to get attraction info for city_0 and city_1
# Set 'attraction_in_which_city' variables for 5 (1 per day) attractions
# For each 'attraction_in_which_city' variable, assert it to be either current city or next city based on transportation arrivals time
# Set 'attraction_index' variables for 5 (1 per day) attractions
# For each 'attraction_index', get specific length info based on attraction in which city variable, assert index are within valid range, and attrations in same city are not repeated
# Set 'attraction_category' variables for each category required
# For each category, iterate through all attraction to check if it is satisfied

# Restaurant information #
# Set restaurant price to be 30 per person per meal, calculate total restaurant price for 3 people for 5 days

# Accommodation information #
# Set accommodation price to be 100 per person per night, calculate total accommodation price for 3 people for 4 nights

# Budget #
# Set budget limit variable to be 5000
# Add 3 transportation price to spent, according to whether transportation method is flight
# Add restaurant price to spent
# Add accommodation price to spent
# Assert current spent is within budget
```