

Large Language Models Can Plan Your Travels Rigorously with Formal Verification Tools

Anonymous ACL submission

Abstract

The recent advancements of Large Language Models (LLMs), with their abundant world knowledge and capabilities of tool-using and reasoning, fostered many LLM planning algorithms. However, LLMs have not shown to be able to accurately solve complex combinatorial optimization problems. In [Xie et al. \(2024\)](#), the authors proposed TravelPlanner, a U.S. domestic travel planning benchmark, and showed that LLMs themselves cannot make travel plans that satisfy user requirements with a best success rate of 0.6%. In this work, we propose a framework that enables LLMs to formally formulate and solve the travel planning problem as a satisfiability modulo theory (SMT) problem and use SMT solvers interactively and automatically solve the combinatorial search problem. The SMT solvers guarantee the satisfiability of input constraints and the LLMs can enable a language-based interaction with our framework. When the input constraints cannot be satisfiable, our LLM-based framework will interactively offer suggestions to users to modify their travel requirements via automatic reasoning using the SMT solvers. We evaluate our framework with TravelPlanner and achieve a success rate of 97%. We also create a separate dataset that contain international travel benchmarks and use both dataset to evaluate the effectiveness of our interactive planning framework when the initial user queries cannot be satisfied. Our framework could generate valid plans with an average success rate of 78.6% for our dataset and 85.0% for TravelPlanner according to diverse humans preferences.

1 Introduction

Recent work has demonstrated that large language models (LLMs) ([Brown et al., 2020](#); [Ouyang et al., 2022](#); [Achiam et al., 2023](#)), with its abundant world knowledge, abilities to collect information via tools, and capabilities of reasoning, have significant potentials in solving planning problems ([Huang et al.,](#)

[2022a](#); [Ahn et al., 2022](#); [Yao et al., 2022](#); [Song et al., 2023](#)). However, the planning scenarios of the existing LLM planning works are still limited to rather simple tasks such as household cleaning in which the agents only consider one or few constraints. Modern LLMs are not well-suited for directly solving highly complex combinatorial optimization problems with multiple levels of constraints as they 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. Imagine you are preparing for your summer vacation travel. You have a one-week vacation, \$3000 budget, and are longing for somewhere with beautiful beach and delicious seafood restaurants. To make a clear and detailed plan, you need to utilize various tools to search for flights, cities with famous sea attractions, seafood restaurants, accommodations, and may need to repeat this process iteratively to ensure the total price is within your budget. Even for humans, it is a complex and time-consuming undertaking to accomplish. Are LLM agents capable of handling complex and realistic planning problems like this? To investigate this problem, [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 even strongest LLM GPT-4 can only achieve a success rate of 0.6% by itself without access to pre-collected information.

The travel planning problem contains diverse constraints including time, budget, destinations, transportation methods, etc., making it extremely challenging for LLM agents to search for a feasible plan considering all constraints. To tackle this problem, 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](#)), boolean satisfiability problem (SAT) ([Kautz and Selman,](#)

1999; Rintanen, 2012), or 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, as human natural language queries have no fixed format, constraint-based planners need to extract key information from input queries accurately to model the problem. Those algorithm-based planning tools usually have a steep learning curve. Crucially, even if the extracted key information is correct, the proposed query can be unsatisfiable itself so that the users will need to query the tools multiple times with modified inputs.

LLMs are good at parsing human input and interactions but hard to rigorously take all constraints into consideration, while SMT solvers are sound and complete in solving multi-constraints satisfiability problems but unable to handle dynamic, general, and sometimes ambiguous natural language requirements. In this work, we propose a framework that combines the advantages of both methods by enabling LLM to utilize SMT solver as a tool to formally formulate, solve, and reason over the travel planning problem. In our framework, the LLM first translates natural language input to the fixed JSON format. Then, with instruction steps and corresponding codes of using SMT solver to solve the example travel planning problem, LLM learns the pattern and generalizes to new inputs. Executing LLM generated codes encodes the query requirements and calls the solver, which guarantees to generate a plan if it exists. Furthermore, if the input query is not satisfiable, SMT solvers can identify the exact constraints that make the problem unsatisfiable, using which LLM could propose suggestions to modify the query by analyzing the unsatisfiable reasons and interactively communicating with humans to modify the query until it becomes satisfiable.

We evaluate our framework over different LLMs using TravelPlanner Xie et al. (2024), which contains 180 satisfiable queries in their validation set and 1000 satisfiable queries in their test set. Our framework achieves a best final pass rate of 98.9% in the validation set and 97.0% in the test set. To further evaluate the plan repair capability for unsatisfiable queries, we modify 12 constraints from TravelPlanner’s training set to be unsatisfiable, and also build another international travel dataset that contains 39 unsatisfiable queries with the types of

constraints different from TravelPlanner. We evaluate on both datasets and show our framework’s capability of interactively make satisfiable travel plans for users with different preferences of modifying their constraints. We also verify with ablation studies the positive effects of key components of our framework including the capabilities to call SMT solver iteratively for verification, to gather unsatisfiable reasons from SMT solver, and to incorporate human feedback to better align with human preferences. Finally, we test the generalization capability of our framework by encoding unseen constraints with existing instruction steps examples. All the experiments demonstrate that our framework could reliably handle diverse human inputs, deliver formally-verified plans, interactively modify unsatisfiable queries based on diverse human preferences, and could be effectively adapted to different LLMs.

2 Related Work

LLM Planning. LLMs have shown significant intelligence in various tasks such as 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 problem with various ways: 1) decomposing the task into sub-tasks and plan sub-goals sequentially (Wei et al., 2022; Yao et al., 2022; Shen et al., 2024); 2) generating multiple plans with methods like tree and graph search and selecting the optimal solution from them (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 from environment (Shinn et al., 2024; Madaan et al., 2024; Chen et al., 2023b); 4) formalizing tasks and aiding the planning with external planner (Liu et al., 2023; Guan et al., 2023; Chen et al., 2023a). These methods are summarized in Huang et al. (2024) with details. While these planning algorithms have shown promising result, their planning scenarios are limited to simple tasks with single objective function. Xie et al. (2024) proposes a realistic and complex travel planning benchmark and tests on various LLM planning algorithms to show that LLMs are not capable of handling multi-constraint tasks. While Liu et al. (2023); Guan et al. (2023) utilize LLM to process information and formulate problems into PDDL (Aeronautiques et al., 1998;

Haslum et al., 2019) to account for multiple objectives, they have to solve PDDL problem with heuristic search can not verify the generated plan.

Algorithm-based Planning. Another way to tackle the travel planning problem is through algorithm-based planning such as heuristic search (Helmert, 2006; Hoffmann and Nebel, 2001; Vidal, 2014) and constraint-based methods (Kautz and Selman, 1999; Rintanen, 2012, 2014; Lozano-Pérez and Kaelbling, 2014; Dantam et al., 2016). However, heuristic search is not able to guarantee to find the plan, and pure constraint-based planning is not able to generalize to diverse human natural language inputs. Our framework enables LLM to utilize the constraint-based planning method by translating and formalizing diverse human queries into a SMT problem. Since SMT solvers are sound and complete, the generated plan is guaranteed to be correct. If SMT solver is not able to find a solution, the problem is verified to be unsatisfiable and the solver can output unsatisfiable reasons for future usage.

LLM Tool-use. Tool-using is crucial for LLMs in that it allows LLMs to utilize powerful external tools to increase reliability. Recent works explore how LLMs could make use of 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 a SMT problem, and calls the SMT solver to solve it. This overcomes LLM’s failure to consider all constraints by encoding all constraints rigorously and solve it with solver.

3 Approach

We propose a framework that equips the LLM with tools to formulate and solve the travel planning problem as an SMT problem, shown in Fig. 1. In our framework, we call LLM multiple times to accomplish a number of distinct functionalities: translating natural language query into a JSON format description, generating steps to formulate the problem, and generating codes based on steps. In addition, when the input query is not satisfiable, LLM reasons about current situation to give suggestion and modifies existing codes based on suggestion.

3.1 Satisfiable Plan Solving

3.1.1 Problem Statement

We define the travel planning problem as: given a natural language description of humans’ personal 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 method for $k + 1$ travels from city to city. The travel visit n attractions, dine in $3n$ restaurants, and live in accommodations for $n - 1$ nights. Table 1 summarizes the constraints \mathcal{C} for the two datasets we used. The output plan should satisfies \mathcal{C} and clearly specifies the city to visit, the transportation method, the attraction, restaurant, and accommodation for each day. See Appendix A for example input query and output plan.

3.1.2 NL-JSON Translation

Travel planning problem is a real-world complex long-horizon planning problem that contains various constraints including time, location, budget, etc. These information are contained in humans’ natural language instructions in different forms. The first step of our framework is to extract important information from the natural language input and translate into a problem description of JSON format, as shown in Fig. 1 part 1). LLM is provided with a description of the required fields in JSON and three translation examples. Refer to C.1.1 for the full NL-JSON Translation prompt.

3.1.3 JSON-Step Generation

The steps to formulate travel plan problems differ with the change of constraints, e.g., number of destination cities, number of travel days, and special restaurant cuisine requirement. However, although the steps may be different, they have similar patterns. We provide LLM with the JSON problem description and three examples of JSON-Step generation in the prompt to enable it to generalize to different input queries. We separate the steps into subsections based on the type of constraints. For example, to specify the "travel spanning 3 days" constraint in the query in Fig. 1, the steps are:

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)

While for a new query that asks to travel 2 destination cities in 5 days, the steps become:

1. Set 't_dates' variables for 3 transportation between cities
2. Assert first transportation happens at first day (day 0),

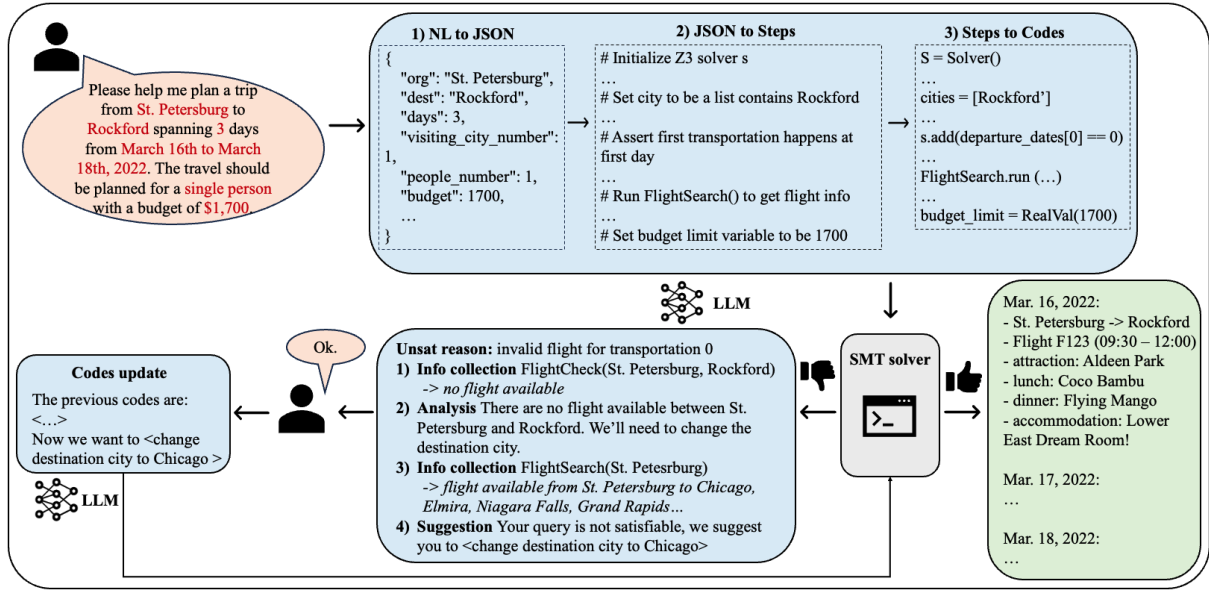


Figure 1: An overview of the framework. The pink region represents human, blue region represents LLM, gray region represents SMT solver, and the green region is the generated plan. Given a natural language query, LLM 1) translates it into JSON format, 2) generates steps to formulate it as a SMT problem, 3) generates corresponding codes that encode the problem and call the solver. If the solver is not able to find the solution, LLM collects information, analyzes current situation, and offer suggestions to modify query interactively. LLM then update the code based on suggestions and call the solver again to find a feasible plan.

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 1: Descriptions of constraints for two datasets. Constraints in green are the constraints only in TravelPlanner. Constraints in blue are the constraints only in our dataset. Constraints in black are common constraints.

last happens at last day (day 4),
and second could happen at any day in between

See Appendix C.1.2 for JSON-Step prompt.

3.1.4 Step-Code Generation

Inspired by Liang et al. (2023), we directly prompt the LLM to generate language model programs in

Python by providing CitySearch, FlightSearch, AttractionSearch, DistanceSearch, AccommodationSearch, RestaurantSearch APIs, and SMT solver, and demonstrating how to use each of these functions. With the generated steps of how to solve each constraint, we provide these steps as instructions to LLM, as shown in Fig. 1 part 3). Fig. 2 shows

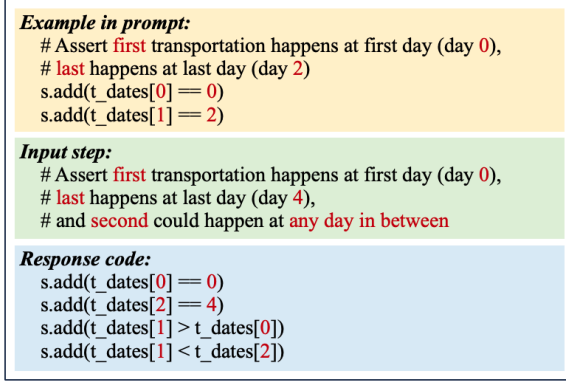


Figure 2: Step to Code translation example.

how LLM generalizes to new instruction steps to write corresponding codes given examples. See Appendix C.1.3 for Step-Code generation prompt.

3.1.5 SMT Solver

After gathering the generated codes, our framework execute the 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. In addition, when 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 (De Moura and Bjørner, 2008). When the framework proves the constraints to be unsatisfiable, it proceeds to interactive plan repair with the reasons.

3.2 Interactive Plan Repair

As travel planning problem is a real-world complex problem that involves diverse constraints to consider, when a proposed query is not satisfiable, LLM’s reasoning capability and commonsense knowledge to analyze current situation and offer suggestions become vital. Furthermore, these capabilities enable an interactive setting, in which humans can agree, disagree, or provide comments and feedback to LLM’s proposed suggestions. LLM can then output 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 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 takes action to collect flight information by calling FlightCheck API. Realizing no flight is available between St. Petersburg and Rockford, LLM analyzes and decides to change the destination city. Then, it runs FlightSearch API to search for all possible destinations going from St. Petersburg and eventually chooses one of them to be the suggested destination. LLM offers this as a suggestion to the user, and waits for the user’s feedback. The feedback could be agree, disagree, any natural language description of preference, or even modifications proposed by users themselves. If the user disagrees with the suggestion or provides their preferences, the framework starts another iteration and proposes new suggestions. If the users 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 gathers the unsatisfiable reasons and starts 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 (Xie et al., 2024) 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 2019¹ and obtain attraction information from Metabase². We utilize Google Flights³ to collect flight information during 12-24-2023 to 12-30-2023 for these ten cities. Compared with TravelPlanner, we omit the detailed information of transportation methods, restaurants, and accommodations but introduces detailed constraints regarding flights and attractions. As shown in Table 1, UnsatChristmas allows users to specify

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>

1) whether they want to take non-stop flight only, 2) the list of airlines they prefer, 3) the list of attraction categories they prefer. 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.

5 Experimental Results

We examine our framework on both TravelPlanner and UnsatChristmas. We use GPT-4 (Achiam et al., 2023) with temperature 0 as our LLM by default, and we also compare with Claude 3 Opus-20240229 (cla) and Mixtral-Large (mix) with temperature 0 for satisfiable plan solving evaluation. We use Z3 solver (De Moura and Bjørner, 2008) as our SMT solver for all experiments.

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 training set. We evaluate our method on both the validation set (180 satisfiable queries) and test set (1000 satisfiable queries) of TravelPlanner. Although our framework evaluates validation and testing set identically, we report results on validation and test sets separately for better comparison.

Evaluation Metric Following Xie et al. (2024), the evaluation metrics include: 1) Delivery Rate: measures whether a final plan is generated within limited time; 2) Commonsense Constraint Pass Rate: evaluates whether LLMs pass the commonsense constraints; 3) Hard Constraint Pass Rate: assesses whether LLMs pass the hard constraints; 4) Final Pass Rate: represents whether LLMs pass all constraints. Please refer to Table 1 for all constraints of TravelPlanner. Please refer to Appendix B for detailed category of commonsense constraints and hard constraints and detailed description of Micro and Macro evaluation modes.

Baselines Xie et al. (2024) evaluated performance of traditional rule-based strategies and various LLM planning methods on TravelPlanner. We compare our framework with three strongest models using different strategies. Greedy Search uses traditional search algorithm and introduces heuristics to set total cost as the optimization objective. TwoStage (GPT-4), the most powerful model among two-stage tool-use frameworks, collects information with ReAct (Yao et al., 2022) and gives plan based on collected information. Direct (GPT-4), the most powerful model among sole-planning frameworks, already has access to all necessary pre-collected information and gives plan without tool-calling needs. To verify the effectiveness of our framework in varied LLMs, we also evaluate our framework with Claude 3 Opus and Mixtral-Large. We tune the prompt with training set and evaluate on validation set. We include the changes we make to the change compared to Ours (GPT-4) in Appendix C.1.4. Due to computational resources and cost considerations, we only evaluate two new LLMs on the validation set with 180 queries.

Results and Analysis Table 2 shows the comparison result of the performance over satisfiable queries. Since some queries are challenging in that they have few feasible plans, we limit SMT solver’s maximum runtime for each query to be 30 minutes and pause the program if this limit is reached. The delivery rate of Ours (GPT-4) is 99.4% for validation set and 97.2% for test set. From the results, we can observe that both LLM planning methods, TwoStage (GPT-4) and Direct (GPT-4), struggle to take all constraints into consideration with a final pass rate of 0.6% and 4.4%. In addition, results of Greedy Search show that without formal specification, the heuristic-based method fails to pass any of the tasks. Ours (GPT-4), with the capability of formally encoding the problem as an SMT problem, achieves a high final pass rate of 98.9% for validation set and 97.0% for test set. This demonstrates our framework’s robustness in solving satisfiable queries of the travel planning problem. In addition, Ours (Mixtral-Large) achieves a delivery rate of 85.0% and final pass rate of 84.4% and Ours (Claude-3) achieves a delivery rate and final pass rate of 98.3%. Ours (Claude-3) could reach comparable results as Ours (GPT-4). Although the delivery rate for Ours (Mixtral-Large) drops 14.4% comparing to Ours (GPT-4), it still significantly outperforms TwoStage (GPT-4) and Direct (GPT-4), and 99.2% of its delivered plans are correct

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
Ours (Mixtral-Large)	85.0	85.0	85.0	79.3	84.4	84.4
Ours (Claude-3)	98.3	98.3	98.3	98.6	98.3	98.3
Ours (GPT-4)	99.4	99.4	99.4	99.5	98.9	98.9
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 (GPT-4)	97.2	97.2	97.2	96.2	97.0	97.0

Table 2: Performance comparison of satisfiable queries for 180 queries in validation set and 1000 queries in test set. The results of Greedy Search, TwoStage, and Direct are from Xie et al. (2024)

plans. We discussed the major failure cases of Ours (Mixtral-Large) in Appendix C.1.5. These results demonstrates the adaptability of our framework to other LLMs.

5.2 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 our framework based on the success rate: whether LLM eventually modifies the constraints to successfully deliver a feasible plan within a limited number of iterations.

Implementation Details The unsatisfiable constraints from UnsatChristmas have four unsatisfactory modes: 1) budget is not enough, 2) the required non-stop flight does not exist, 3) the required airline does not exist, 4) the required attraction category does not exist. We test our framework with mimic users. In addition to a user who will agree to all suggestions proposed by LLM, we have users who have a hard constraint for budget, non-stop flight, airline, attraction category, destination cities, respectively. They will refuse any suggestion that changes their hard constraint, and provide feedback indicating they will not change this information.

The unsatisfiable constraints we modified from TravelPlanner have three unsatisfactory modes: 1) budget is not enough, 2) the required transportation method does not exist, 3) the required house type does not exist. The mimic users have hard constraints for budget, destination cities, transportation methods, and house type. We limit LLM to take at most ten iterations to offer ten suggestions. We

include the prompt in Appendix C.2.

Ablation Studies Several key components in our framework is 1) LLM receives unsatisfiable reasons from the solver; 2) LLM collects information based on the reasons, and then analyzes and offers suggestions; 3) LLM receives human preferences regarding offered suggestions; 4) LLM modifies codes to encode problem; 5) SMT solver gives satisfiable or unsatisfiable verification. We perform ablation studies to examine these key components. We compare the following:

- **No Reason.** This baseline evaluates the need of unsatisfiable reasons by asking LLM to resolve unsatisfiable queries without providing unsatisfiable reasons.
- **No Feedback.** This baseline tests how well the framework incorporate human feedback and preference by not providing human feedback to LLM’s suggestions. Instead, it only provide binary “agree” or “disagree”.
- **No Solver.** This baseline examines the importance of tool-using by removing the SMT solver. The LLM collects information and give a list suggestions for only one iteration because it is not able to call the solver to verify the updated query.
- **Ours.** Our approach as described in 3.2
- **Ours-20.** A variant of our approach that changes the maximum number of iterations to be 20.

Results and Analysis Table 3 and Table 4 show the results of interactive plan repair performance. Our framework could address diverse human preferences with an average of 78.6% and 85.0%

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

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-Code	91.7	75	100	83.3	75	85.0
Ours-20	100	83.3	100	91.7	83.3	91.7

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

across all types of mimic humans. **Ours-20**, a variant of our approach that increases the maximum suggestion iterations to be 20 with a tradeoff of generation time, could raise the success rate to 81.6% and 91.7%. This shows the potentials of increasing the iteration limit to achieve better results. For both queries from TravelPlanner and UnsatChristmas, **Ours** significantly outperforms **No Solver** by an average of 53.4% and 60.0% across all types of mimic humans. This suggests that LLM’s capability to utilize 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% across all types of mimic humans. These results validate the effectiveness of our key components.

5.3 Generalization to unseen constraints

Since 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. As shown in Table 1, we design UnsatChristmas to have different constraints as TravelPlanner. We show that by adding several lines of task description in the JSON-Step Generation prompt, LLM could generate instruction steps for new query in UnsatChristmas without the need to add new examples. Figure 3 shows how

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 generalize to unseen constraints. Texts with yellow background are the unseen constraint types, texts with green background are corresponding generated steps.

our framework encodes unseen constraints in UnsatChristmas. Please see Appendix C.3 for the added task description and see Appendix D for full steps generated from a JSON input.

6 Conclusion

In this work, we propose a framework that enables LLM to utilize SMT solver to formally formulate and solve a complex travel planning problem as a SMT problem. Our framework can generalize to natural language query inputs, almost guaranteeing to deliver plan if the query is satisfiable with a pass rate of 97%, and can interactively work with humans to modify the input query if it is not satisfiable. Finally, we show that our framework can generalize to unseen constraint type without the need to add new examples to the prompt.

7 Limitation

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

Prompt Designing We need careful and specific design of instruction steps and corresponding codes to encode the problem. It is time-consuming to formulate the problem from scratch. However, the potential of our framework to generalize to the unseen constraints ease the future efforts needed to incorporate more constraints into the framework.

Solver Runtime For massive database with more destination city choices, various constraint types, and queries that only have few feasible plans, our framework could take 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 source 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.
- 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*.
- Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins Sri, Anthony Barrett, Dave Christianson, et al. 1998. Pddl the planning domain definition language. *Technical Report, Tech. Rep.*
- 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, 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.
- Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. 2023. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, 36:79081–79094.
- 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*.

697	Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni,	et al. 2024. Self-refine: Iterative refinement with	752
698	Christian Muise, Ronald Brachman, Francesca Rossi,	self-feedback. <i>Advances in Neural Information Pro-</i>	753
699	and Peter Stone. 2019. <i>An introduction to the</i>	cessing Systems, 36.	754
700	<i>planning domain definition language</i> , volume 13.		
701	Springer.		
702	Malte Helmert. 2006. The fast downward planning	Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida,	755
703	system. <i>Journal of Artificial Intelligence Research</i> ,	Carroll Wainwright, Pamela Mishkin, Chong Zhang,	756
704	26:191–246.	Sandhini Agarwal, Katarina Slama, Alex Ray, et al.	757
705		2022. Training language models to follow instruc-	758
706	Jörg Hoffmann and Bernhard Nebel. 2001. The ff plan-	tions with human feedback. <i>Advances in neural in-</i>	759
707	ning system: Fast plan generation through heuristic	formation processing systems, 35:27730–27744.	760
708	search. <i>Journal of Artificial Intelligence Research</i> ,		
709	14:253–302.	Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng,	761
710		Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou	762
711	Wenlong Huang, Pieter Abbeel, Deepak Pathak, and	Yu, Weizhu Chen, et al. 2023. Check your facts and	763
712	Igor Mordatch. 2022a. Language models as zero-	try again: Improving large language models with	764
713	shot planners: Extracting actionable knowledge for	external knowledge and automated feedback. <i>arXiv</i>	765
714	embodied agents. In <i>International Conference on</i>	<i>preprint arXiv:2302.12813</i> .	766
715	<i>Machine Learning</i> , pages 9118–9147. PMLR.		
716		Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt,	767
717	Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan,	Noah A Smith, and Mike Lewis. 2022. Measuring	768
718	Jacky Liang, Pete Florence, Andy Zeng, Jonathan	and narrowing the compositionality gap in language	769
719	Tompson, Igor Mordatch, Yevgen Chebotar, et al.	models. <i>arXiv preprint arXiv:2210.03350</i> .	770
720	2022b. Inner monologue: Embodied reasoning		
721	through planning with language models. <i>arXiv</i>	Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan	771
722	<i>preprint arXiv:2207.05608</i> .	Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang,	772
723		Bill Qian, et al. 2023. Toolllm: Facilitating large	773
724	Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei	language models to master 16000+ real-world apis.	774
725	Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruim-	<i>arXiv preprint arXiv:2307.16789</i> .	775
726	ing Tang, and Enhong Chen. 2024. Understanding		
727	the planning of llm agents: A survey. <i>arXiv preprint</i>	Jussi Rintanen. 2012. Planning as satisfiability: Heuris-	776
728	<i>arXiv:2402.02716</i> .	tics. <i>Artificial intelligence</i> , 193:45–86.	777
729			
730	Henry Kautz and Bart Selman. 1999. Unifying sat-	Jussi Rintanen. 2014. Madagascar: Scalable planning	778
731	based and graph-based planning. In <i>IJCAI</i> , vol-	with sat. <i>Proceedings of the 8th International Plan-</i>	779
732	ume 99, pages 318–325.	ning Competition (IPC-2014), 21:1–5.	780
733			
734	Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yu-	Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta	781
735	taka Matsuo, and Yusuke Iwasawa. 2022. Large lan-	Raileanu, Maria Lomeli, Eric Hambro, Luke Zettle-	782
736	guage models are zero-shot reasoners. <i>Advances in</i>	moyer, Nicola Cancedda, and Thomas Scialom. 2024.	783
737	<i>neural information processing systems</i> , 35:22199–	Toolformer: Language models can teach themselves	784
738	22213.	to use tools. <i>Advances in Neural Information Pro-</i>	785
739		cessing Systems, 36.	786
740	Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol		
741	Hausman, Brian Ichter, Pete Florence, and Andy	Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li,	787
742	Zeng. 2023. Code as policies: Language model	Weiming Lu, and Yueting Zhuang. 2024. Hugging-	788
743	programs for embodied control. In <i>2023 IEEE In-</i>	gpt: Solving ai tasks with chatgpt and its friends	789
744	<i>ternational Conference on Robotics and Automation</i>	in hugging face. <i>Advances in Neural Information</i>	790
745	<i>(ICRA)</i> , pages 9493–9500. IEEE.	<i>Processing Systems</i> , 36.	791
746			
747	Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu,	Noah Shinn, Federico Cassano, Ashwin Gopinath,	792
748	Shiqi Zhang, Joydeep Biswas, and Peter Stone.	Karthik Narasimhan, and Shunyu Yao. 2024. Re-	793
749	2023. Llm+ p: Empowering large language mod-	flexion: Language agents with verbal reinforcement	794
750	els with optimal planning proficiency. <i>arXiv preprint</i>	learning. <i>Advances in Neural Information Process-</i>	795
751	<i>arXiv:2304.11477</i> .	<i>ing Systems</i> , 36.	796
752			
753	Tomás Lozano-Pérez and Leslie Pack Kaelbling. 2014.	Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit	797
754	A constraint-based method for solving sequential ma-	Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox,	798
755	nipulation planning problems. In <i>2014 IEEE/RSJ</i>	Jesse Thomason, and Animesh Garg. 2023. Prog-	799
756	<i>International Conference on Intelligent Robots and</i>	prompt: Generating situated robot task plans using	800
757	<i>Systems</i> , pages 3684–3691. IEEE.	large language models. In <i>2023 IEEE International</i>	801
758		<i>Conference on Robotics and Automation (ICRA)</i> ,	802
759	Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler	pages 11523–11530. IEEE.	803
760	Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon,		
761	Nouha Dziri, Shrimai Prabhumoye, Yiming Yang,	Chan Hee Song, Jiaman Wu, Clayton Washington,	804
762		Brian M Sadler, Wei-Lun Chao, and Yu Su. 2023.	805

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.

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.

A 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": "-"
}

```

B Satisfiable Plan Solving Evaluation Details

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 number 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.

C Prompts

C.1 Prompts for Satisfiable Plan Solving

C.1.1 NL-JSON prompt

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

Please assist me in extracting valid information from a given natural language text and reconstructing it in JSON format, as demonstrated in the following example.

In the JSON, "org" denotes the departure city. "dest" denotes the destination city. "days" denotes the total number of travel days. When "days" exceeds 3, "visiting_city_number" specifies the number of cities to be covered in the destination state. "date" includes the detailed date to visit. 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 neither are mentioned in the text, make the value to be null.

Here are three examples:

—EXAMPLE 1—

Text: Please help me plan a trip from St. Petersburg to Rockford spanning 3 days from March 16th to March 18th, 2022. The travel should be planned for a single person with a budget of \$1,700.

JSON:

```
{
  "org": "St. Petersburg",
  "dest": "Rockford",
  "days": 3,
  "visiting_city_number": 1,
  "date": ["2022-03-16", "2022-03-17", "2022-03-18"],
  "people_number": 1,
  "local_constraint": {
    "house rule": null,
    "cuisine": null,
    "room type": null,
    "transportation": null
  },
  "budget": 1700
}
```

—EXAMPLE 2—

Text: {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.}

JSON:

```
{
  "org": "Fort Lauderdale",
  "dest": "Milwaukee",
  "days": 3,
  "visiting_city_number": 1,
  "date": ["2022-03-08", "2022-03-09", "2022-03-10"],
  "people_number": 2,
  "local_constraint": {
    "house rule": null,
    "cuisine": ["American", "Chinese"],
    "room type": null,
    "transportation": null
  },
  "budget": 1100
}
```

—EXAMPLE 3—

Text: {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.}

JSON:

```
{
  "org": "Atlanta",
```

```

"dest": "Minnesota",
"days": 5,
"visiting_city_number": 2,
"date": ["2022-03-03", "2022-03-04", "2022-03-05", "2022-03-06", "2022-03-07"],
"people_number": 3,
"local_constraint": {
  "house rule": "parties",
  "cuisine": null,
  "room type": "entire room",
  "transportation": "no self-driving"
},
"budget": 7900
}
——EXAMPLES END——
Text: {query}
JSON:

```

873

C.1.2 JSON-Step prompt

874

The instruction prompt for JSON description to steps translation prompt is provided as follows:

875

You are given a constraint to satisfy for a travel plan problem in JSON format.

In the JSON, "org" denotes the departure city. When total travel days is 5 or 7, "dest" denotes the destination state; when total travel day is 3, "dest" denotes the destination city. "days" denotes the total number of travel days. When "days" equals 5 or 7, "visiting_city_number" specifies the number of cities to be covered in the destination state. "date" includes the specific date to visit. 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.

Your job is to give a detailed step by step instruction to encode this constraint as code.

Here are some example steps for different constraint:

——EXAMPLE 1——

JSON Constraint:

```

{
  "org": "Atlanta",
  "dest": "Minnesota",
  "days": 5,
  "visiting_city_number": 2,
  "date": ["2022-03-03", "2022-03-04", "2022-03-05", "2022-03-06", "2022-03-07"],
  "people_number": 3,
  "local_constraint": {
    "house rule": "parties",
    "cuisine": null,
    "room type": "entire room",
    "transportation": "no self-driving"
  },
  "budget": 7900
}

```

876

Steps:

```
# Destination cities #
# Run CitySearch to get all possible destination cities in Minnesota State 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
# 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 attractions 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—

JSON Constraint:

```
{
  'org': 'Indianapolis',
  'dest': 'Colorado',
  'days': 7,
  'visiting_city_number': 3,
  'date': ['2022-03-11', '2022-03-12', '2022-03-13', '2022-03-14', '2022-03-15', '2022-03-16',
    '2022-03-17'],
  'people_number': 5,
  'local_constraint': {
    'house rule': 'pets',
    'cuisine': ['Mexican', 'Italian', 'Mediterranean', 'Indian'],
    'room type': 'entire room',
    'transportation': None
  },
  'budget': 15100
}
```

```

}
Steps:
# Destination cities #
# Run CitySearch to get all possible destination cities in Colorado State 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
# 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 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 attractions 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 'Entire
home/apt' exist for all accommodations, assert 'No pets' 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——
JSON Constraint:
{
  "org": "Fort Lauderdale",
  "dest": "Milwaukee",
  "days": 3,
  "visiting_city_number": 1,
  "date": ["2022-03-08", "2022-03-09", "2022-03-10"],
  "people_number": 2,
  "local_constraint": {
    "house rule": null,
    "cuisine": ["American", "Chinese"],
    "room type": null,
    "transportation": "no flight"
  },
},

```

```

"budget": 1100
}
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
# 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 required
# For each cuisine type, iterate through all restaurant to check if it is satisfied

# Attraction information #
# Run AttractionSearch to get attraction info for city0
# Set 'attractioninwhichcity' variables for 3 (1 per day) attractions
# For each 'attractioninwhichcity' variable, assert it to be either current city or next city based on
transportation arrivals time
# Set 'attractionindex' variables for 3 (1 per day) attractions
# For each 'attractionindex', get specific length info based on attraction in which city variable,
assert index are within valid range, and attractions in same city are not repeated

# Accommodation information #
# Run AccommodationSearch to get accommodation info and accommodation constraints for city0
# Set 'accommodationindex' variables for 1 (1 per city) accommodations
# For each 'accommodationindex', get specific price info based on accommodation in which city
variable, assert 'accommodationindex' 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

# 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 JSON constraint.
Note to keep the format in examples and start each line containing steps with '#'
JSON Constraint: {JSON}
Steps:

```

C.1.3 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
    ⇨ simultaneously 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
    ⇨ simultaneously 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
    ⇨ simultaneously 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 instruction, 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 instruction, 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#####.

```

893

894

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 1')
        s.assert_and_track(Implies(arrives[date_index] > 15, variable == city_list[date_index]), 'eat in which city 1')
        s.assert_and_track(Implies(arrives[date_index] < 10, variable == city_list[date_index+1]), 'eat in which city 1')
    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 instruction, 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 attractions 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 instruction, 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#####.
```

C.1.4 Prompt difference of GPT-4, Claude 3, and Mixtral-Large

With the prompt we have for GPT-4 as the starting point, we adjust the prompts (add more explanations or examples) for Claude-3 and Mixtral-Large with 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, Mixtral-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 Mixtral-Large sometimes misses some information in JSON translation.
- Claude-3 uses the same JSON-Step prompt as Mixtral-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 Mixtral-Large in general produces more code generation errors compared to GPT-4 and Claude-3, thus needing more examples and explanations provided in prompts.

C.1.5 Failure cases of Mixtral-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 Mixtral-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 Mixtral-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 modes and is producing 7.8% failed delivered plan (15.0% in total). Other than this, the failure are induced by runtime issue or some occasional code generation errors.

C.2 Prompts for Interactive Plan Repair

C.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 analyzed 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 explanations, 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 analyzed 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 explanations, you need to give a suggestion within 15 iterations:

Input query: {query}
Unsatisfiable reasons: {reasons}
Collected information: {info}

C.2.2 Suggestion-No Reason prompt

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

C.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 analyzed 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 explanations, 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"

961

962

963

964

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 analyzed 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 explanations, you need to give a suggestion within 15 iterations:

Input query: {query}

Collected information: {info}

966

C.2.4 Code modify prompt

967

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}

968

C.3 Prompts for Generalization Evaluation

969

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:

970

971

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

972

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.

D Example input JSON and output step for generalization evaluation

We include the full example of JSON-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 attractions 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

```