Generalized Planning in PDDL Domains with Pretrained Large Language Models

Anonymous Author(s)
Affiliation
Address
email

Abstract

Recent work has considered whether large language models (LLMs) can function as planners: given a task, generate a plan. We investigate whether LLMs can serve as generalized planners: given a domain and training tasks, generate a program that efficiently produces plans for other tasks in the domain. In particular, we consider PDDL domains and use GPT-4 to synthesize Python programs. We also consider (1) Chain-of-Thought (CoT) summarization, where the LLM is prompted to summarize the domain and propose a strategy in words before synthesizing the program; and (2) automated debugging, where the program is validated with respect to the training tasks, and in case of errors, the LLM is re-prompted with four types of feedback. We evaluate this approach in seven PDDL domains and compare it to four ablations and four baselines. Overall, we find that GPT-4 is a surprisingly powerful generalized planner. We also conclude that automated debugging is very important, that CoT summarization has non-uniform impact, that GPT-4 is far superior to GPT-3.5, and that just two training tasks are often sufficient for strong generalization.

1 Introduction

While some classes of sequential decision-making tasks are provably intractable [1], others can be solved efficiently with a single domain-specific program. In the latter case, there is considerable interest in *automatically synthesizing* these programs given a small number of training tasks. In AI planning, several approaches to this *generalized planning* problem have been proposed, with programs expressed as lifted decision lists, as finite state machines, or in domain-specific languages [2–5]. In reinforcement learning, goal-conditioned policies and value functions can be understood as particular kinds of programs learned with the same generalized planning objective [6–9]. Despite these efforts, it remains challenging to efficiently synthesize programs from few training tasks that generalize to a wide variety of held-out tasks.

Given the tremendous recent progress in large language models (LLMs) [10–12], especially in code generation, this work asks a simple question: *can pretrained LLMs be used for generalized planning?* In particular, we investigate whether GPT-4 [13] can be used to write a domain-specific Python program that solves a set of tasks in a planning domain. For each domain, we prompt GPT-4 with the domain and a small number of training tasks, all encoded in the Planning Domain Definition Language (PDDL) [14]. We then ask GPT-4 to write a Python program that consumes a (parsed) task description and outputs a plan. To prevent it from writing domain-general search-based code — a natural inclination given the association between PDDL and search in its pretraining data — we instruct GPT-4 to implement "a simple strategy that does not use search."

Beyond this basic protocol, we consider two extensions. First, inspired by Chain-of-Thought (CoT) [15, 16], we prompt GPT-4 to write a natural language *summary* of the PDDL domain. We

Preprint. Under review.

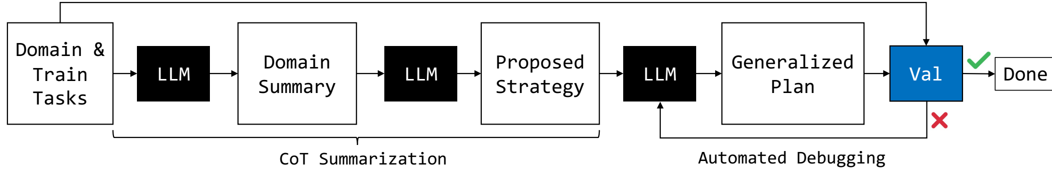

Figure 1: Overview of pipeline for generalized planning with pretrained LLMs. See text for details.

then ask it to describe a solution strategy before finally implementing the strategy in Python. Second, inspired by Inner Monologue [17] and Corrective Re-prompting [18], we automatically provide feedback to GPT-4 in the case where it fails to solve training tasks. For example, if executing the Python code results in an exception, we present GPT-4 with that exception and ask it to fix the code. We repeat this *automated debugging* process up to four times or until all training tasks are solved. See Figure 1 for an overview of this pipeline.

In our experiments, we evaluate this approach on seven PDDL domains: six from recent work in generalized planning [19], and a seventh novel domain. We find that the approach is a strong baseline compared to existing generalized planning approaches. This is an important finding that we expect to inform further research in generalized planning. We also present a suite of ablations and additional analyses to unpack the contributions of CoT summarization, automated debugging, names in the PDDL, and GPT-4 vs. GPT-3.5. Our results suggest that automated debugging, PDDL names, and GPT-4 are very important, while the impact of CoT is non-uniform. Finally, we provide qualitative analyses of common failure cases, suggesting directions for future work. Altogether, we conclude that GPT-4 is a surprisingly powerful generalized planner, when properly guided.

2 Related Work

LLMs for (PDDL) Planning. *Generalized planning* with LLMs can be seen as an alternative to *planning* with LLMs [20–22, 18, 23]. Most relevant is work by Valmeekam et al. [24], Silver et al. [25] who consider LLM-based planning in PDDL domains. There are several advantages to using LLMs for generalized planning, rather than planning: (1) programs produced by the LLM can be inspected and validated; (2) running a synthesized program can be much faster (and cheaper) than querying the LLM for each new task; (3) synthesized programs can scale to arbitrarily large tasks, whereas current LLMs are limited by context window size. Pallagani et al. [26] consider fine-tuning an LLM to solve PDDL tasks. Other recent work has considered using LLMs for translating between natural language and PDDL [27–30]. These efforts could be combined with our approach.

Generalized Planning. This work contributes to a growing literature on generalized planning [31, 4]. Prior work has considered synthesizing generalized plans in several ways: (1) performing a search through a hypothesis class of generalized policies [32–35]; (2) using example plans to construct a generalized plan, often represented with a finite-state machine [36–38]; and (3) discovering state and action abstractions and then using them in a generalized plan [39]. One pervasive challenge is that there are often many valid plans for any given task, and only some of these plans are consistent with a simple generalized plan. PG3 addresses this challenge by using candidate generalized plans (represented as lifted decision list goal-conditioned policies) to constrain the generation of example plans [19]. We use PG3 as the main point of comparison in experiments.

LLMs for Code Generation. Our work builds on recent techniques that use LLMs for code generation [11, 40]. CoT summarization is related to several techniques that ask the LLM to outline its "thinking" before arriving at a final implementation [15, 16, 41]. A number of recent works also use programs as prompts (i.e., a structured chain of thought) in an attempt to help LLMs perform mathematical reasoning [42, 43]. Related to our automated debugging, Xia and Zhang [44], Chen et al. [45] consider automated program repair by re-prompting the LLM with feedback from failed validation checks. Chen et al. [46] consider a related paradigm, but where feedback comes from humans, rather than automated checks. Also relevant are efforts to generate code that can be used for robotic decision-making [47, 48]. Beyond LLMs, code generation has been studied extensively in program synthesis [49, 50] and inductive logic programming [51, 52].

3 Background and Problem Setting

PDDL Domains and Tasks. We consider deterministic, fully-observed planning tasks represented in PDDL. In experiments, we use the STRIPS subset with types and negative preconditions. We describe PDDL informally and refer the reader to other references for a formal treatment [14]. A PDDL *domain* is characterized by a name, a set of types, a set of predicates, and a set of operators. For example, in the Delivery domain, a robot must pick up newspapers from a home base and then deliver them to certain locations. The domain has two types: `loc` and `paper`. One predicate is `(at ?l - loc)`, where `?l` is a placeholder for a `loc` object. The domain has three operators: `(pick-up ?p - paper ?l - loc)`, `(move ?from - loc ?to - loc)`, `(deliver ?p - paper ?l - loc)`. Each operator has preconditions and effects. For example, the `pick-up` operator is shown in Figure 2.

A PDDL *task* is characterized by a domain, a set of objects, an initial state, and a goal. An object has a name and a type, e.g., `paper1 - paper`. A ground atom is a predicate and a tuple of objects of the appropriate types, e.g., `(unpacked paper1)`. A state consists of a conjunction of ground atoms that are true, assuming all other ground atoms to be false. A goal is a conjunction of ground atoms that must be true in any *goal state*. (More general goal expressions are also possible in PDDL.) For example, in Delivery, the goal may include `(satisfied loc1)` and `(satisfied loc2)`.

```
(:action pick-up
  :parameters (?p - paper
               ?l - loc)
  :precondition (and (at ?l)
    (isHomeBase ?l)
    (unpacked ?p))
  :effect (and (carrying ?p)
    (not (unpacked ?p))))
```

Figure 2: PDDL operator example.

An *action* is an operator and a tuple of objects of the appropriate types, e.g., `(pick-up paper1 loc4)`. The operator's preconditions determine whether the action is applicable and the effects define what ground atoms would be added or deleted if the operator is executed. A *plan* is a finite sequence of actions. The plan is *valid* for a task if all actions are applicable when executed in succession from the initial state and if the final state is a goal state.

PDDL domains, types, predicates, operators, objects, and types often include human-readable names like the ones shown above. These names are not important for standard AI planners or previous generalized planning approaches. However, the names are very important for humans — and, we expect, for LLMs — trying to make sense of the PDDL.

Generalized Planning in PDDL Domains. A generalized planning instance is characterized by a PDDL domain and a distribution of tasks. A small set of *training tasks* (10 or fewer in experiments) from the distribution is given at training time. At evaluation time, a set of held-out *evaluation tasks* — typically involving many more objects — are used to measure performance. The objective is to use the training tasks to synthesize a program that will produce valid plans for all of the evaluation tasks. We consider an evaluation task solved if the program returns a valid plan within a fixed wall-clock time budget (30 seconds in experiments). In other words, we are interested in satisficing, not optimal, planning, and our primary concern is the efficiency of planning itself.

4 Generalized Planning with LLMs

We are interested in the extent to which pretrained large language models (LLMs) can be used for generalized planning in PDDL domains. We assume familiarity with LLMs [10–13]. To use LLMs for generalized planning, we need to define a protocol for prompting.

4.1 Prompting Protocol

Previous work on Chain-of-Thought (CoT) prompting has shown that asking an LLM to "think step by step" can improve performance in reasoning tasks [15]. With these results in mind, we hypothesized that decomposing generalized planning into three stages — domain summarization, strategy proposal, and strategy implementation — would improve performance.

Domain Summarization. Our first prompt to the LLM is shown on the right. To compensate for the limited context window size of transformer-based LLMs like GPT-4, we abbreviate the encoding of the training tasks in two ways. First, we always use

> Domain: [PDDL Domain]
> Example problems: [PDDL Training Tasks]
> Write a short summary of this domain in words.

only two training tasks, even when more are given. Second, within each training task, we limit the number of objects and initial state ground atoms shown. For each object type, if the number of objects of that type exceeds 10, we truncate the object set and add ellipses. Similarly, for each predicate, if the number of ground atoms with that predicate exceeds 10, we truncate and add ellipses. The fact that we only need to communicate the "gist" of the task distribution, rather than whole tasks, is another advantage of generalized planning with LLMs versus *planning* with LLMs.

Strategy Proposal. After the LLM responds to the first prompt, we ask for a generalized planning strategy, as shown on the right. In preliminary experiments, omitting the phrase "without using search" would often lead the LLM to propose a search-based planning strategy. We hypothesize that including this phrase together with the "simple" qualifier is enough to guide the LLM toward efficient solutions. We test this hypothesis in experiments.

> There is a simple strategy for solving all problems in this domain without using search. What is that strategy?

Strategy Implementation. Finally, we ask the LLM to implement the strategy as a Python program:

> Implement the strategy as a Python function. The code should be of the form
>
> ```
> def get_plan(objects, init, goal):
>     # Your code here
>     return plan
> ```
>
> where
>
> - `objects` is a set of (object name, type name) tuples
> - `init` is a set of ground atoms represented as tuples of predicate names and arguments (e.g., ('predicate-foo', 'object-bar', ...))
> - `goal` is also a set of ground atoms represented in the same way
> - `plan` is a list of actions, where each action is a ground operator represented as a string (e.g., '(operator-baz object-qux ...)')

In domains without object types, `objects` is instead just a set of object names.

4.2 Automated Interactive Debugging

After the LLM has proposed an implementation of `get_plan`, we use the training tasks to *validate* the implementation. For each training task, we execute `get_plan` until it returns an output, throws an exception, or reaches a timeout (30 seconds). If the output is a valid plan, we continue onto the next training task. Otherwise, we re-prompt with one of four types of feedback.

Python Exceptions. If executing `get_plan` results in a Python exception, we capture the traceback and report it to the LLM along with the input. An example is shown on the right, with the traceback abbreviated for clarity. In preliminary experiments, we found that including the full traceback, rather than the exception alone, can improve performance.

> Given this task: [PDDL Training Task]
> The code raised the following exception:
>
> ```
> File "<file-name-omitted>", line 86
> lift_at = {atom[1]: atom[2] ...}
>                     ~~~~~~^
> IndexError: tuple index out of range
> ```
>
> Fix the code.

Timeout. If `get_plan` does not finish before the timeout, we report to the LLM that the program did not finish and suggest that an infinite loop may be to blame. We also provide a traceback showing where the program was executing when it was interrupted. An example is shown on the right. The traceback is again abbreviated for clarity. Note that the `KeyboardInterrupt` is automatically thrown after 30 seconds. In

> Given this task: [PDDL Training Task]
> The code raised the following exception:
>
> ```
> File "<file-name-omitted>", line 23
> while not any(span_loc[1] == ...:
> ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
> KeyboardInterrupt
> ```
>
> The code was interrupted because it timed out (possible infinite loop).
> Fix the code.

practice, nearly all timeouts we observe are due to logic errors in the code, rather than inefficient but correct implementations.

Plan Syntax. If `get_plan` returns an output, we check its syntax: whether it is a list of strings, whether each string is enclosed in parentheses and space-separated, and whether the action names, object names, and number of objects per action are valid with respect to the domain and task. If any of these checks fail, we report the failure to the LLM. For this type of failure, we also remind the LLM about the valid operators. An example is shown on the right. The full plan is shown to the LLM but abbreviated in the example for clarity. The issue in this example is that the actions are not enclosed in parentheses.

> Given this task: [PDDL Training Task]
> The code returned this plan:
>
> `['walk r0_c0 r0_c1', 'walk ...]`
>
> However, the action `walk r0_c0 r0_c1` is invalid at step 0. NOTE: the valid operators are: `(climb ?from ?to) (walk ?from ?to)`.
> Fix the code.

Plan Semantics. If all of the previous checks pass, we use the VAL tool [53] to check whether the `get_plan` output is a semantically valid plan. If not, VAL provides "plan repair advice", e.g., if there is an action with invalid preconditions. We extract this plan repair advice and report it to the LLM. An example is shown on the right. The full plan is again shown to the LLM but abbreviated for clarity.

> Given this task: [PDDL Training Task]
> The code failed. It returned the following plan:
>
> `['(pick-up paper-1 loc-0)', ...].`
>
> NOTE: `(pick-up paper-0 loc-0)` has an unsatisfied precondition at time 3
>
> `(Set (at loc-0) to true)`
>
> Fix the code.

Additional Details. After re-prompting the LLM, we repeat the process of checking the code and reporting any failures up to four times. To handle rare cases where the LLM implements its own helper functions and then assumes during debugging that the helper functions are still available, we append each new response from the LLM to a growing Python file, rather than overwriting the previous responses. If a failure is still encountered on the last attempt, the final response is used during evaluation.

5 Experiments and Results

Through experiments, we address these questions: **1.** Can GPT-4 be used for generalized (PDDL) planning? **2.** Are the synthesized programs efficient? **3.** Does CoT summarization help? **4.** Does automated debugging help? **5.** To what extent does GPT-4 rely on names in the PDDL? **6.** How does GPT-4 compare to GPT-3.5? **7.** Do each of the four error types help? **8.** How many training tasks are needed? We address each of these questions quantitatively and report qualitative observations.

5.1 Experimental Setup

We evaluate nine generalized planning approaches on seven PDDL domains over 10 random seeds. Tasks are randomly generated for each seed.

Domains. The first six domains (and tasks) are taken directly from the previous work by Yang et al. [19]. Of these, four (Gripper, Miconic, Ferry, Spanner) are standard planning benchmarks and the other two (Delivery, Forest) were introduced by that work. The last domain (Heavy) is new to this work. The pretraining data for GPT-4 is not publicly available, but it is likely that the *domain definitions* for at least the four standard domains were included in that data. However, we believe it is unlikely that *generalized plans* were included, and for the Heavy domain, we can guarantee that neither the domain nor generalized plans were included. We now briefly describe each domain. Unless otherwise specified, there are 10 training tasks and 30 evaluation tasks per domain and seed.

- **Delivery**: Newspapers at a home base must be delivered to multiple locations. There are five training tasks with 9–17 objects; evaluation tasks have 70–100 objects.
- **Forest**: A hiker must navigate a 2D grid to reach a goal location while climbing hills and avoiding water. A marked trail leads to the goal, but there are shorter paths through dirt. There are 4 training tasks with 64-100 objects; evaluation tasks have 100–144 objects.

Domain	GPT-4	No CoT	No Debug	No Names	GPT-3.5	PG3	PE	PC	Random
Delivery	0.90	0.70	0.10	0.10	0.00	1.00	0.00	0.10	0.00
Forest	1.00	1.00	0.62	0.11	0.32	1.00	1.00	0.16	0.03
Gripper	0.90	0.80	0.50	0.10	0.00	1.00	0.00	0.20	0.00
Miconic	0.01	0.13	0.00	0.00	0.00	1.00	0.00	0.10	0.13
Ferry	0.80	0.20	0.26	0.00	0.00	1.00	0.00	0.90	0.00
Spanner	0.10	0.00	0.00	0.00	0.00	1.00	1.00	0.56	0.06
Heavy	0.60	1.00	0.20	0.00	0.00	0.00	0.00	0.00	0.00

Table 1: Fraction of evaluation tasks solved. All results are averaged over 10 random seeds and 30 evaluation tasks per seed. See Table 3 in the appendix for maximum performance over seeds.

- **Gripper**: Balls must be transported between rooms by a robot with two grippers. Training tasks have 20–30 objects; evaluation tasks have 60–80 objects.
- **Miconic**: Passengers in multiple buildings, each with an elevator, must be picked up and dropped off on different floors. Training tasks have 6–30 objects; evaluation tasks have 11–150 objects.
- **Ferry**: Cars must be sailed between islands using a ferry that can carry at most one car. Training tasks have 13–20 objects; evaluation tasks have 30–50 objects.
- **Spanner**: Wrenches (spanners) and nuts are distributed along a one-way corridor. An agent must move down the corridor, pick up wrenches, and tighten the nuts, using each wrench at most once. Training tasks have 9–15 objects; evaluation tasks have 30–60 objects.
- **Heavy**: Items must be stacked into an empty box. An item can only be stacked on another item if the latter is heavier. The weight relations are expressed via a `(heavier ?x ?y)` predicate. One challenge is in determining which item to place into the box first, i.e., which item is the heaviest. Training tasks have 3–10 objects; evaluation tasks have 100–250 objects.

Approaches. We evaluate the main approach, four ablations, and four baselines. The baselines are taken from the work by Yang et al. [19]; see that work for details.

- **GPT-4**: Our main approach with CoT summarization and automated debugging.
- **No CoT**: An ablation of the main approach that does not use CoT summarization. The three initial prompts are combined and “Write a short summary of this domain in words.” and “What is that strategy?” are removed.
- **No Debug**: An ablation of the main approach that does not use automated debugging. The first implementation of `get_plan` is used for evaluation.
- **No Names**: An ablation of the main approach where all names in the PDDL domains and tasks are replaced with nondescriptive identifiers. For instance, predicates are renamed to `predicate1`, `predicate2`, etc., operators are renamed to `operator1`, `operator2`, etc. Altogether, the names of the domain, problem, predicates, operators, variables, types, and objects are ablated.
- **GPT-3.5**: GPT-3.5 with CoT summarization and automated debugging.
- **PG3**: The generalized planning approach proposed by Yang et al. [19]. The synthesized programs are goal-conditioned policies implemented as lifted decision lists. Synthesis is performed via heuristic search in policy space with their novel heuristic.
- **Policy Evaluation (PE)**: An approach from Yang et al. [19] that is identical to PG3 except that the heuristic used for policy search is sparse: each candidate policy is scored based on the number of training tasks that is fully solves.
- **Plan Compare (PC)**: Another approach from Yang et al. [19] that is identical to PG3 except for the policy search heuristic: example plans for each training task are generated offline, and the policy is scored based on its agreement with the example plans.
- **Random**: Valid actions are randomly sampled and executed until a dead-end is encountered, the goal is reached, or a maximum horizon (default 1000, but see the previous work) is exceeded.

Experimental Details. We used a Macbook Pro laptop with an M1 chip and 64 GB RAM. Since an API for GPT-4 is not publicly available, we used the ChatGPT browser interface for all experiments (including the GPT-3.5 baseline). The pipeline is fully automated except that prompts and responses are manually copied and pasted between the terminal and browser, with the clipboard programmatically updated. To facilitate reproducibility, we have released all chat logs and code.

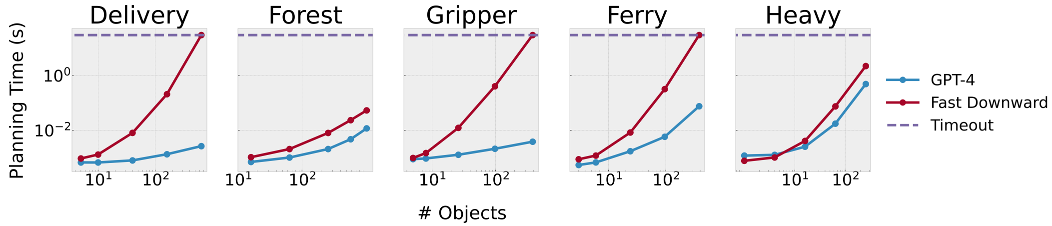

Figure 3: GPT-4 synthesized program runtime compared to a state-of-the-art planner (Fast Downward). Note the log-log axes. Each point is a median over 10 newly generated tasks, over all seeds where generalized planning solved all evaluation tasks.

5.2 Results and Analysis

Main results are presented in Table 1. Examples of synthesized programs are presented in the appendix. Overall, the performance of GPT-4 with CoT summarization and automated debugging is strong in Delivery, Forest, Gripper, Ferry, and Heavy, and poor in Miconic and Spanner. Note that the reported success rates are averaged over all LLM conversations. In practice, performance could be boosted by restarting the conversation multiple times and using the best-found program [12]. The strong performance in Heavy is especially notable. The generalized planning baselines fail in this domain because lifted decision lists are overly restrictive as program representations and cannot discover a concept like "heaviest overall" from pairwise heavier relations. GPT-4's ability to write general Python code is one of its biggest advantages as a generalized planning approach.

We also observe that in nearly all cases, GPT-4 either (1) solves all of the training tasks and then solves all of the evaluation tasks; or (2) fails to solve at least one training task and then fails to solve all of the evaluation tasks. In other words, overfitting to the training tasks is very rare, and evaluation performance is typically all-or-nothing. See Table 3 in the appendix for the maximum fraction of evaluation tasks solved over seeds.

Miconic failures. GPT-4 has a number of consistent failure modes in Miconic. First, at the strategy proposal level, it often fails to recognize that there can be multiple buildings, each with their own elevator. This is admittedly difficult to recognize given the PDDL encoding: buildings exist only implicitly based on the `above` relation between floor objects. For example, one would need to see that neither `(above f1_b1, f1_b2)` nor `(above f1_b2, f1_b1)` are true and conclude that the floors are in two different buildings. However, especially after automated debugging, GPT-4 *can* realize that there are multiple buildings, and furthermore, that building names (e.g., `b1`, `b2`) can be extracted from the floor names. But then other failures often occur, for example, attempting and failing to create a total ordering of the floors from the `above` predicate. Overall, we believe that Miconic is just beyond the limit of GPT-4's current capabilities and would likely be solved by the next generation of LLMs, or by GPT-4 with additional guidance.

Spanner failures. GPT-4 consistently fails in Spanner during strategy proposal. In particular, GPT-4 does not appear to realize that locations in Spanner are connected in a *one-way* chain. The strategy proposed is often "first collect all of the spanners, then tighten all of the nuts" or similar. A correct strategy would instead be to "move to each location in the chain, picking up any spanners and tightening any nuts at each location." Recognizing the existence of the one-way chain requires examining the `link` atoms in the training problems. Even after automated debugging, GPT-4 often assumes, incorrectly, that links are commutative.

Program efficiency. Although we prompt the LLM to implement a "simple" program that does not use search, it is still possible in theory for the LLM to produce a program that does use search or is slow for other reasons (e.g., poor algorithmic complexity). To check whether this is the case, we measure synthesized program runtime. As a baseline for our comparison we use a state-of-the-art domain-independent PDDL planner LAMA [54] via Fast Downward [55], stopping after the first plan is found.[1] In Figure 3, we plot wall-clock runtimes as a function of problem size (number of objects).

[1]Our intention is not to compare planners, but rather to provide a frame of reference for runtime.

Error Type	All	Success	Failure
Python Exception	40.0	28.9	42.5
Plan Semantics	34.0	44.7	31.4
Plan Syntax	13.0	18.4	11.7
Timeout	13.0	8.0	14.4

Table 2: Percentages of error types encountered by GPT-4 in training tasks over all domains and seeds. "All" is the breakdown for all training tasks; "Success" is the breakdown for trials where all evaluation tasks were subsequently solved; "Failure" is the breakdown for the non-Success trials.

Overall, we see that the synthesized programs not only scale favorably with respect to the planner, but also consistently beat the planner in absolute runtime by large margins. This is notable given that the LLM synthesizes Python programs, while the PDDL planner uses a highly optimized combination of Python and C++ code. The bottleneck for Fast Downward is often operator grounding. The LLM's programs do not need to ground operators — they can go directly from task to plan.

The role of CoT. Comparing GPT-4 to No CoT, we see that the impact of CoT summarization is mixed: it seems to help in most cases, but hurt in Miconic and Heavy. Miconic is an especially interesting case. When using CoT summarization, GPT-4 nearly always proposes a "sweep" strategy, where the elevator(s) are first moved to the bottom floor; then moved up one floor at a time until the top floor, picking up and dropping off passengers along the way; then moved down one floor at a time, again picking up and dropping off passengers. This strategy would work in theory, but it requires finding a total ordering of floors within buildings. Without CoT, GPT-4 often attempts a different strategy: pick up, move, and drop off each passenger, one at a time. The latter strategy does not require a total ordering over floors and is arguably simpler to implement in Python. This example shows that CoT can influence the strategy proposed by GPT-4. Moreover, strategies that are "simple" to describe in natural language may not be simple to implement in code. In Heavy, there is not a clear difference in strategies with and without CoT. Since a good strategy is evidently discernible from the PDDL alone, it is possible that CoT "distracts" GPT-4 during implementation.

The role of automated debugging. Comparing GPT-4 to No Debug, we see that automated debugging generally improves performance dramatically. Figure 4 shows that even one step of automated debugging helps substantially, and further steps exhibit diminishing marginal improvements. Table 2 reports the fraction of error types encountered during training across. Python exceptions are most common, followed closely by errors in plan semantics, then errors in plan syntax, and finally timeouts. We also see that the error types are well-distributed within successful trials, suggesting that each of the four types of feedback given during automated debugging are beneficial. In general, GPT-4 tends to make small, local corrections to code during automated debugging. If the code is structurally flawed and requires a significant rewrite, restarting the dialogue from the beginning may be required.

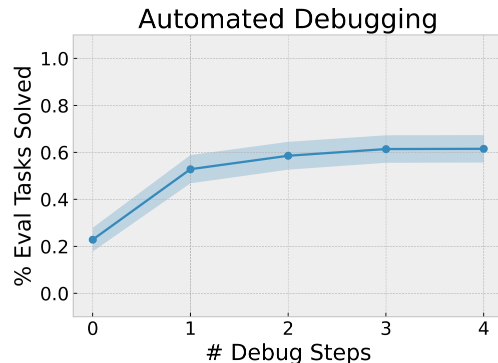

Figure 4: Fraction of evaluation tasks solved by GPT-4 vs. number of debugging steps averaged over all domains. The shaded region is standard error.

The role of PDDL names. Examining the results for the No Names ablation, we see performance overall is very poor. This confirms our hypothesis that the terms present in the PDDL domains and tasks are helpful to the LLM, as they would be to a human. Note that planners like Fast Downward and generalized planners like PG3 would be unaffected by name changes. However, there are a few cases where the No Names ablation does succeed, suggesting that the LLM has some capacity for purely syntactic generalized planning.

GPT-3.5 vs. GPT-4. Examining the results for GPT-3.5, we see that it performs much worse than GPT-4. This is consistent with other reports [13, 56] that GPT-4 is far superior on reasoning and

coding tasks. Qualitatively, the programs proposed by GPT-3.5 are flawed in myriad ways and do not usually appear "close". They also do not seem to improve with automated debugging.

Data efficiency. Figure 5 shows the number of training tasks *used* in successful trials. A training task is used if it appeared in the prompt and/or triggered feedback during automated debugging. Since two training tasks are always used in the prompt, the minimum used is two. Interestingly, in the vast majority of cases, only those two training tasks are used. During automated debugging, these two prompting tasks are always checked first, and most of the time, they are sufficient to identify issues. In a small number of cases, a third task is also used during automated debugging. This result speaks to the strong few-shot learning capabilities of GPT-4. We expect that in many cases, even one training task would suffice, although we did witness a drop in performance in preliminary experiments with one task.

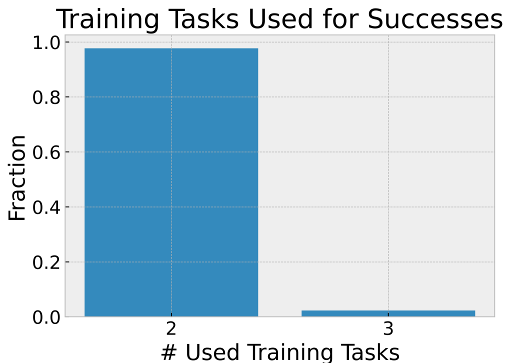

Figure 5: Data efficiency of GPT-4. The reported fractions are over all trials, over all domains.

6 Discussion and Future Work

In this work, we showed that GPT-4 with CoT summarization and automated debugging is a surprisingly strong generalized planner in PDDL domains. We conclude with limitations of this work, reflections about the implications of our findings, and opportunities for future work.

Limitations. A major limitation of this work and previous work on generalized planning is that it is easy enough to hand-design generalized plans for all of the domains considered. Nonetheless, we expect this line of work to be practically useful for at least three reasons. (1) In some cases, it may be considerably easier to specify PDDL domain and problem descriptions than it is to directly specify a generalized plan. (2) In a fully autonomous system, where operators and predicates are learned *in association to natural language*, we would want the system to also synthesize generalized plans autonomously. (3) Beyond PDDL, generalized planning with LLMs would be an even more attractive option, since other approaches rely strongly on formal specifications. Another limitation of this work is our use of training tasks to communicate the task distribution of interest to the LLM. In general, a few example tasks may be insufficient to express the full distribution. Other representations like natural language or procedural generation code may be better, but would require more human input.

Is (generalized) planning now obsolete? No. First, there remains a performance gap between GPT-4 and PG3, and other generalized planners may be even better. However, even if this gap is closed by the next generation of LLMs, we would still say no. *Planning* remains essential in domains where no simple program exists. An interesting direction for future work would be automatically detecting whether a simple program might exist before attempting to synthesize one.[2] *Generalized* planning without LLMs remains important in cases where domain and problem descriptions are not human-readable, e.g., because the predicates and operators are learned (e.g., [57]). Even with natural language descriptions, combining "classical" approaches with LLMs may be best.

What if we gave the LLM access to a planner? Giving an LLM access to APIs is a very powerful idea [58] and one such API could be a PDDL planner [30]. An LLM could potentially use such a planner for generalized planning, especially given that approaches like PG3 rely on access to a planner to generate example plans. In some domains, generating example plans naively would likely confuse the LLM. For example, plans generated in the Forest domain would follow arbitrary paths through the dirt rather than following the slightly longer marked trail. In other cases, though, example plans could be very useful, especially if the LLM generates them in a targeted way.

[2]We tried the Sokoban domain and found that GPT-4 correctly indicates that no simple program exists. However, this property of Sokoban is well-known, so it is likely parroting pretraining data.

References

[1] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[2] Siddharth Srivastava. Foundations and applications of generalized planning. *AI Communications*, 24(4):349–351, 2011.

[3] Blai Bonet and Hector Geffner. Policies that generalize: Solving many planning problems with the same policy. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[4] Sergio Jiménez, Javier Segovia-Aguas, and Anders Jonsson. A review of generalized planning. *The Knowledge Engineering Review*, 34, 2019.

[5] Or Rivlin, Tamir Hazan, and Erez Karpas. Generalized planning with deep reinforcement learning. *arXiv preprint arXiv:2005.02305*, 2020.

[6] Richard S Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M Pilarski, Adam White, and Doina Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 761–768, 2011.

[7] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *International conference on machine learning*, pages 1312–1320. PMLR, 2015.

[8] Vitchyr Pong, Shixiang Gu, Murtaza Dalal, and Sergey Levine. Temporal difference models: Model-free deep rl for model-based control. *arXiv preprint arXiv:1802.09081*, 2018.

[9] Karl Pertsch, Oleh Rybkin, Frederik Ebert, Shenghao Zhou, Dinesh Jayaraman, Chelsea Finn, and Sergey Levine. Long-horizon visual planning with goal-conditioned hierarchical predictors. *Advances in Neural Information Processing Systems*, 33, 2020.

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

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[12] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[13] OpenAI. Gpt-4 technical report, 2023.

[14] Drew McDermott. The 1998 AI Planning Systems competition. *AI Magazine*, 21(2):35–55, 2000.

[15] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

[16] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689*, 2023.

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

[18] Shreyas Sundara Raman, Vanya Cohen, Eric Rosen, Ifrah Idrees, David Paulius, and Stefanie Tellex. Planning with large language models via corrective re-prompting. *arXiv preprint arXiv:2211.09935*, 2022.

[19] Ryan Yang, Tom Silver, Aidan Curtis, Tomas Lozano-Perez, and Leslie Pack Kaelbling. PG3: Policy-guided planning for generalized policy generation. In *IJCAI*, 2022.

[20] Pratyusha Sharma, Antonio Torralba, and Jacob Andreas. Skill induction and planning with latent language. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1713–1726, 2022.

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

[22] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning (ICML)*, 2022.

[23] Bill Yuchen Lin, Chengsong Huang, Qian Liu, Wenda Gu, Sam Sommerer, and Xiang Ren. On grounded planning for embodied tasks with language models. *arXiv preprint arXiv:2209.00465*, 2022.

[24] Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Large language models still can't plan (a benchmark for llms on planning and reasoning about change). *arXiv preprint arXiv:2206.10498*, 2022.

[25] Tom Silver, Varun Hariprasad, Reece S Shuttleworth, Nishanth Kumar, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Pddl planning with pretrained large language models. In *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022.

[26] Vishal Pallagani, Bharath Muppasani, Keerthiram Murugesan, Francesca Rossi, Lior Horesh, Biplav Srivastava, Francesco Fabiano, and Andrea Loreggia. Plansformer: Generating symbolic plans using transformers. *arXiv preprint arXiv:2212.08681*, 2022.

[27] Katherine M Collins, Catherine Wong, Jiahai Feng, Megan Wei, and Joshua B Tenenbaum. Structured, flexible, and robust: benchmarking and improving large language models towards more human-like behavior in out-of-distribution reasoning tasks. *arXiv preprint arXiv:2205.05718*, 2022.

[28] Kevin Lin, Christopher Agia, Toki Migimatsu, Marco Pavone, and Jeannette Bohg. Text2motion: From natural language instructions to feasible plans. *arXiv preprint arXiv:2303.12153*, 2023.

[29] Yaqi Xie, Chen Yu, Tongyao Zhu, Jinbin Bai, Ze Gong, and Harold Soh. Translating natural language to planning goals with large-language models. *arXiv preprint arXiv:2302.05128*, 2023.

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

[31] Richard E Fikes, Peter E Hart, and Nils J Nilsson. Learning and executing generalized robot plans. *Artificial intelligence*, 3:251–288, 1972.

[32] John Levine and David Humphreys. Learning action strategies for planning domains using genetic programming. In *Workshops on Applications of Evolutionary Computation*, pages 684–695. Springer, 2003.

[33] Sergio Jiménez and Anders Jonsson. Computing plans with control flow and procedures using a classical planner. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS-15*, pages 62–69, 2015.

[34] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Computing hierarchical finite state controllers with classical planning. *Journal of Artificial Intelligence Research*, 62:755–797, 2018.

[35] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Generalized planning as heuristic search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 569–577, 2021.

[36] H. Levesque. Planning with loops. In *IJCAI*, 2005.

[37] Siddharth Srivastava, N. Immerman, S. Zilberstein, and Tianjiao Zhang. Directed search for generalized plans using classical planners. In *ICAPS*, 2011.

[38] Elly Zoe Winner. *Learning Domain-Specific Planners from Example Plans*. PhD thesis, Carnegie Mellon University, USA, 2008.

[39] Blai Bonet and Hector Geffner. Features, projections, and representation change for generalized planning. *CoRR*, abs/1801.10055, 2018. URL `http://arxiv.org/abs/1801.10055`.

[40] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023.

[41] Wenqing Zheng, SP Sharan, Ajay Kumar Jaiswal, Kevin Wang, Yihan Xi, Dejia Xu, and Zhangyang Wang. Outline, then details: Syntactically guided coarse-to-fine code generation. *arXiv preprint arXiv:2305.00909*, 2023.

[42] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.

[43] Shima Imani, Liang Du, and Harsh Shrivastava. Mathprompter: Mathematical reasoning using large language models, 2023.

[44] Chunqiu Steven Xia and Lingming Zhang. Conversational automated program repair, 2023.

[45] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[46] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R. Bowman, Kyunghyun Cho, and Ethan Perez. Improving code generation by training with natural language feedback, 2023.

[47] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022.

[48] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*, 2022.

[49] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[50] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[51] Stephen Muggleton. Inductive logic programming. *New generation computing*, 8:295–318, 1991.

[52] Andrew Cropper and Sebastijan Dumančić. Inductive logic programming at 30: a new introduction. *Journal of Artificial Intelligence Research*, 74:765–850, 2022.

[53] Richard Howey, Derek Long, and Maria Fox. Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl. In *16th IEEE International Conference on Tools with Artificial Intelligence*, pages 294–301. IEEE, 2004.

[54] Silvia Richter and Matthias Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

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

[56] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[57] Tom Silver, Rohan Chitnis, Nishanth Kumar, Willie McClinton, Tomas Lozano-Perez, Leslie Pack Kaelbling, and Joshua Tenenbaum. Predicate invention for bilevel planning. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2023.

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

Domain	GPT-4	No CoT	No Debug	No Names	GPT-3.5
Delivery	1.00	1.00	1.00	1.00	0.00
Forest	1.00	1.00	1.00	0.93	1.00
Gripper	1.00	1.00	1.00	1.00	0.00
Miconic	0.07	0.00	0.00	0.00	0.00
Ferry	1.00	1.00	1.00	0.00	0.00
Spanner	1.00	1.00	0.00	0.00	0.00
Heavy	1.00	1.00	1.00	0.00	0.00

Table 3: Max (over seeds) fraction of evaluation tasks solved.

A Additional Results

Table 3 reports the maximum fraction of evaluation tasks solved over seeds. In most cases, the LLM either solves all or none of the evaluation tasks. This suggests that the LLM does not overfit to the training tasks, even though only a very small number of them are used (see main text).

B Example Programs

Examples of programs synthesized by GPT-4 with CoT summarization and automated debugging are shown in Listings 1, 2, 3, 4, 5, 6, and 7. Any comments in the code are added by GPT-4 itself.

```
def get_plan(objects, init, goal):
    locs = [obj[0] for obj in objects if obj[1] == "loc"]
    papers = [obj[0] for obj in objects if obj[1] == "paper"]

    home_base = next(loc for loc in locs if ("ishomebase", loc) in init)
    safe_locs = [loc for loc in locs if ("safe", loc) in init]
    wants_paper = [loc for loc in locs if ("wantspaper", loc) in init]

    plan = []

    for loc in wants_paper:
        if loc in safe_locs:
            # Pick up a newspaper at the home base
            paper = papers.pop()
            plan.append(f"(pick-up {paper} {home_base})")

            # Move to the safe location that wants a newspaper
            plan.append(f"(move {home_base} {loc})")

            # Deliver the newspaper to the location
            plan.append(f"(deliver {paper} {loc})")

            # Move back to the home base
            plan.append(f"(move {loc} {home_base})")

    return plan
```

Listing 1: **Delivery** domain example synthesized program (success)

```python
def get_plan(objects, init, goal):
    def find_location(init):
        for atom in init:
            if atom[0] == "at":
                return atom[1]
        return None

    def find_adjacent_trail(current_loc, init):
        for atom in init:
            if atom[0] == "ontrail" and atom[1] == current_loc:
                return atom[2]
        return None

    def is_hill(loc, init):
        for atom in init:
            if atom[0] == "ishill" and atom[1] == loc:
                return True
        return False

    current_loc = find_location(init)
    goal_loc = find_location(goal)
    plan = []

    while current_loc != goal_loc:
        next_loc = find_adjacent_trail(current_loc, init)

        if is_hill(next_loc, init):
            action = f"(climb {current_loc} {next_loc})"
        else:
            action = f"(walk {current_loc} {next_loc})"

        plan.append(action)
        current_loc = next_loc

    return plan
```

Listing 2: **Forest** domain example synthesized program (success)

```python
def get_plan(objects, init, goal):
    def get_objects_of_type(type_name):
        return [obj for obj in objects if obj.startswith(type_name)]

    def get_location(state, ball):
        for atom in state:
            if atom[0] == 'at' and atom[1] == ball:
                return atom[2]
        return None

    balls = get_objects_of_type('ball')
    rooms = get_objects_of_type('room')
    grippers = get_objects_of_type('gripper')

    robby_location = None
    for atom in init:
        if atom[0] == 'at-robby':
            robby_location = atom[1]
            break

    plan = []
    for ball in balls:
        initial_location = get_location(init, ball)
        goal_location = get_location(goal, ball)

        if goal_location is None:
            continue

        if initial_location != goal_location:
            plan.append(f'(move {robby_location} {initial_location})')
            robby_location = initial_location

            plan.append(f'(pick {ball} {initial_location} {grippers[0]})')

            plan.append(f'(move {robby_location} {goal_location})')
            robby_location = goal_location

            plan.append(f'(drop {ball} {goal_location} {grippers[0]})')

    return plan
```

Listing 3: **Gripper** domain example synthesized program (success)

```
def get_plan(objects, init, goal):
    floors_by_building = {}
    passengers_by_building = {}

    for obj, obj_type in objects:
        if obj_type == "floor":
            building = obj.split("_")[-1]
            if building not in floors_by_building:
                floors_by_building[building] = []
            floors_by_building[building].append(obj)
        elif obj_type == "passenger":
            building = obj.split("_")[-1]
            if building not in passengers_by_building:
                passengers_by_building[building] = []
            passengers_by_building[building].append(obj)

    for building in floors_by_building:
        floors_by_building[building] = sorted(floors_by_building[building])

    origins = {p: f for p, f in (atom[1:] for atom in init if atom[0] == "origin")}
    destins = {p: f for p, f in (atom[1:] for atom in init if atom[0] == "destin")}
    lift_at = {atom[1]: atom for atom in init if atom[0] == "lift-at"}

    plan = []
    boarded = set()

    for building, floors in floors_by_building.items():
        passengers = passengers_by_building[building]
        current_lift_at = lift_at[building]

        for floor in floors + list(reversed(floors)):
            if current_lift_at != floor:
                if current_lift_at < floor:
                    plan.append(f"(up {current_lift_at} {floor})")
                else:
                    plan.append(f"(down {current_lift_at} {floor})")
                current_lift_at = floor

            for passenger in passengers:
                if origins.get(passenger) == floor:
                    plan.append(f"(board {floor} {passenger})")
                    boarded.add(passenger)

                if destins.get(passenger) == floor and passenger in boarded:
                    plan.append(f"(depart {floor} {passenger})")
                    boarded.remove(passenger)

    return plan
```

Listing 4: **Miconic** domain example synthesized program (failure)

```python
def get_plan(objects, init, goal):
    plan = []

    # Helper function to find a car's location
    def get_location(car, state):
        for atom in state:
            if atom[0] == 'at' and atom[1] == car:
                return atom[2]
        return None

    cars = [obj for obj in objects if obj.startswith('c')]
    ferry_location = [atom[1] for atom in init if atom[0] == 'at-ferry'][0]

    for car in cars:
        initial_location = get_location(car, init)
        goal_location = get_location(car, goal)

        # Sail to the car's initial location
        if ferry_location != initial_location:
            plan.append(f'(sail {ferry_location} {initial_location})')
            ferry_location = initial_location

        # Board the car onto the ferry
        plan.append(f'(board {car} {initial_location})')

        # Sail to the car's goal location
        if ferry_location != goal_location:
            plan.append(f'(sail {ferry_location} {goal_location})')
            ferry_location = goal_location

        # Debark the car at the goal location
        plan.append(f'(debark {car} {goal_location})')

    return plan
```

Listing 5: **Ferry** domain example synthesized program (success)

```
def get_plan(objects, init, goal):
    plan = []
    bob_location = get_bob_location(init)
    spanner_locations = get_spanner_locations(init)
    nuts_to_tighten = {n for pred, *args in goal if pred == 'tightened' for n in args}

    while nuts_to_tighten:
        closest_spanner = None
        min_distance = float('inf')

        for loc, spanner in spanner_locations.items():
            distance = get_shortest_path(init, bob_location, loc)
            if distance < min_distance:
                min_distance = distance
                closest_spanner = spanner
                closest_spanner_loc = loc

        # Move Bob to the closest spanner
        path_to_spanner = get_path(init, bob_location, closest_spanner_loc)
        for next_location in path_to_spanner:
            plan.append(f'(walk {bob_location} {next_location} bob)')
            bob_location = next_location

        # Pick up the spanner
        plan.append(f'(pickup_spanner {closest_spanner_loc} {closest_spanner} bob)')

        # Move Bob to the gate
        gate = 'gate'
        path_to_gate = get_path(init, bob_location, gate)
        for next_location in path_to_gate:
            plan.append(f'(walk {bob_location} {next_location} bob)')
            bob_location = next_location

        # Tighten a nut using the spanner
        nut = nuts_to_tighten.pop()
        plan.append(f'(tighten_nut {gate} {closest_spanner} bob {nut})')

        # Remove the used spanner from the available spanners
        if closest_spanner_loc in spanner_locations:
            del spanner_locations[closest_spanner_loc]

    return plan

def get_path(init, start, end):
    paths = [start]
    while paths[-1] != end:
        for pred, l1, l2 in init:
            if pred == 'link' and paths[-1] == l1 and l2 not in paths:
                paths.append(l2)
                break
    return paths[1:]
```

Listing 6: **Spanner** domain example synthesized program (failure)

```python
def extract_heavier_relations(init):
    heavier_relations = set()
    for atom in init:
        if atom[0] == 'heavier':
            heavier_relations.add(atom)
    return heavier_relations

def sort_objects_by_weight(objects, heavier_relations):
    sorted_objects = []
    while len(objects) > 0:
        for obj in objects.copy():
            if all(('heavier', other, obj) not in heavier_relations for other in objects):
                sorted_objects.append(obj)
                objects.remove(obj)
    return sorted_objects

def get_plan(objects, init, goal):
    heavier_relations = extract_heavier_relations(init)
    sorted_objects = sort_objects_by_weight(set(objects), heavier_relations)

    plan = []
    first_object = sorted_objects.pop(0)
    plan.append(f"(pack-first {first_object})")

    for obj in sorted_objects:
        plan.append(f"(stack {first_object} {obj})")
        first_object = obj

    return plan
```

Listing 7: **Heavy** domain example synthesized program (success)