PDDLEGO: Iterative Planning in Textual Environments

Li Zhang[1*] **Peter Jansen**[3] **Tianyi Zhang**[1]
Peter Clark[2] **Chris Callison-Burch**[1] **Niket Tandon**[2]
[1]University of Pennsylvania [2]Allen Institute for Artificial Intelligence
[3]The University of Arizona
{zharry}@upenn.edu {nikett}@allenai.org

Abstract

Planning in textual environments have been shown to be a long-standing challenge even for current models. A recent, promising line of work uses LLMs to generate a formal representation of the environment that can be solved by a symbolic planner. However, existing methods rely on a fully-observed environment where all entity states are initially known, so a one-off representation can be constructed, leading to a complete plan. In contrast, we tackle partially-observed environments where there is initially no sufficient information to plan for the end-goal. We propose PDDLEGO that **iteratively** construct a planning representation that can lead to a partial plan for a given sub-goal. By accomplishing the sub-goal, more information is acquired to augment the representation, eventually achieving the end-goal. We show that plans produced by few-shot PDDLEGO are 43% more efficient than generating plans end-to-end on the Coin Collector simulation, with strong performance (98%) on the more complex Cooking World simulation where end-to-end LLMs fail to generate coherent plans (4%).[1]

1 Introduction

Planning with LLMs has witnessed a surge of interest in the NLP community, not only because it showcases AI systems' ability to reason about complex events, but also because of the need of many downstream applications like goal-driven robotics (Huang et al., 2022a,b) and intelligent planning assistants (Lyu et al., 2021). The most intuitive approach of this task is using LLMs as planners to produce a sequence of actions executed to arrive at a goal state (Valmeekam et al., 2023a; Stein and Koller, 2023). While applicable in many domains, this LLM-based approach is found to underperform in textual simulated environments (Valmeekam et al., 2023c,b) and to lack interpretability compared to symbolic planning methods that derive a plan from a formal representation of the environment. We join the efforts that combine both approaches, effectively translating the textual input into a symbolic representation expressed in the planning domain definition language (PDDL) (see Appendix A for an introduction), which can then be solved by a symbolic planner (Collins et al., 2022; Lyu et al., 2023; Liu et al., 2023; Xie et al., 2023; Wong et al., 2023). This neurosymbolic approach has gained popularity as it combines LLMs' flexibility to understand rich NL and classical planners' determinism and faithfulness.

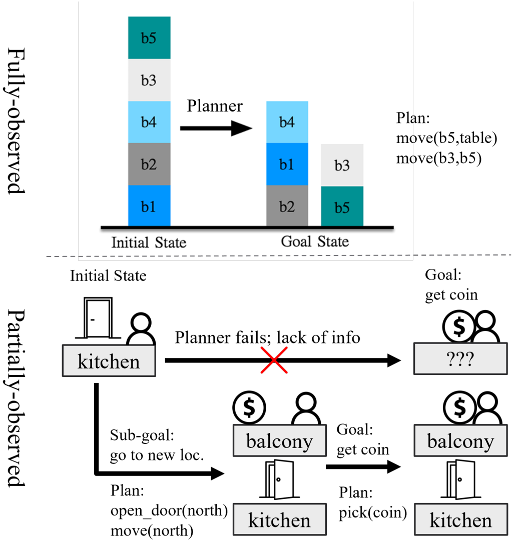


Figure 1: A fully-observed environment like BlocksWorld (upper, to rearrange objects from and to a given configuration) can be tackled by generating a PDDL problem file, while a partially observed one like Coin Collector (lower, to look for an object in an unknown location) cannot until sufficient exploration.

All previous work on LLM generating PDDL has only experimented on **fully-observed** environments where all entity states are initially known, thus requiring no exploration. Take BlocksWorld, a

[*] Work done as an intern at AI2.
[1] Our code can be found at https://github.com/zharry29/nl-to-pddl.

common benchmark for such work, as an example (Figure 1, upper), both the initial and goal states are initially spelled out, in which case the LLM's job is akin to translating the textual descriptions of the environment into a PDDL problem file which specifies the initial and goal entity states. Assuming also a domain file, a one-off plan can be found and executed to reach the end-goal. In contrast, many real-world environments are **partially-observed** (Figure 1, lower), where the entity states dynamically get uncovered during exploration. Since the necessary initial and goal states might also be unknown (e.g., looking for an item without knowing where it is), the previous approach falls apart due to the impossibility to specify a complete problem file. This causes a chicken-and-egg problem where a plan is required for exploration, while exploration is required to build PDDL that results in a plan. Given this challenge, past work on partially-observed environments has only used LLMs to directly generate plans (Shinn et al., 2023; Majumder et al., 2023), but not a planning representation.

To break the above stalemate, we propose PDDLEGO, a methodology to use LLMs to iteratively build a PDDL problem file from textual observations from the environment. In this problem file, the initial states (or rather current states) reflect the current knowledge of the environment, while the goal states can be dynamically adjusted. In case the problem file does not contain sufficient information to plan for the end-goal (e.g., find a coin), PDDLEGO recursively falls back to a provided sub-goal (e.g., go to an unvisited room). This way, a plan can be found to reach the sub-goal, leading to new observations by exploring the environment, and iteratively refine the problem file until a plan can be found for the end-goal.

We evaluate PDDLEGO on benchmarks of textual interactive virtual environments akin to the robotic planning simulations where PDDL is known for. Our PDDL-induced plans are 43% more efficient than LLMs generating plans directly on the Coin Collector simulation. On one setting of the more complex Cooking World simulation PDDLEGO achieves near-perfect 98% success rate where LLMs that predict action achieves only 4%, while on a more challenging setting, 46% over 0%.

2 Methodology

Our approach is illustrated in Figure 2. We operate in a partially-observed, textual, simulated environment which functions as a multi-turn interaction between the environment and the agent (e.g., *a game to find an item*). Specifically, the environment provides an observation (*objects in a room*) along with a list of permitted actions (*move, pick up*). Then, the agent selects on of these actions, and repeats. The environment can be seen as a finite state machine where each state consists of the conjunction of all entity states and determines the permitted actions. The agent succeeds when a goal state is reached (*the sought item is in hand*); it fails when it cannot possibly reach goal state.

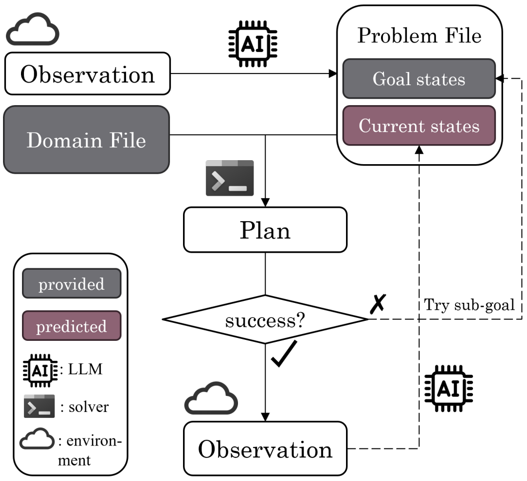


Figure 2: The pipeline of PDDLEGO. A PDDL problem file is iteratively built during exploration.

Like most prior work in using LLMs to generate a planning representation like PDDL, we assume that a domain file that defines the available actions is provided; this domain file can solve a problem file that defines the initial and goal entity states (*where the agent is, where the item is, how are these two locations connected*) when possible to result in a plan (*go west, pick up item*). We also assume a sub-goal structure, namely, an array of goal states defined in PDDL that a model can fall back to when the current goal is unattainable.

Formally, we are initially presented with the first observation o_1 with the end-goal G. We use an LLM to construct an initial problem file PF_1 ({current states, goal states}) to plan for this end-goal.

$$PF_1 = \{LLM(o_1), G\} \quad (1)$$

If this problem file can be solved by the provided domain file with a solver, a plan containing one or more actions is found.

$$Plan_1 := (a_1^1, a_1^2, \dots) = solver(DF, PF_1) \quad (2)$$

If a plan cannot be found due to a lack of information in the problem file, the goal G is swapped out by an immediate sub-goal G', and the solver retries. The actions in the plan are then sequentially executed in the current environment E, resulting in a list of new observations.

$$(E, o_2^1, o_2^2, \dots) = exec(E, a_1^1, a_1^2, \dots) \quad (3)$$

Thus begins the second iteration. Using the new observations, the previous problem file is regenerated (referred to as **PDDL-gen**).

$$PF_2 = \{LLM(PF_1, o_2), G\} \quad (4)$$

The process goes on until one observation fulfills the termination condition.

Unlike prior work that generates the problem file once, PDDLEGO's having LLMs iteratively generating the problem file often result in inconsistencies and errors (e.g., missing a connectivity relation between two rooms, using the name a room in a relation without declaring the room, missing a parenthesis, etc.). To tackle this, we have the LLMs only predict the change in the problem file (i.e., the change of entity states), which we deterministically applied to the previous problem file (referred to as **PDDL-edit**).

$$\Delta_2 = LLM(PF_1, o_2),\ PF_2 = PF_1 + \Delta_2 \quad (4')$$

We will compare our two approaches above with the baseline where LLMs directly generate an action (referred to as **Action-gen**).

$$Plan_i = LLM(o_i) \quad (2')$$

3 Environments

We experiment with two goal-oriented, partially-observed simulated environments, or text games, that span a variety of difficulty and flavor.

Coin Collector (Yuan et al., 2019) focuses on navigation, which is an indispensable element of most simulations. The agent's task is to explore rooms, some connected by locked doors, and find a coin, similar to the running example above. Just as previously discussed, the previous approach on generating a PDDL problem file cannot be applied to Coin Collector because the location of the coin is unknown until the agent enters the same room as the coin. Therefore, the sub-goal structure for this tasks is defined as:

1. pick up coin (requires the location of the coin)
2. go to a room that has not been visited (reveals location of the coin)

The sub-goal of "going to an unvisited room" results in monotonously increasing progress to the end-goal of "finding the coin". In similar search-related tasks, this singular sub-goal or strategy suffices, though it may not work for all situations.

Cooking World (Madotto et al., 2020) subsumes Coin Collector with more complex tasks. The agent' task is to first explore rooms to find ingredients required by a recipe, much like Coin Collector. Next, it should cook the ingredient in some specified location using some specified appliance. Finally, when all ingredients are cooked correctly, a meal can be successfully prepared. Therefore, the sub-goal structure for this tasks is defined as:

1. prepare meal (requires having obtained each ingredient and located relevant appliances)
2. pick up each ingredient (requires the location of each ingredient; obtains ingredients)
3. go to a room that has not been visited (reveals location of ingredients and appliances)

To better understand these simulations, example trajectories are shown in Appendix D.

4 Evaluation

For both simulations, we use the implementation from Jansen and Côté (2022). For Coin Collector, we use the most complex setting; for Cooking World, we consider an easy and a hard setting with varying number of locations and ingredients. See more details in Appendix C. For the choice of LLM, we consider `gpt-3.5-turbo-1106` (GPT 3.5 Turbo) and `gpt-4-1106-preview` (GPT 4 Turbo) across baseline methods (i.e., Action-gen, PDDL-gen, and PDDL-edit). For Action-gen, we prompt the LLM with a full description of the simulation, and for PDDL methods, with a hand-annotated domain file containing well-defined actions. For the PDDL-edit setting, we prompt the LLM to generate templated edits (add, replace, and delete lines in the problem file). The prompt of each method include a 1-shot demonstration of the output format. See details of prompt design and domain file annotation in the Appendix B.

Regarding **performance**, Table 1 shows a drastic performance degradation of Action-gen moving from Coin Collector (only 2 valid actions: move, open door each with 4 direction arguments) to the much more complex Cooking World (with 8 more actions with infinite possible arguments, like processing an ingredient). Moreover, in Cooking World, an agent would fail if an ingredient is processed incorrectly (e.g., fried instead of grilled, was not chopped before roasted). Therefore, LLMs generating actions on the fly are more likely to make irrevocable mistakes and fail the task. In con-

	random	GPT 3.5 Turbo			GPT 4 Turbo		
		Action-gen	PDDL-gen†	PDDL-edit†	Action-gen	PDDL-gen†	PDDL-edit†
Coin	4%	68%	26%	28%	**94%**	58%	78%
Cooking (easy)	0%	0%	70%	68%	4%	94%	**98%**
Cooking (hard)	0%	0%	4%	6%	0%	16%	**46%**

Table 1: The percentage where the agent succeeds by taking no more than the maximum steps on the test set. The † sign specifies methods under our proposed PDDLEGO methodology.

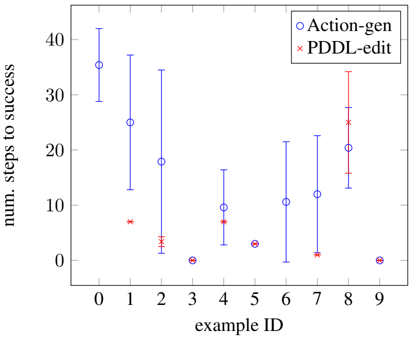


Figure 3: On Coin Collector, the mean and standard deviation of number of steps to success (less is better) for each development example, each over 5 trials with different random seeds of `gpt-4-1106-preview`, comparing Action-gen and PDDL-edit. The error bar represents the sample standard deviation. On example 0 and 6, PDDL-edit fails and thus not shown.

trast, our two-stage PDDL generation approaches ensure the correctness of the plan to process the ingredients (in the second stage) *assuming* that the ingredients are gathered and that the appliances are identified (in the first stage). Logically, the failures of PDDLEGO indicates an inconsistency between the environmental observation and the problem file. For example, the connectivity of the rooms may not be updated correctly upon entrance to a new room, causing no plan or invalid plans to be found. By lessen the burden on LLMs, PDDL-edit notably ameliorates but cannot eliminate this issue. On Coin Collector, issues frequently arise in a loop, where opening a new door leads to a visited room. Notably, GPT3.5 is far worse than GPT4 in generating PDDL, in line with the observations by Zhang et al. (2024) and Silver et al. (2023).

Regarding **efficiency**, Figure 3 shows that on Coin Collector, PDDL-edit is no less efficient than Action-gen on 7 out 8 examples (red crosses are often lower than the blue circles) in the development set where PDDL-edit terminates successfully. Scaling up to the entire test set, with GPT4, PDDL-edit has an average step to success of 7.8 compared to Action-gen's 13.6 among successful attempts, a 43% improvement on efficiency. Among these steps, 3.3 of Action-gen are invalid (e.g., moving through a closed door) compared to merely 0.2 of PDDLEGO, a significant difference when trials and errors are expensive. PDDLEGO also shows better **stability**. In Figure 3, PDDL-edit exhibits a much smaller variance across runs than Action-gen. For example, if the coin happens to be immediately to the west of the initial room, deciding to go west initially would result in a prompt success, while exploring the east portion initially would result in a notable detour. Our approach of PDDL generation leaves only the task of parsing environmental configuration to the LLM, while the planning task is done deterministically by the solver, leading to more consistent plans across runs.

Regarding **interpretability** and **correctability**, the black-box nature of LLMs results in no faithful interpretation behind the decisions (c.f., thought-process). In Coin Collector, for example, if the coin has not be found at the maximum permitted steps, a problematic Action-gen trajectory is almost impossible to manually correct unless a human is to plot a map and keep track of the exploration. On the other hand, both PDDL-gen and PDDL-edit guarantees the correctness of the plan assuming that the generated or edited problem file is correct. Hence, upon failure, a human only needs to inspect and correct the most recent observation and the PDDL. For PDDL-edit, the job is even easier as only the change in the problem needs to be considered. An example learned problem file can be found in Appendix E.

5 Conclusion

We propose PDDLEGO, the first approach to use LLMs to iteratively learn a planning representation while exploring partially-observed environments. We quantitatively show the improvement of performance, efficiency and stability, while qualitatively argue the benefit of interpretability and correctability. Future work might remove the assumption of a domain file and a sub-goal structure.

Limitations

Despite the many benefits of PDDLEGO, it also poses the following shortcomings compared to having LLMs directly generating the plan or actions.

The first is speed and cost, as both the input and output become much longer to include PDDL code. For the OpenAI model we experiment with, PDDL-gen and PDDL-edit are on average about 5x slower than Action-gen. On the other hand, it is difficult to compare the cost which is highly dependent on prompt design. In our work, Action-gen keeps appending the chosen action, new observation and valid actions to the prompt, resulting in a longer input and higher cost for every exploration step. However, our PDDL methods only retain the most recent observation and problem file, so the input length, though initially longer, is roughly constant.

The second is flexibility, which is the strongsuit of methods leveraging LLMs to do most of the work. For each environment we experiment with, a certain extent of hard-coding is required for our methods to work, hindering generalization. In our case, the domain file and sub-goals of one or more problem file for each environment must be manually annotated. Doing so presumes some prior insight into the environment, and therefore PDDLEGO is not truly a zero-shot methodology.

While the aim of this work is to show the preliminary gains of generating PDDL while exploring partially-observed environments, there could be stronger Action-gen baselines, such as using chain-of-thought to formulate a plan first instead of selecting actions on the fly, or more advanced methods in the literature.

Acknowledgements

This work is supported in part by the DARPA KAIROS Program (contract FA8750-19-2-1004), AFRL (contract FA8750-23-C-0507), the Office of the Director of National Intelligence (ODNI) via the IARPA HIATUS Program (contract 2022-22072200005), the NSF (Award 1928631), and gifts from Roblox and Salesforce. Approved for Public Release, Distribution Unlimited. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, ODNI, IARPA, NSF, AFRL, the U.S. Government, or of Roblox or Salesforce. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

References

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

Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. 1998. PDDL - the planning domain definition language. Technical Report "CVC TR-98-003/DSC TR-1165", Yale Center for Computational Vision and Control.

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

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. 2022b. Inner monologue: Embodied reasoning through planning with language models. In *arXiv preprint arXiv:2207.05608*.

Peter A. Jansen and Marc-Alexandre Côté. 2022. Textworldexpress: Simulating text games at one million steps per second. *arXiv*.

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

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. *arXiv preprint arXiv:2301.13379*.

Qing Lyu, Li Zhang, and Chris Callison-Burch. 2021. Goal-oriented script construction. In *Proceedings of the 14th International Conference on Natural Language Generation*, pages 184–200, Aberdeen, Scotland, UK. Association for Computational Linguistics.

Andrea Madotto, Mahdi Namazifar, Joost Huizinga, Piero Molino, Adrien Ecoffet, Huaixiu Zheng, Alexandros Papangelis, Dian Yu, Chandra Khatri, and Gokhan Tur. 2020. Exploration based language learning for text-based games. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 1488–1494. International Joint Conferences on Artificial Intelligence Organization. Main track.

Bodhisattwa Prasad Majumder, Bhavana Dalvi Mishra, Peter Jansen, Oyvind Tafjord, Niket Tandon, Li Zhang, Chris Callison-Burch, and Peter Clark. 2023. Clin: A continually learning language agent for rapid task adaptation and generalization.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning.

Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B. Tenenbaum, Leslie Pack Kaelbling, and Michael Katz. 2023. Generalized planning in pddl domains with pretrained large language models.

Katharina Stein and Alexander Koller. 2023. Autoplanbench:: Automatically generating benchmarks for llm planners from pddl. *arXiv preprint arXiv:2311.09830*.

Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. 2023a. Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. 2023b. On the planning abilities of large language models–a critical investigation. *arXiv preprint arXiv:2305.15771*.

Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. 2023c. Large language models still can't plan (a benchmark for llms on planning and reasoning about change).

Lionel Wong, Jiayuan Mao, Pratyusha Sharma, Zachary S Siegel, Jiahai Feng, Noa Korneev, Joshua B Tenenbaum, and Jacob Andreas. 2023. Learning adaptive planning representations with natural language guidance. *arXiv preprint arXiv:2312.08566*.

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

Xingdi Yuan, Marc-Alexandre Côté, Alessandro Sordoni, Romain Laroche, Remi Tachet des Combes, Matthew Hausknecht, and Adam Trischler. 2019. Counting to explore and generalize in text-based games.

Tianyi Zhang, Li Zhang, Zhaoyi Hou, Ziyu Wang, Yuling Gu, Peter Clark, Chris Callison-Burch, and Niket Tandon. 2024. Proc2pddl: Open-domain planning representations from texts. In *Proceedings of the 2st Workshop on Natural Language Reasoning and Structured Explanations (NLRSE)*, Bangkok, Thailand. Association for Computational Linguistics.

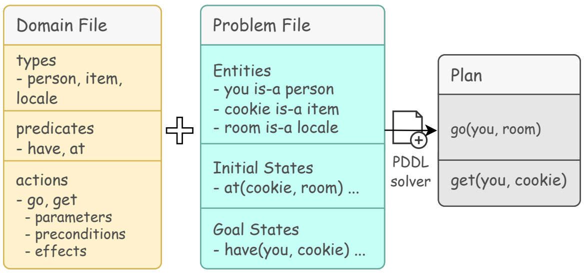


Figure 4: A PDDL solver produces a plan based on a minimal domain file and problem file. Previous work assumes the domain file as given, while we predict the action definitions in the domain file.

A Formulation of PDDL

As shown in Figure 4, an instance of PDDL (Ghallab et al., 1998) consists of a domain file, describing the actions, and a problem file, describing the initial and goal states of entities. A well-formed pair of domain and problem files can be solved by a symbolic planner, whose output is a sequence of actions.

B Annotated Domain Files and Prompts

PDDLEGO is a method to iteratively construct problem files based on a provided domain file. Figure 5 and 6 show the annotated domain files for Coin Collector and Cooking World, respectively. Note that the actions and their parameter lists in the domain file strictly maps to the permitted actions in the simulations, so that a PDDL plan can be mapped onto executable actions in the environment. Based on the domain file, our prompts for either generating (PDDL-gen) or editing (PDDL-edit) the problem file are simply (for Coin Collector):

> You will continue to build a PDDL representation of an environment while exploring it. We will be using the following domain file: «domain file» For example, for the given observation:
>
> You are in the kitchen. To the South you see a closed wooden door.
>
> Your task is to go to a location you have not been yet. You will generate the following problem file: «example domain file»
>
> Now, let's start afresh.

For PDDL-edit, a few more details are appended.

> «the above prompt»
>
> Let's work with an example. Say you're given this observation: You are in the kitchen. To the South you see a closed wooden door. To the East you see a closed glass door.
>
> You will modify the above problem file using add, replace, and delete operations (in a JSON

> format). You SHOULD NOT provide a problem file directly.
>
> ```
> {
>   "objects": {
>     "add": [
>       "loc1 - location",
>       "loc2 - location"
>     ],
>     "replace": {},
>     "delete": []
>   },
>   "init": {
>     "add": [
>       "(connected kitchen loc1 south)",
>       "(closed_door kitchen loc1)",
>       "(connected kitchen loc2 east)",
>       "(closed_door kitchen loc2)"
>     ],
>     "replace": {},
>     "delete": []
>   }
> }
> ```
>
> Note a couple of things:
>
> 1. When you see a closed door, you would use a placeholder for the room behind the door.
>
> 2. When you enter a room, you learn the name of the room and will replace the placeholder with the name. You should also make sure to replace that name for all relations under "init".
>
> 3. When you enter a room, you're "at" the room and it becomes "visited". You should also delete other "at" conditions because you can only be at one room.
>
> 4. You should never delete the "visited" relations, because once a room is visited, it will remain that way.

For Cooking World, the prompt is mostly the same for the first stage (looking for ingredients), with an additional LLM instance to identify closed containers, and their contents once opened. As described above, all found ingredients are mechanically picked up (hard-coded).

For Action-gen, the prompt is simply a description of the simulation, providing as much information as specified in the above domain files. For Coin Collector, it is:

> You will play a game where your goal is to collect a coin. You need to move through rooms explore them. Sometimes, two rooms are connected by closed door that you need to open before you can go from one to another. You should also keep track of which room you have visited, and the direction at which you enter a room.
>
> I will provide you with a description of the environment, and you will take one of the valid actions. Ready?

For Cooking World, it is:

> You will play a game where your goal is to read a recipe, find ingredients, cook a meal, and eat the meal. The recipe includes the ingredients that you'll need to collect. The ingredients are scattered around rooms and may be found in containers. After you find the ingredients, you need to process them as required in the recipe. Here are how the ingredients are processed:
>
> - slice: use a knife to slice the ingredient
>
> - chop: use a knife to chop the ingredient
>
> - dice: use a knife to dice the ingredient
>
> - grill: use a toaster or a barbeque to cook the ingredient will grill it
>
> - roast: use an oven to cook the ingredient will roast it
>
> - fry: use a stove to cook the ingredient will fry it
>
> You have to process the ingredients as specified in the recipe, otherwise you will fail. Once the ingredients are processed, you can cook the meal and eat the meal in the kitchen, so make sure you go back to the kitchen at that point.
>
> Now, I will provide you with a description of the environment, and you will take one of the valid actions. Ready?

```
(define (domain environment)
  (:requirements :strips :typing :negative-preconditions :disjunctive-
        preconditions)
  (:types
    location
    direction
  )
  (:predicates
    (at ?loc - location)
    (visited ?loc - location)
    (connected ?loc1 - location ?loc2 - location ?dir - direction)
    (closed_door ?loc1 - location ?loc2 - location)
  )

  (:action move
    :parameters (?loc1 - location ?loc2 - location ?dir - direction)
    :precondition (and (at ?loc1) (connected ?loc1 ?loc2 ?dir) (not (
          closed_door ?loc1 ?loc2)))
    :effect (and (not (at ?loc1)) (at ?loc2))
  )

  (:action open_door
    :parameters (?loc1 - location ?loc2 - location)
    :precondition (and (at ?loc1) (closed_door ?loc1 ?loc2))
    :effect (not (closed_door ?loc1 ?loc2))
  )
)
```

Figure 5: Annotated domain file for Coin Collector.

C Hyperparameters

For both simulations, we use the implementation from Jansen and Côté (2022). For Coin Collector, we use the most complex setting supported by the system of 11 rooms with random connectivity, allowing up to 50 exploration steps. For Cooking World, we consider an easy setting with 2 rooms and 2 ingredients up to 20 steps and a hard setting of 5 rooms and 5 ingredients up to 50 steps. For both datasets, we vary the random random seed to generate randomize environment configurations, and use 0-9 as the development set, and 10-59 as the test set.

For the choice of LLM, we consider `gpt-3.5-turbo-1106` (GPT 3.5 Turbo) and `gpt-4-1106-preview` (GPT 4 Turbo) across baseline methods (i.e., Action-gen, PDDL-gen, and PDDL-edit). We set the temperature to 1 to

```
(define (domain environment)
  (:requirements :strips :typing :negative-preconditions :disjunctive-
       preconditions)

  (:types
    ingredient container knife toaster stove oven barbeque - object
    location
    direction
  )

  (:predicates
    (at ?loc - location)
    (obj_at ?obj - object ?loc - location)
    (visited ?loc - location)
    (connected ?loc1 - location ?loc2 - location ?dir - direction)
    (closed_door ?loc1 - location ?loc2 - location)

    (grilled ?ing - ingredient)
    (roasted ?ing - ingredient)
    (fried ?ing - ingredient)
    (chopped ?ing - ingredient)
    (sliced ?ing - ingredient)
    (diced ?ing - ingredient)
    (have ?obj - object)
  )

  (:action move
    :parameters (?loc1 - location ?loc2 - location ?dir - direction)
    :precondition (and (at ?loc1) (connected ?loc1 ?loc2 ?dir) (not (
         closed_door ?loc1 ?loc2)))
    :effect (and (not (at ?loc1)) (at ?loc2))
  )

  (:action open_door
    :parameters (?loc1 - location ?loc2 - location)
    :precondition (and (at ?loc1) (closed_door ?loc1 ?loc2))
    :effect (not (closed_door ?loc1 ?loc2))
  )

  (:action use_stove
    :parameters (?ing - ingredient ?loc - location ?sto - stove)
    :precondition (and (at ?loc) (obj_at ?sto ?loc) (have ?ing))
    :effect (fried ?ing)
  )

  (:action use_toaster
    :parameters (?ing - ingredient ?loc - location ?toa - toaster)
    :precondition (and (at ?loc) (obj_at ?toa ?loc) (have ?ing))
    :effect (grilled ?ing)
  )

  (:action use_oven
    :parameters (?ing - ingredient ?loc - location ?ove - oven)
    :precondition (and (at ?loc) (obj_at ?ove ?loc) (have ?ing))
    :effect (roasted ?ing)
  )

  (:action use_barbeque
    :parameters (?ing - ingredient ?loc - location ?bbq - barbeque)
    :precondition (and (at ?loc) (obj_at ?bbq ?loc) (have ?ing))
    :effect (grilled ?ing)
  )

  (:action chop
    :parameters (?ing - ingredient ?kni - knife)
    :precondition (and (have ?ing) (have ?kni))
    :effect (chopped ?ing)
  )

  (:action slice
    :parameters (?ing - ingredient ?kni - knife)
    :precondition (and (have ?ing) (have ?kni))
    :effect (sliced ?ing)
  )

  (:action dice
    :parameters (?ing - ingredient ?kni - knife)
    :precondition (and (have ?ing) (have ?kni))
    :effect (diced ?ing)
  )
)
```

Figure 6: Annotated domain file for Cooking World.

study stability. For Action-gen, we prompt the LLM with a full description of the simulation with the aim that a human player can learn to succeed. For the PDDL approaches, whenever the generated or edited problem cannot be solved by the domain file, indicating an error, the model is allowed to retry up to 5 times before failing the task. Models are provided with a hand-annotated domain file for each task contains permitted actions (e.g., move, open door, pick up) with well-defined types and predicates, parameters, pre-conditions, and effects. For the PDDL-edit setting, we prompt the LLM to generate edits (add, replace, and delete lines in the problem file) in JSON with the force-JSON flag set. The prompt of each method include a 1-shot demonstration of the output format (i.e., an action, a problem file, or edits to a problem file). See details of prompt design and domain file annotation in the Appendix B.

D Example Trajectories

A sample trajectory by GPT 4 Turbo with PDDL-edit on Cooking World (hard) is shown in Figure 7. The navigation portion of the interaction (look around, open door, move) is inherited from Coin Collector.

E Generated Problem Files

Figure 8 demonstrates a PDDL problem file learned throughout exploration in Coin Collector that indicates the existence and connectivity of all rooms the agent has access to before finding the coin.

F Results on the Development Set

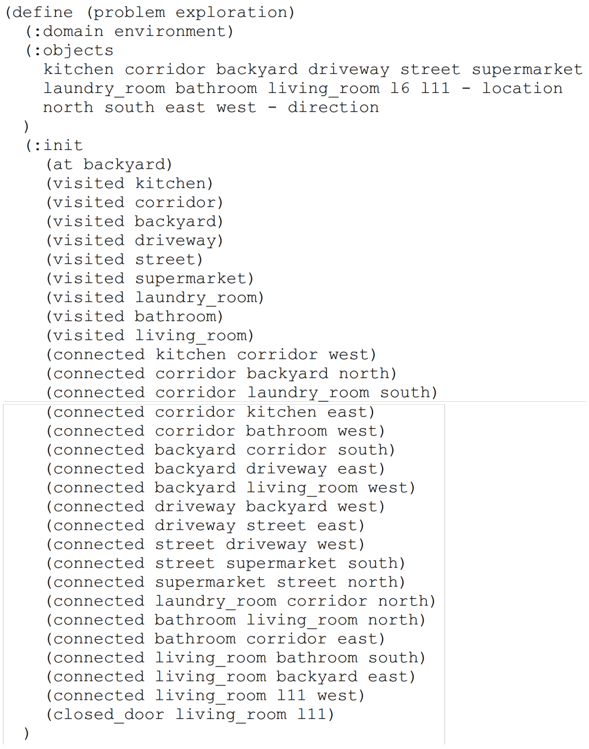
Table 2 is the counterpart of Table 1 showcasing the results on the development set.

```
< examine cookbook
> Gather all following ingredients and follow the directions to prepare this tasty meal.
Ingredients:
  black pepper, block of cheese, salt, red apple, yellow potato
Directions:
  slice the block of cheese, dice the red apple, chop the yellow potato, grill the yellow potato, prepare meal
< look around
> You are in the kitchen. In one part of the room you see a stove. There is also an oven. You also see a fridge that is closed. In another part of the
       room you see a counter that has a knife, a red apple, a raw yellow potato, and a cookbook on it. In one part of the room you see a kitchen
       cupboard that is closed. There is also a cutlery drawer that is closed. You also see a trash can that is closed. In another part of the room you
       see a dishwasher that is closed. In one part of the room you see a dining chair, that has nothing on it.
To the South you see a closed plain door. To the West you see the corridor.
< open fridge
> You open the fridge. The fridge contains a block of cheese.
< take block of cheese
> You take the block of cheese.
< take knife
> You take the knife.
< take red apple
> You take the red apple.
< take yellow potato
> You take the yellow potato.
< open kitchen cupboard
> You open the kitchen cupboard. It's empty inside.
< open cutlery drawer
> You open the cutlery drawer. It's empty inside.
< open trash can
> You open the trash can. It's empty inside.
< open dishwasher
> You open the dishwasher. It's empty inside.
< move west
> You are in the corridor. In one part of the room you see a key holder, that has nothing on it. There is also a shoe cabinet that is closed. You also
       see a umbrella stand, that has nothing on it. In another part of the room you see a hat rack, that has nothing on it. In one part of the room you
        see a coat hanger, that has nothing on it.
To the North you see a closed wood door. To the East you see the kitchen. To the West you see a closed sliding patio door.
< open shoe cabinet
> You open the shoe cabinet. It's empty inside.
< open door to north
> You open the wood door, revealing the bedroom.
< move north
> You are in the bedroom. In one part of the room you see a dressing table, that has nothing on it. There is also a desk chair, that has nothing on it.
        You also see a desk, that has nothing on it. In another part of the room you see a chest of drawers that is closed. In one part of the room you
       see a wardrobe that is closed. There is also a night stand, that has nothing on it. You also see a bed, that has nothing on it.
Through an open wood door, to the South you see the corridor.
< open chest of drawers
> You open the chest of drawers. It's empty inside.
< open wardrobe
> You open the wardrobe. It's empty inside.
< move south
> You are in the corridor. (description omitted)
< open door to west
> You open the sliding patio door, revealing the backyard.
< move west
> You are in the backyard. In one part of the room you see a barbeque. There is also a workbench, that has nothing on it. You also see a patio chair,
       that has nothing on it. In another part of the room you see a patio table, that has nothing on it. In one part of the room you see a clothes line
       , that has nothing on it. There is also a garden, that has nothing on it.
Through an open sliding patio door, to the East you see the corridor.
< move east
> You are in the corridor. (description omitted)
< move east
> You are in the kitchen. (description omitted)
< open door to south
> You open the plain door, revealing the pantry.
< move south
> You are in the pantry. In one part of the room you see a folding chair, that has nothing on it. There is also a shelf that has some black pepper, and
        some salt on it.
Through an open plain door, to the North you see the kitchen.
< take black pepper
> You take the black pepper.
< take salt
> You take the salt.
< dice red apple
> You dice the red apple.
< slice block of cheese
> You slice the block of cheese.
< chop yellow potato
> You chop the yellow potato.
< move north
> You are in the kitchen. (description omitted)
Through an open plain door, to the South you see the pantry. To the West you see the corridor.
< move west
> You are in the corridor. (description omitted)
< move west
> You are in the backyard. (description omitted)
< cook yellow potato in barbeque
> You grill the yellow potato with the barbeque.
< move east
> You are in the corridor. (description omitted)
< move east
> You are in the kitchen. (description omitted)
< prepare meal
> Adding the meal to your inventory.
< eat meal
> You eat the meal. It is delicious.
```

Figure 7: An example trajectory performed by GPT 4 Turbo and PDDL-edit on Cooking World (hard).

	random	GPT 3.5 Turbo			GPT 4 Turbo		
		Action-gen	PDDL-gen†	PDDL-edit†	Action-gen	PDDL-gen†	PDDL-edit†
Coin	20%	80%	30%	70%	**90%**	50%	80%
Cooking (easy)	0%	0%	100%	70%	10%	90%	**100%**
Cooking (hard)	0%	0%	0%	0%	0%	0%	**50%**

Table 2: The percentage where the agent succeeds by taking no more than the maximum steps on the development set. The † sign specifies methods under our proposed PDDLEGO methodology.



```
(define (problem exploration)
  (:domain environment)
  (:objects
    kitchen corridor backyard driveway street supermarket
    laundry_room bathroom living_room l6 l11 - location
    north south east west - direction
  )
  (:init
    (at backyard)
    (visited kitchen)
    (visited corridor)
    (visited backyard)
    (visited driveway)
    (visited street)
    (visited supermarket)
    (visited laundry_room)
    (visited bathroom)
    (visited living_room)
    (connected kitchen corridor west)
    (connected corridor backyard north)
    (connected corridor laundry_room south)
    (connected corridor kitchen east)
    (connected corridor bathroom west)
    (connected backyard corridor south)
    (connected backyard driveway east)
    (connected backyard living_room west)
    (connected driveway backyard west)
    (connected driveway street east)
    (connected street driveway west)
    (connected street supermarket south)
    (connected supermarket street north)
    (connected laundry_room corridor north)
    (connected bathroom living_room north)
    (connected bathroom corridor east)
    (connected living_room bathroom south)
    (connected living_room backyard east)
    (connected living_room l11 west)
    (closed_door living_room l11)
  )
```

Figure 8: An example PDDL problem file learned throughout exploration in Coin Collector.