LLM-Guided Probabilistic Program Induction for POMDP Model Estimation

Anonymous Author(s)
Affiliation
Address
email

Abstract

Partially Observable Markov Decision Processes (POMDPs) model decision making under uncertainty. While there are many approaches to approximately solving POMDPs, we aim to address the problem of learning such models. In particular, we are interested in a subclass of POMDPs wherein the components of the model, including the observation function, reward function, transition function, and initial state distribution function, can be modeled as low-complexity probabilistic graphical models in the form of a short probabilistic program. Our strategy to learn these programs uses an LLM as a prior, generating candidate probabilistic programs that are then tested against the empirical distribution and adjusted through feedback. We experiment on a number of classical toy POMDP problems, simulated MiniGrid domains, and two real mobile-base robotics search domains involving partial observability. Our results show that using an LLM to guide in the construction of a low-complexity POMDP model can be more effective than tabular POMDP learning, behavior cloning, or direct LLM planning.

1 Introduction

Decision making under uncertainty is a central challenge in robotics, autonomous systems, and artificial intelligence more broadly. Partially Observable Markov Decision Processes (POMDPs) provide a principled framework for modeling and solving such problems by explicitly representing uncertainty in state perception, transitions, and rewards. Many prior works have used the POMDP formulation to solve real-world problems such as intention-aware decision-making for autonomous vehicles Song et al. (2016), collaborative control of smart assistive wheelchairs Ghorbel et al. (2018), robotic manipulation in cluttered environments Pajarinen and Kyrki (2017), and generalized object search Zheng et al. (2023). Despite their conceptual clarity, practical application of POMDPs is bottlenecked by difficulties in specifying accurate models of environments, which requires careful engineering and a thorough understanding of the theory and available solvers.

In this work, we address the critical challenge of learning interpretable, low-complexity POMDP models directly from data. We specifically target a class of POMDPs whose components, namely the observation function, reward function, transition dynamics, and initial state distribution, can be succinctly represented as probabilistic graphical models encoded by short probabilistic programs. To efficiently identify these programs, we leverage recent advancements in Large Language Models (LLMs) to serve as informative priors, generating candidate probabilistic programs. These candidates are evaluated against empirical observations and iteratively refined through LLM-generated feedback.

We evaluate our approach across several domains including classical POMDP problems, minigrid navigation and manipulation problems, and a real-world robotics setting involving a mobile-base robot searching for a target object. Our experimental results demonstrate that guiding model construction

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

with an LLM significantly enhances sample efficiency compared to traditional tabular model or behavior learning methods and achieves greater accuracy than directly querying an LLM.

2 Related Work

Learning world models for partially observable decision-making is a form of model-based reinforcement learning Moerland et al. (2020), for which there are many methods. In this section, we focus on methods that emphasize extreme data efficiency through the use of explicit models and representations, such as probabilistic programs, that facilitate efficient learning. Additionally, we discuss methods that use LLMs to synthesize and refine these models, enabling the integration of human priors and domain-specific constraints to create world models for downstream solvers.

POMDP Model Learning A substantial body of research has addressed the challenge of learning Partially Observable Markov Decision Process (POMDP) models from experience. For example, Mossel and Roch Mossel and Roch (2005) provide an average-case complexity result showing that estimating the parameters of certain Hidden Markov Models, an essential subproblem in POMDP learning, is computationally intractable in general. Despite these challenges, several approaches have made significant progress by introducing tractability under specific assumptions.

Bayesian methods, such as Bayes-Adaptive POMDPs Ross et al. (2007), incorporate model uncertainty directly into decision-making by unifying model learning, information gathering, and exploitation. This, however, increases the overall complexity of the POMDP. In contrast, spectral techniques like Predictive State Representations (PSRs) Boots et al. (2009) offer computational efficiency by bypassing full Bayesian inference, although they require strong structural assumptions and extensive exploratory data.

Recent work on optimism-based exploration algorithms has yielded theoretical guarantees for efficient learning in specific POMDP subclasses, even though these methods can be challenging to apply directly to real-world, complex domains Jin et al. (2020). Additionally, apprenticeship learning approaches estimate POMDP parameters by leveraging expert demonstrations, assuming that expert behavior encapsulates informative state-transition dynamics Makino and Takeuchi (2012). Such methods help reduce the burden of exploration but are sensitive to the quality of the expert demonstrations. In dialogue systems, for instance, these techniques have successfully learned user models without relying on manual annotations Thomson et al. (2010).

Probabilistic Program Induction. Probabilistic programming provides a powerful framework for modeling complex systems using concise, symbolic representations Bingham et al. (2018); Cusumano-Towner et al. (2019); Goodman et al. (2012). Several works in probabilistic program induction have demonstrated that such representations can lead to markedly improved data efficiency and enable few-shot learning, in stark contrast to more data-intensive conventional methods Ellis et al. (2020); Lake et al. (2015). Nonetheless, the vast search space inherent to probabilistic programs can be a major computational bottleneck. Recent advances have attempted to address this issue by integrating language models with probabilistic programming, thereby infusing human priors into the model discovery process Li et al. (2024); Wong et al. (2023); Grand et al. (2024).

LLM Model Learning for Decision-Making. The application of large language models (LLMs) to world-modeling for decision-making is an emerging and rapidly evolving area. Prior work has primarily focused on fully observable settings, where transition dynamics and reward structures are represented using frameworks such as Planning Domain Definition Language (PDDL) or code-based models Liang et al. (2025); Tang et al. (2024). Other approaches have utilized code-based representations as constraints or as part of optimization frameworks Curtis et al. (2024); Hao et al. (2024); Ye et al. (2024). To our knowledge, our work is the first to extend these techniques to the POMDP setting, thereby addressing the additional complexities introduced by partial observability.

3 Background

3.1 Partially Observable Markov Decision Processes

Partially Observable Markov Decision Processes (POMDPs) provide a principled framework for sequential decision making under uncertainty. A POMDP is defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{Z}, \mathcal{R}, \gamma)$, where $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{O}$ denote the state, action, and observation spaces, respectively. In this work we

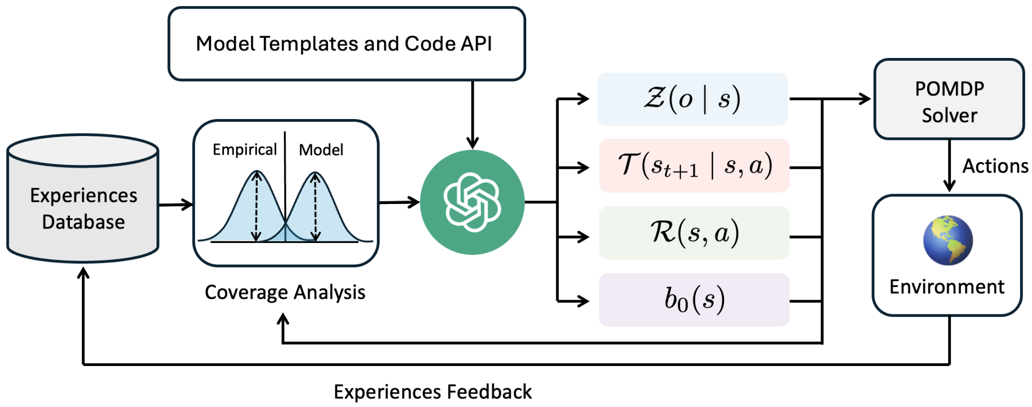

Figure 1: An architecture diagram for our POMDP coder method

assume all these spaces are discrete, but the POMDP formulation supports continuous spaces in general. $\mathcal{T}(s_{t+1} \mid s_t, a_t)$ indicates the distribution over next states expected when taking action a_t in state s_t. The observation model $\mathcal{Z}(o_{t+1} \mid s_{t+1}, a_t)$ represents the distribution over observations expected when executing action a_t resulting in a subsequent state s_{t+1}. $\mathcal{R}(s_t, a_t, s_{t+1})$ is the reward function. Lastly, $\gamma \in (0, 1)$ is the discount which weighs the value of current over future rewards.

Since the agent does not have direct access to the true state s_t, it maintains a belief $b_t(s)$, or a probability distribution over possible states, which must be updated for replanning after each action is taken and new observation received. Given an action and observation, a belief can be updated via *particle filtering* Thrun et al. (2005). We use particle filtering as our belief-updating mechanism across all domains.

The objective of a POMDP is to find a policy π that maximizes the expected discounted reward:

$$\max_{\pi} \mathbb{E}_{b_0,\pi} \left[\sum_{t=0}^{\infty} \gamma^t \sum_{s \in \mathcal{S}} b_t(s) \sum_{s_{t+1} \in \mathcal{S}} \mathcal{T}(s_{t+1} \mid s, a_t)\, \mathcal{R}(s, a_t, s_{t+1}) \right] \tag{1}$$

To illustrate the POMDP formulation and serve as a running example, consider the classic Tiger problem Kaelbling et al. (1998). An agent faces two doors: one hides a tiger, the other a treasure. The true state s consisting of the the tiger's location is hidden. The agent can `listen` to receive a noisy observation o of which door the tiger is behind, or `open` a door to gain a reward or incur a penalty. An ideal policy is to wait long enough to be confident in the tiger's location before opening the door with the treasure.

3.2 Probabilistic Programs

Probabilistic programming offers an expressive and concise way to represent complex probabilistic models as executable code. In our work, each component of the POMDP including the initial state model, transition dynamics, observation function, and reward structure is encoded as a short probabilistic program. We leverage *Pyro* Bingham et al. (2018), a flexible probabilistic programming framework built on Python, to specify these models. Pyro enables us to define generative models with inherent stochastic behavior. In our tiger problem example, the below implementation would be a correct probabilistic program for the observation model.

```
def tiger_observation_func(state: TigerState, act: TigerActions):
    if act != TigerActions.LISTEN:
        return TigerObservation(NONE)

    correct = bool(pyro.sample("listen_correct", Bernoulli(torch.tensor(0.85))))
```

```
    tiger_left = state.tiger_location == 0
    hear_left  = (correct and tiger_left) or (not correct and not tiger_left)
    return TigerObservation(HEAR_LEFT if hear_left else HEAR_RIGHT)
```

3.3 POMDP Solvers

While traditional offline solvers aim to compute a global policy over the entire belief space Mundhenk et al. (1997); Littman (1995); Cassandra et al. (2013), these methods often face scalability challenges in high-dimensional or continuous environments. In contrast, online solvers focus on finding a solution from a specific initial belief and replan after every step as new observations become available. Online approaches such as Partially Observable Monte Carlo Planning (POMCP) Silver and Veness (2010); Curtis et al. (2022) and Partially Observable Upper Confidence Trees (POUCT) Sunberg and Kochenderfer (2017) combine Monte Carlo sampling with tree search to approximate optimal policies in real time. Additionally, determinized belief space planners simplify the stochastic nature of the problem by converting it into a deterministic surrogate, thereby enabling rapid replanning Yoon et al. (2007); Kaelbling and Lozano-Perez (2013); Chatterjee et al. (2021); Curtis et al. (2024). In our work, we adopt the determinized belief space planning approach due to its computational efficiency in larger problems. Please refer to Appendix B for specifics on our solver implementation.

4 POMDP Coder

In our approach, which we call POMDP Coder, we decompose the problem into two major components: learning the probabilistic models that define the POMDP and using these learned models for online planning. The first part involves leveraging a Large Language Model (LLM) to generate, refine, and validate candidate probabilistic programs that represent the initial state, transition, observation, and reward functions. The second component uses these models within an online POMDP solver along with the current belief to find optimal actions to take in the environment.

We assume the agent is provided access to an initial set of ten human-generated demonstrations $\mathcal{D}$, where each demonstration consists of $(s_t, a_t, o_{t+1}, s_{t+1})$ transitions. Since we are learning models instead of policies, there are no strict assumptions made about the optimality or correctness of these demonstrations. However, we do make an assumption of post-hoc full observability Pinto et al. (2017). That is, we assume the agent gets access to the intermediate states after an episode has terminated (see Section 7 for details).

Additionally POMDP Coder is provided a code-based API defining the structure of the state, action, and observation space. Below is an example for the Tiger domain.

```
class TigerActions(enum.IntEnum):
    OPEN_LEFT = 0, OPEN_RIGHT = 1, LISTEN = 2

class TigerObservation(Observation):
    obs: int # 0 = hear left, 1 = hear right, 2 = none

class TigerState(State):
    tiger_location: int  # 0 = left, 1 = right
```

In some cases, we additionally expose python libraries and deterministic helper functions alongside the API to help with more complex calculations (see Appendix D for details).

Given these inputs, POMDP Coder proceeds as outlined in Algorithm 1. It proposes an initial set of models that comprise the POMDP problem (Line 3), using a learning procedure detailed in Section 4.1. After an initial set of models is decided on, these models are passed to a POMDP solver along with the initial belief (Line 6) to find an optimal first action to take in the environment (Line 7). After an action is taken and an observation received, the belief is updated using particle filtering to

Algorithm 1 POMDP Coder

1: **Input:** A demo dataset $\mathcal{D}$, max episodes E, num particles N, empty models $\theta = \emptyset$
2: **for** episode $= 1$ **to** E **do**
3: $\quad \theta = (\theta_{\text{trans}}, \theta_{\text{rew}}, \theta_{\text{obs}}, \theta_{\text{init}}) \leftarrow$ `LearnModels`$(\mathcal{D}, \theta)$ ▷ Update all models using $\mathcal{D}$
4: $\quad b \leftarrow N$ samples from θ_{init}
5: $\quad$ **while** episode not terminated **do**
6: $\quad\quad a \leftarrow$ `POMDPSolver`(b, θ) ▷ Plan best next action, see Appendix B
7: $\quad\quad$ Execute a in the world, observe o
8: $\quad\quad b \leftarrow$ `ParticleFilter`(b, a, o, θ) ▷ Update belief
9: $\quad\quad$ Append (a, o) to trajectory τ
10: $\quad$ **end while**
11: $\quad \mathcal{D} \leftarrow \mathcal{D} \cup \tau$ ▷ Update demo data with new trajectory
12: **end for**
13: **return** θ

form a new belief (Line 8). This process continues until the episode terminates or times out. Lastly, at the end of each episode, the trajectory is added to the dataset (Line 8) and the learning process repairs any inaccuracies that the previous model may have had under the new data (Line 3).

4.1 Learning Models

We aim to learn four core components of a POMDP: the initial state distribution $P(s_0)$, the transition model $P(s_{t+1} \mid s_t, a_t)$, the observation model $P(o_{t+1} \mid s_{t+1}, a_t)$, and the reward model $R(s_t, a_t, s_{t+1})$. Each of these components is expressed as a short probabilistic program.

The objective of our learning procedure is to maximize a dataset *coverage* metric, which we define to be the proportion of data in $\mathcal{D}$ that has support under the model as follows:

$$\text{coverage}(P_\theta, \mathcal{D}) \;=\; \frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} \mathbf{1}\Big[\, P_\theta\big(y_i \mid x_i\big) > 0 \,\Big]. \tag{2}$$

Although we experimented with other metrics such distributional distance metrics, we found that those were more prone to overfitting and less interpretable to an LLM than binary coverage feedback. Still, the coverage metric has its own limitations, which we discuss in Section 7.

Our approach to learning these models builds on strategies previously developed for reward model learning Tang et al. (2024), but extends them to the more general setting of POMDP model learning across multiple stochastic components (transition, observation, initial state, and reward), replaces accuracy with coverage, and introduces the notion of a testing and training split to avoid overfitting.

At its core, our model learning strategy uses two operations: (1) *LLM program proposal* given a model function template and a set of examples from the database and and (2) *LLM program repair* given a previous model and set of examples that the previous model failed to cover. We run a stochastic procedure for sampling which program to repair next, which is biased toward repairing programs that have high coverage. The pseudocode and additional details can be found in the Appendix A.

5 Experiments

5.1 Simulated Experiments

Simulated experiments were conducted on two categories of problems: classical POMDP problems from the literature and MiniGrid tasks. The classical POMDP problems such as Tiger and Rock Sample Smith and Simmons (2012) serve as simplified benchmarks that capture the core challenges of decision making under uncertainty while keeping the problem domains small and tractable.

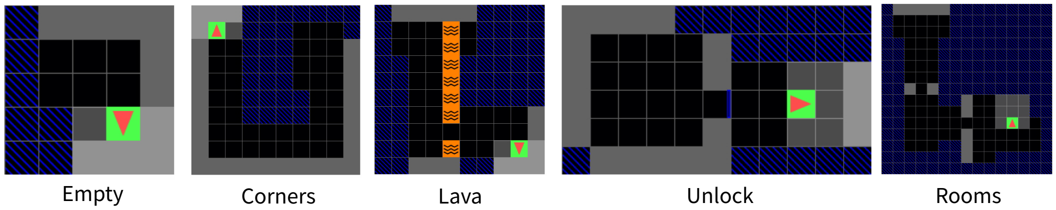

Figure 2: A visualization of the final belief state for each of the MiniGrid tasks. The green square is the goal, the red triangle is the agent, and the blue squares are places that the agent has not viewed.

MiniGrid is a set of minimalistic gridworld tasks for testing navigation and planning under partial observability originally designed for reinforcement learning Chevalier-Boisvert et al. (2023). We evaluate on five of these environments shown in Figure 2. Detailed descriptions for each of these environments can be found in Appendix C. Each MiniGrid environment is modified from the original implementation Chevalier-Boisvert et al. (2023). This modification demonstrates the ability of our method to generalize to new environments not seen during LLM pretraining.

5.2 Baselines

In our experiments, we evaluate our method against several diverse baselines to comprehensively assess its performance in partially observable environments. One baseline, termed the *oracle*, uses POMDP models that are hardcoded to exactly match the true dynamics of the environment, thereby serving as an upper-bound on achievable performance. In contrast, the *random* baseline takes actions arbitrarily at every step, establishing a lower-bound benchmark for comparison.

Another baseline, referred to as *direct LLM*, involves querying a large language model for the next action at each decision point. In this setup, the LLM is provided with all the same information provided to POMDP Coder during model learning. The exact prompt template used for this method is detailed in Appendix F.4. Next, our evaluation includes a *behavior cloning* baseline, where a policy is constructed by mapping states to actions using a dictionary learned from the demonstration dataset. In addition, we consider a *tabular baseline* in which the POMDP models are learned as conditional probability tables derived from counts in the demonstration dataset.

Lastly, we test against two ablations of POMDP Coder. The first is the *offline only* ablation which only makes use of the human demonstrations and does not update the model with its own experiences. Conversely, the *online only* ablation does not make use of the expert demonstrations, learning only from its own experiences. All other baselines are given access to both offline and online data.

5.3 Simulation Results

We evaluate various methods using expected discounted reward defined in Section 3.1, measuring both total cumulative reward and efficiency. We use GPT-4o OpenAI (2024) with temperature 0 as the large language model across all experiments. The same ten demonstrations are provided to all methods. The results of our evaluations can be seen in Figure 3. We see POMDP Coder match or outperform all baseline methods across all domains.

We observe that the behavior cloning and tabular baselines were fundamentally limited in their ability to generalize. This is because the set of possible initial states for many tasks was orders of magnitude larger than the training set. In contrast, the probabilistic programs written by POMDP Coder use symbolic abstraction to cover large portions of the state and observation spaces, allowing them to generalize to new situations. While the direct LLM approach was sometimes effective, such as in the rock sample domain, it frequently got stuck in infinite loops, failing to understand constraints such as obstacle obstruction despite the examples of collision it had access to in the dataset.

POMDP Coder outperformed both the online-only and offline-only ablations across most environments. A common failure mode of the offline-only ablation was missing transitions outside expert demonstrations. For example, it will run into lava without knowing it causes death. In contrast, the online-only ablation struggled to discover informative actions due to inefficient random exploration.

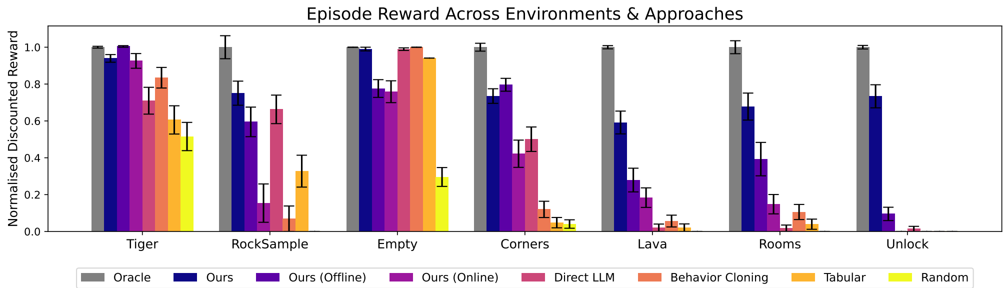

Figure 3: Experimental results for the MiniGrid and Classical POMDP domains. We show the expected discounted returns ($\gamma = 0.98$) of each method across five learning seeds with ten episodes per seed. The error bars show standard error across all episodes. We normalize the expected discounted returns by the performance of Oracle.

For instance, in the unlock environment, if the agent never used the key, the model failed to learn that behavior, and random actions rarely uncovered it.

In addition to success metrics, we record some additional runtime statistics such as the number of candidate programs generated during offline and online learning in Table 2 as well as the training and testing coverage scores after offline model learning in Table 3.

5.4 Real Robot Experiments

Our real robot experiments are conducted using a Boston Dynamics Spot robot. The robot carries an in-hand camera mounted on a 6-DoF arm at its back. April tags are distributed throughout the area, enabling precise localization. The goal is for the robot to find an pick up an apple placed within the scene. Before any demonstrations are gathered, we construct a map of the empty room by scanning it with PolyCam Polycam (2025). This scan is used to construct a scene representation that includes an object-centric scene graph, encoding "on" relationships derived from geometric cues, and an occupancy grid delineating forbidden zones corresponding to physical obstacles. For each real-world task, ten demonstrations are collected by commanding the robot via keyboard. The agent's action space is discretized into fixed theta rotations to the left and right and movements in the four cardinal directions. Objects are detected online using Grounding SAM, an open vocabulary object detector Liu et al. (2023); Ren et al. (2024).

We test our method in two distinct spaces. The first is a small, closed-off room, as shown in the top row of Figure 4, which contains a few tables, chairs, and drawer cabinets. The second is a large, open lobby area depicted in the bottom row of Figure 4, furnished with more than twenty pieces of furniture. Within these environments, task distributions are defined by varying the location of the apple. In the Small-Cabinets configuration, which takes place in the small room, the apple is consistently placed on top of one of the three drawer cabinets for each demonstration. In the Large-Tables setting within the large room, the apple is positioned on one of the five round tables.

Our approach is compared against a subset of baselines evaluated in Section 5.2. The evaluation includes behavior cloning, direct LLM execution, and a hardcoded uniform baseline that assumes the object is placed uniformly throughout the search space. Although the uniform baseline requires additional task-specific human input and is not strictly an apples-to-apples comparison, it demonstrates that our method can outperform a naively designed initial state distribution.

The results shown in Table 1 demonstrate that POMDP Coder achieves more efficient and accurate exploration by understanding and generalizing trends in initial state distribution seen in the training data. Specifically, our approach learns that objects are always on top of objects of a particular class, and constructs an initial state distribution that captures that without overfitting to specific initial states.

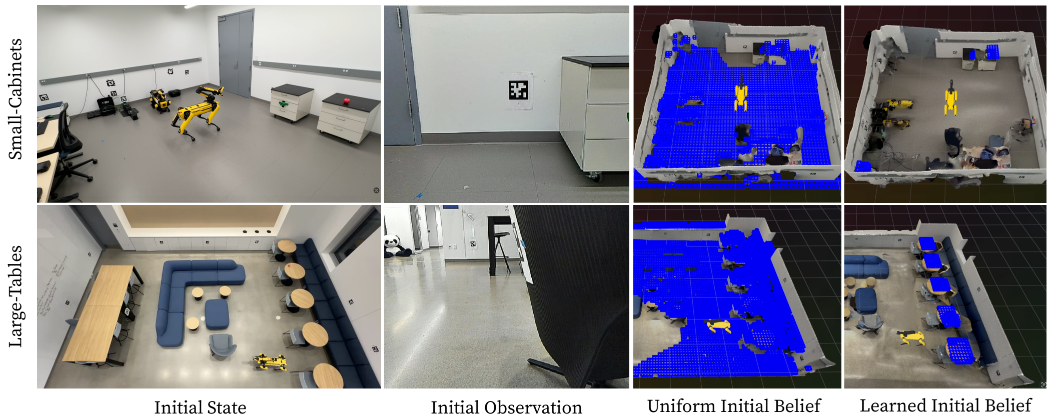

Figure 4: The two real-world experimental setups wherein a robot is searching for an apple in a partially observable world. The blue cells represent the robot's belief about where the apple could be in the world. In the uniform initial belief, the robot thinks the apple could be anywhere it has not looked yet. The learned initial belief found by POMDP Coder has a narrower initial belief leading to more efficient exploration.

	Ours	Uniform	Direct LLM	BC	Tabular
Small-Cabinets	$\mathbf{0.89} \pm \mathbf{0.09}$ (10)	0.68 ± 0.21 (10)	0.25 ± 0.42 (3)	0.28 ± 0.39 (4)	0.35 ± 0.41 (5)
Large-Tables	$\mathbf{0.73} \pm \mathbf{0.08}$ (10)	0.40 ± 0.26 (8)	0.15 ± 0.32 (2)	0.13 ± 0.28 (2)	0.13 ± 0.22 (3)

Table 1: Real-world experiment results on the small room domain in the top row of Figure 4 and the large room domain in the bottom row of Figure 4. The table shows the mean and standard deviation of expected discounted reward (with $\gamma = 0.98$) under the ground-truth reward model (1 if the agent is holding the apple and 0 otherwise) along with the number of successes over ten runs in parenthesis.

6 Discussion

In this work, we have presented a novel approach to learning interpretable, low-complexity POMDP models by integrating LLM-guided probabilistic program induction with online planning. Our method leverages large language models to generate and iteratively refine candidate probabilistic programs that capture the dynamics, observation functions, initial state distributions, and reward structures of complex environments. Experimental results on simulated MiniGrid domains and real-world robotics scenarios demonstrate that our approach can significantly enhance sample efficiency and predictive accuracy compared to traditional tabular learning methods, behavior cloning, or direct LLM planning.

Our findings further suggest that environments represented with structured scene graphs and other rich input representations can be better modeled by learning a world model within which a reasoning agent can operate, rather than by directly learning a policy that maps observation histories to actions or by attempting to apply a language model in a zero-shot setting. This is particularly evident in large, partially observable worlds where the belief space is considerably more complex and challenging to cover with training examples than the space of states itself. The use of code to represent these models is especially advantageous, as language models are adept at generating concise, executable snippets that can be interpreted, debugged, and evaluated post-hoc, thus providing an additional layer of transparency and robustness in model evaluation.

7 Limitations

Despite these promising results, several limitations remain. Our approach currently relies on human expertise to design the underlying representation over which the world model is learned, which may constrain its applicability to domains where such structured representations are not readily available.

Additionally, due to our post-hoc observability assumption, collecting datasets outside of a simulator requires one of the following: human state annotation, complete robot exploration after the episode, or third-party perspectives such as externally mounted cameras. In our real robot experiments, this was not a challenge because the only state variability was in the position of the goal object, which is fully determined upon completion of the task. More complex problems with multiple dimensions of both task-relevant and task-irrelevant uncertainty would require more than just the agent's perspective. Alleviating this assumption may require jointly reasoning about the interrelated structure of the constituent models, and is a valuable direction for future work.

Moreover, the particle filter employed in our current implementation does not scale well to arbitrarily large state spaces. Future work may address this limitation by incorporating more advanced inference techniques, such as factored particle filters or other scalable methods, to improve performance in high-dimensional settings.

Another area for improvement is in the sometimes overly broad distributions proposed by the LLM due to the coverage metric indirectly rewarding broader distributions. While this doesn't make the problem infeasible, it can lead to less efficient behavior. A direction for future work could be to use inference methods on the hidden variables of the proposed probabilistic program to strike a balance between the empirical distribution and the overly broad model distribution.

Lastly, our study focuses exclusively on discrete state and action spaces, despite robotics tasks requiring search over continuous spaces such as grasps and poses. Extending our learning strategy and adopting continuous-space POMDP solvers would broaden our framework to these domains, enabling more complex manipulation and navigation tasks.

Ultimately, our work opens up exciting avenues for combining the strengths of probabilistic programming and large language models to construct robust, interpretable models for decision-making under uncertainty.

References

W. Song, G. Xiong, and H. Chen. Intention-aware autonomous driving decision-making in an uncontrolled intersection. *Mathematical Problems in Engineering*, 2016:1–15, 01 2016. doi: 10.1155/2016/1025349.

M. Ghorbel, J. Pineau, R. Gourdeau, S. Javdani, and S. Srinivasa. A decision-theoretic approach for the collaborative control of a smart wheelchair. *International Journal of Social Robotics*, 10, 01 2018. doi: 10.1007/s12369-017-0434-7.

J. Pajarinen and V. Kyrki. Robotic manipulation of multiple objects as a pomdp. *Artificial Intelligence*, 247:213–228, 2017. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2015.04.001. URL `https://www.sciencedirect.com/science/article/pii/S0004370215000570`. Special Issue on AI and Robotics.

K. Zheng, A. Paul, and S. Tellex. A system for generalized 3d multi-object search. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1638–1644, 2023. doi: 10.1109/ICRA48891.2023.10161387.

T. M. Moerland, J. Broekens, and C. M. Jonker. Model-based reinforcement learning: A survey. *CoRR*, abs/2006.16712, 2020. URL `https://arxiv.org/abs/2006.16712`.

E. Mossel and S. Roch. Learning nonsingular phylogenies and hidden markov models. *CoRR*, abs/cs/0502076, 2005. URL `http://arxiv.org/abs/cs/0502076`.

S. Ross, B. Chaib-draa, and J. Pineau. Bayes-adaptive pomdps. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007. URL `https://proceedings.neurips.cc/paper_files/paper/2007/file/3b3dbaf68507998acd6a5a5254ab2d76-Paper.pdf`.

B. Boots, S. M. Siddiqi, and G. J. Gordon. Closing the learning-planning loop with predictive state representations, 2009. URL `https://arxiv.org/abs/0912.2385`.

C. Jin, S. M. Kakade, A. Krishnamurthy, and Q. Liu. Sample-efficient reinforcement learning of undercomplete pomdps. *CoRR*, abs/2006.12484, 2020. URL `https://arxiv.org/abs/2006.12484`.

T. Makino and J. Takeuchi. Apprenticeship learning for model parameters of partially observable environments, 2012. URL `https://arxiv.org/abs/1206.6484`.

B. Thomson, F. Jurčíček, M. Gašić, S. Keizer, F. Mairesse, K. Yu, and S. Young. Parameter learning for pomdp spoken dialogue models. In *2010 IEEE Spoken Language Technology Workshop*, pages 271–276, 2010. doi: 10.1109/SLT.2010.5700863.

E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. A. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep universal probabilistic programming. *CoRR*, abs/1810.09538, 2018. URL `http://arxiv.org/abs/1810.09538`.

M. F. Cusumano-Towner, F. A. Saad, A. K. Lew, and V. K. Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 221–236, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314642. URL `https://doi.org/10.1145/3314221.3314642`.

N. D. Goodman, V. Mansinghka, D. M. Roy, K. A. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. *CoRR*, abs/1206.3255, 2012. URL `http://arxiv.org/abs/1206.3255`.

K. Ellis, C. Wong, M. I. Nye, M. Sablé-Meyer, L. Cary, L. Morales, L. B. Hewitt, A. Solar-Lezama, and J. B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *CoRR*, abs/2006.08381, 2020. URL `https://arxiv.org/abs/2006.08381`.

B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015. doi: 10.1126/science.aab3050. URL https://doi.org/10.1126/science.aab3050.

M. Y. Li, E. B. Fox, and N. D. Goodman. Automated statistical model discovery with language models, 2024. URL https://arxiv.org/abs/2402.17879.

L. Wong, G. Grand, A. K. Lew, N. D. Goodman, V. K. Mansinghka, J. Andreas, and J. B. Tenenbaum. From word models to world models: Translating from natural language to the probabilistic language of thought, 2023. URL https://arxiv.org/abs/2306.12672.

G. Grand, V. Pepe, J. Andreas, and J. B. Tenenbaum. Loose lips sink ships: Asking questions in battleship with language-informed program sampling, 2024. URL https://arxiv.org/abs/2402.19471.

Y. Liang, N. Kumar, H. Tang, A. Weller, J. B. Tenenbaum, T. Silver, J. F. Henriques, and K. Ellis. Visualpredicator: Learning abstract world models with neuro-symbolic predicates for robot planning, 2025. URL https://arxiv.org/abs/2410.23156.

H. Tang, D. Key, and K. Ellis. Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment, 2024. URL https://arxiv.org/abs/2402.12275.

A. Curtis, N. Kumar, J. Cao, T. Lozano-Pérez, and L. P. Kaelbling. Trust the proc3s: Solving long-horizon robotics problems with llms and constraint satisfaction, 2024. URL https://arxiv.org/abs/2406.05572.

Y. Hao, Y. Chen, Y. Zhang, and C. Fan. Large language models can plan your travels rigorously with formal verification tools. In *arxiv preprint*, 2024. URL https://arxiv.org/abs/2404.11891.

X. Ye, Q. Chen, I. Dillig, and G. Durrett. Satlm: Satisfiability-aided language models using declarative prompting. *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/pdf/2305.09656.

S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. ISBN 0262201623.

L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1):99–134, 1998. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(98)00023-X. URL https://www.sciencedirect.com/science/article/pii/S000437029800023X.

M. Mundhenk, J. Goldsmith, and E. Allender. The complexity of policy evaluation for finite-horizon partially-observable markov decision processes. In I. Prívara and P. Ružička, editors, *Mathematical Foundations of Computer Science 1997*, pages 129–138, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69547-9.

M. Littman. The witness algorithm: Solving partially observable markov decision processes. 02 1995.

A. R. Cassandra, M. L. Littman, and N. L. Zhang. Incremental pruning: A simple, fast, exact method for partially observable markov decision processes. *CoRR*, abs/1302.1525, 2013. URL http://arxiv.org/abs/1302.1525.

D. Silver and J. Veness. Monte-Carlo planning in large POMDPs. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

A. Curtis, L. Kaelbling, and S. Jain. Task-directed exploration in continuous pomdps for robotic manipulation of articulated objects, 2022. URL https://arxiv.org/abs/2212.04554.

Z. Sunberg and M. J. Kochenderfer. POMCPOW: an online algorithm for pomdps with continuous state, action, and observation spaces. *CoRR*, abs/1709.06196, 2017. URL http://arxiv.org/abs/1709.06196.

S. W. Yoon, A. Fern, and R. Givan. Ff-replan: A baseline for probabilistic planning. In *International Conference on Automated Planning and Scheduling*, 2007. URL https://api.semanticscholar.org/CorpusID:15013602.

L. Kaelbling and T. Lozano-Perez. Integrated task and motion planning in belief space. *The International Journal of Robotics Research*, 32:1194–1227, 08 2013. doi: 10.1177/0278364913484072.

I. Chatterjee, T. Kusnur, and M. Likhachev. Fast bounded suboptimal probabilistic planning with clear preferences on missing information. *Proceedings of the International Symposium on Combinatorial Search*, 12(1):37–45, Jul. 2021. doi: 10.1609/socs.v12i1.18549. URL https://ojs.aaai.org/index.php/SOCS/article/view/18549.

A. Curtis, G. Matheos, N. Gothoskar, V. Mansinghka, J. Tenenbaum, T. Lozano-Pérez, and L. P. Kaelbling. Partially observable task and motion planning with uncertainty and risk awareness, 2024. URL https://arxiv.org/abs/2403.10454.

L. Pinto, M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel. Asymmetric actor critic for image-based robot learning. *CoRR*, abs/1710.06542, 2017. URL http://arxiv.org/abs/1710.06542.

H. Tang, K. Hu, J. P. Zhou, S. Zhong, W.-L. Zheng, X. Si, and K. Ellis. Code repair with llms gives an exploration-exploitation tradeoff. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 117954–117996. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/d5c56ec4f69c9a473089b16000d3f8cd-Paper-Conference.pdf.

T. Smith and R. G. Simmons. Heuristic search value iteration for pomdps. *CoRR*, abs/1207.4166, 2012. URL http://arxiv.org/abs/1207.4166.

M. Chevalier-Boisvert, B. Dai, M. Towers, R. de Lazcano, L. Willems, S. Lahlou, S. Pal, P. S. Castro, and J. Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *CoRR*, abs/2306.13831, 2023.

OpenAI. Gpt-4o system card, 2024. URL https://arxiv.org/abs/2410.21276. Accessed: 2025-05-08.

Polycam. Polycam – capture and share 3d models, 2025. URL https://poly.cam/. Accessed: 2025-04-30.

S. Liu, Z. Zeng, T. Ren, F. Li, H. Zhang, J. Yang, C. Li, J. Yang, H. Su, J. Zhu, et al. Grounding dino: Marrying dino with grounded pre-training for open-set object detection. *arXiv preprint arXiv:2303.05499*, 2023.

T. Ren, S. Liu, A. Zeng, J. Lin, K. Li, H. Cao, J. Chen, X. Huang, Y. Chen, F. Yan, Z. Zeng, H. Zhang, F. Li, J. Yang, H. Li, Q. Jiang, and L. Zhang. Grounded sam: Assembling open-world models for diverse visual tasks, 2024.

A Model Learning

The learning algorithm follows the pseudocode in Algorithm 2. Firstly, in order to mitigate overfitting to spurious patterns, we split the demonstration dataset into separate training and test sets (Line 3). Given the training set and a code-based interface that specifies how states, actions, and observations are represented, we query an LLM to propose an initial code snippet for a given component (Line 4, see Appendix F.2 for prompt).

Algorithm 2 `LearnModel`

1: **Input:** demonstration dataset $\mathcal{D}$, initial model θ_{prev}, budget N, smoothing constant C
2: **Output:** Learned model θ_{new}
3: Split $\mathcal{D}$ into $\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}}$
4: **if** $\theta = \emptyset$ **then** $\theta_{\text{prev}} \leftarrow$ LLM Init using $\mathcal{D}_{\text{train}}$ ▷ Appendix F.2
5: coverage, $\mathcal{D}_{\text{errors}} \leftarrow$ `Eval`$(\theta_{\text{prev}}, \mathcal{D}_{\text{test}}, \mathcal{D}_{\text{train}})$ ▷ Discrepancy between model & empirical
6: beta $\leftarrow$ Beta$(1 + C \cdot \text{coverage}, 1 + C \cdot (1 - \text{coverage}))$
7: $\mathcal{M} \leftarrow \{(\theta_{\text{prev}}, \text{beta}, \mathcal{D}_{\text{errors}})\}$
8: **while** coverage < 1.0 **and** iterations $< M$ **do**
9: $\quad(\theta_{\text{new}}, \text{Beta}(\alpha, \beta), \mathcal{D}_{\text{errors}}) \leftarrow \text{argmax}_{\mathcal{M}}(p \sim \text{beta})$ ▷ Thompson sampling
10: $\quad\theta'_{\text{new}} \leftarrow$ LLM refinement using θ_{new} and $\mathcal{D}_{\text{errors}}$ ▷ Appendix F.3
11: $\quad$coverage$'$, $\mathcal{D}'_{\text{errors}} \leftarrow$ `Eval`$(\theta_{\text{new}}, \mathcal{D}_{\text{test}}, \mathcal{D}_{\text{train}})$
12: $\quad$beta$' \leftarrow$ Beta$(\alpha + C \cdot \text{coverage}', \beta + C \cdot (1 - \text{coverage}'))$
13: $\quad$**insert** $(\theta'_{\text{new}}, \text{beta}', \text{coverage}')$ **into** $\mathcal{M}$
14: **end while**
15: **return** $\text{argmax}_{\mathcal{M}}(\text{coverage})[0]$ ▷ Return the model with the best overall coverage

Following the initial candidate model proposal, we evaluate them against the empirical conditional probability distributions observed in the demonstration data (Line 5). Determining if a particular outcome is possible under an arbitrary code model is not analytically possible in most probabilistic programming languages, including Pyro, so we use Monte Carlo approximation of the model density. Any empirical sample that is never produced is treated as a failure and recorded in an error set $\mathcal{D}_{\text{errors}}$. During evaluation, we estimate the model's coverage on a combination of the training and testing sets, evaluating models based on their ability to generalize beyond the training examples (Line 5).

We proceed with an iterative learning procedure that uses a Thompson sampling exploration strategy to build out a tree of candidate models. In addition to a candidate code block, each node contains a Beta distribution capturing uncertainty over its true coverage performance. Initially, the root node contains the LLM's first code proposal evaluated against the data (Line 7). At each iteration, we select a node to expand using Thompson sampling: we sample from each node's Beta distribution and pick the node with the highest sampled value (Line 9).

The selected node is refined by prompting the LLM with its associated training set coverage mismatch errors $\mathcal{D}_{\text{errors}}$, encouraging the LLM to generate a corrected or improved version of the model (Line 10, see Appendix F.3 for prompt). This produces a child node with updated code, which is re-evaluated to obtain new train and test coverage statistics (Line 11). The parent's Beta distribution is updated using a smoothing constant C to encourage stable learning from finite samples, and the child node is added to the tree (Line 13).

This iterative process continues until the overall coverage across the empirical distribution is sufficiently high, or until a pre-defined iteration budget is exhausted. Ultimately, we return the candidate model with the highest empirical coverage among all nodes (Line 26).

B Belief-Space Planner

Our online planning routine conducts forward search directly in belief space, determinizing the stochastic dynamics to enable an A*-style expansion strategy that balances exploitation (reward) and exploration (information gain). Algorithm 3 provides the complete procedure.

Algorithm 3 `BeliefPlanner`

1: **Input:** initial belief b_0, models $\mathcal{T}$, $\mathcal{O}$, $\mathcal{R}$, horizon H, action cost c, hyper-parameters λ, α
2: **Output:** best first action a^*
3: Open $\leftarrow \{(b_0, g{=}0)\}$, Closed$\leftarrow \emptyset$, Cost$\leftarrow \{b_0 : 0\}$
4: **while** Open $\neq \emptyset$ **and** iterations $< H$ **do**
5: $(b, g) \leftarrow$ pop_lowest(Open)
6: **if** b is terminal **then**
7: **continue**
8: **end if**
9: **if** $b \in$ Closed **then**
10: **continue**
11: **end if**
12: add b to Closed
13: **for each** action $a \in \mathcal{A}$ **do**
14: Draw n state particles $s' \sim \mathcal{T}(s, a)$ for $s \sim b$
15: Draw observations $o \sim \mathcal{O}(s', a)$
16: Form child belief $b' = \texttt{Branch}(b, a, o)$
17: $\hat{r} \leftarrow \mathbb{E}_{s,s'}[\mathcal{R}(s, a, s')]$
18: $\hat{p} \leftarrow \Pr[o \mid b, a], \quad \hat{h} \leftarrow H(b')$
19: $g' \leftarrow g - \hat{r} - \lambda \log \hat{p} + \alpha \hat{h} + c$
20: **if** $b' \notin$ Cost **or** $g' <$ Cost$[b']$ **then**
21: Cost$[b'] \leftarrow g'$
22: insert (b', g') into Open
23: **end if**
24: **end for**
25: **end while**
26: **return** first action in the path to the node in Open $\cup$ Closed with minimal g

We begin by inserting the initial belief b_0 into an *open* priority queue with zero cost-to-come and initializing an empty *closed* set (Line 3). Each queue element stores the belief, its cumulative cost g, and bookkeeping metadata such as depth. During each iteration (Line 4), we pop the node with the lowest priority value; if it is terminal or has already been expanded (i.e., in the closed set), we skip further expansion (Lines 6–9).

Otherwise, for every action a (Line 13), we draw next-state particles from the transition model $\mathcal{T}$ (Line 14) and sample the corresponding observations through the observation model $\mathcal{O}$ (Line 15). Conditioning on the sampled observation yields a child belief b' (Line 16). We estimate the expected reward $\hat{r}$ under $\mathcal{R}$, the likelihood $\hat{p}$ of the observation, and the entropy $\hat{h}$ of b' (Lines 17–18). These statistics define the child's cost

$$g' \;=\; g \;-\; \hat{r} \;-\; \lambda \log \hat{p} \;+\; \alpha \hat{h} \;+\; \text{cost}(a), \tag{3}$$

where λ and α trade off risk sensitivity and information gain (Line 19). The child node is inserted into the queue only if it is not yet discovered or has a lower cumulative cost than a previously seen version (Line 20). The process continues until the queue is empty or a computational budget is exhausted.

Finally, we return the first action in the path from b_0 to the node with the minimum accumulated cost (Line 26), thereby maximizing the composite objective of long-term reward, low risk, and maximal information gathering.

While this planner is similar in many ways to POUCT Sunberg and Kochenderfer (2017), it has the additional feature that enables graph-based search rather than strictly tree-based search, which proved computationally necessary for many of our larger MiniGrid problems. It is important to note that this planner is not optimal, but it suffices for all of the problems we tested, and can easily be substituted for other planning methods in the POMDP Coder framework.

C Minigrid Environment Details

In the *empty* environment, the agent is deterministically placed in the top left cell of a 5×5 grid and must navigate to the green square in the bottom right. In the *corners* environment, the agent is randomly positioned with an arbitrary orientation in a 10×10 grid while the green square appears in a randomly selected corner. In the *lava* environment, the agent starts in the upper left corner of a 10×10 grid that features a randomly positioned column of lava with a gap forming a narrow passage to the green square. In the *unlock* environment, the agent is randomly placed in the left room of a two-room layout, must collect a randomly placed key to open a locked door, and then proceeds to the green square fixed at the center of the right room. In the *rooms* environment, the agent is initialized in the upper right-hand corner of a multi-room setting and must traverse through the rooms to reach the green square randomly located in the bottom right room.

D API Interfaces

D.1 Tiger

```python
from __future__ import annotations

import copy
import enum
import random
from dataclasses import dataclass
from typing import Any, Dict, List, Tuple

from uncertain_worms.structs import (
    Environment,
    Heuristic,
    InitialModel,
    Observation,
    ObservationModel,
    RewardModel,
    State,
    TransitionModel,
)


# --------------------------- Enums & Dataclasses -----------------------------#



class TigerObservationEnum(enum.IntEnum):
    """Possible observations the agent can receive."""
    HEAR_LEFT = 0, HEAR_RIGHT = 1, NONE = 2

class TigerActions(enum.IntEnum):
    """Agent actions in the classic Tiger problem."""
    OPEN_LEFT = 0
    OPEN_RIGHT = 1
    LISTEN = 2

@dataclass(frozen=True)
class TigerObservation(Observation):
    """Observation dataclass."""
    obs: int # 0 = hear left, 1 = hear right, 2 = none

@dataclass(frozen=True)
```

```python
class TigerState(State):
    """Underlying hidden state: tiger behind LEFT (0) or RIGHT (1) door."""
    tiger_location: int  # 0 = left, 1 = right
```

D.2 Rock Sample

```python
import copy
import enum
import random
from dataclasses import dataclass
from typing import Any, Dict, List, Tuple

from uncertain_worms.structs import (
    Environment,
    Heuristic,
    InitialModel,
    Observation,
    ObservationModel,
    RewardModel,
    State,
    TransitionModel,
)



# ---------------------------------------------------------------------------#
# Domain parameters (feel free to tweak)                                     #
# ---------------------------------------------------------------------------#
NUM_ROCKS    = 2
GRID_SIZE    = 5
ROCK_POSITIONS = [(1, 1), (1, 4)]    # len == NUM_ROCKS

# ---------------------------------------------------------------------------#
# Actions                                                                    #
# ---------------------------------------------------------------------------#
class RockSampleActions(enum.IntEnum):
    """They can be added directly to the state position."""
    MOVE_NORTH  = 0
    MOVE_SOUTH  = 1
    MOVE_EAST   = 2
    MOVE_WEST   = 3
    SAMPLE      = 4
    EXIT        = 5

    CHECK_ROCK_0 = 6
    CHECK_ROCK_1 = 7


CHECK_ACTIONS: List[int] = [
    RockSampleActions.CHECK_ROCK_0,
    RockSampleActions.CHECK_ROCK_1,
]

# ---------------------------------------------------------------------------#
# Observation                                                                #
# ---------------------------------------------------------------------------#
@dataclass
class RockSampleObservation(Observation):
    """Observation after an action.
```

```
    Always embeds the rover pose (x, y).
    For sensor actions:
        * ``rock_idx``  - index of inspected rock
        * ``is_good``   - noisy reading (True = GOOD, False = BAD)
    For all other actions both fields are ``None``.
    """
    x: int
    y: int
    rock_idx: int | None
    is_good: bool | None

# ---------------------------------------------------------------------------#
# State                                                                      #
# ---------------------------------------------------------------------------#
@dataclass
class RockSampleState(State):
    """Full underlying state (fully observable to the simulator)."""
    x: int
    y: int
    rocks: Tuple[bool, ...]   # immutable tuple of good/bad flags

    # --- Equality / hashing -------------------------------------------------
    def __eq__(self, other: object) -> bool:                  # type:
    ↪  ignore[override]
        return (
            isinstance(other, RockSampleState)
            and self.x == other.x
            and self.y == other.y
            and self.rocks == other.rocks
        )

    # Convenience -------------------------------------------------------------
    def at_rock(self) -> int | None:
        """Return the index of the rock at the agent's (x,y) or ``None``."""
        try:
            return ROCK_POSITIONS.index((self.x, self.y))
        except ValueError:
            return None
```

D.3 Minigrid

```
from dataclasses import dataclass
from enum import IntEnum
from typing import Any, List, Optional, Tuple

import numpy as np
from numpy.typing import NDArray

AGENT_DIR_TO_STR = {0: ">", 1: "V", 2: "<", 3: "^"}
DIR_TO_VEC = [
    # Pointing right (positive X)
    np.array((1, 0)),
    # Down (positive Y)
    np.array((0, 1)),
    # Pointing left (negative X)
    np.array((-1, 0)),
    # Up (negative Y)
    np.array((0, -1)),
]

SEE_THROUGH_WALLS = True
```

```python
class ObjectTypes(IntEnum):
    unseen = 0
    empty = 1
    wall = 2
    open_door = 4
    closed_door = 5
    locked_door = 6
    key = 7
    ball = 8
    box = 9
    goal = 10
    lava = 11
    agent = 12


class Direction(IntEnum):
    facing_right = 0
    facing_down = 1
    facing_left = 2
    facing_up = 3


class Actions(IntEnum):
    left = 0  # Turn left
    right = 1  # Turn right
    forward = 2  # Move forward
    pickup = 3  # Pick up an object
    drop = 4  # Drop an object
    toggle = 5  # Toggle/activate an object
    done = 6  # Done completing the task


@dataclass
class MinigridObservation(Observation):
    """
    Args:
        `image`: field of view in front of the agent.

        `agent_pos`: agent's position in the real world. It differs from the
        ↪  position
                  in the observation grid.
        `agent_dir`: agent's direction in the real world. It differs from the
        ↪  direction
                  of the agent in the observation grid.
        `carrying`: what the agent is carrying at the moment.
    """

    image: NDArray[np.int8]
    agent_pos: Tuple[int, int]
    agent_dir: int
    carrying: Optional[int] = None


@dataclass
class MinigridState(State):
    """An agent exists in an indoor multi-room environment represented by a
    grid."""

    grid: NDArray[np.int8]
    agent_pos: Tuple[int, int]
    agent_dir: int
    carrying: Optional[int]
```

```python
    @property
    def front_pos(self) -> Tuple[int, int]:
        """Get the position of the cell that is right in front of the agent."""

        return (
            np.array(self.agent_pos) + np.array(DIR_TO_VEC[self.agent_dir])
        ).tolist()

    @property
    def width(self) -> int:
        return self.grid.shape[0]

    @property
    def height(self) -> int:
        return self.grid.shape[1]

    def get_type_indices(self, type: int) -> List[Tuple[int, int]]:
        idxs = np.where(self.grid == type)  # Returns (row_indices, col_indices)
        return list(zip(idxs[0], idxs[1]))  # Combine row and column indices


    def get_field_of_view(self, view_size: int) -> NDArray[np.int8]:
        """Returns the field of view in front of the agent.

        DO NOT modify this function.
        """

        # Get the extents of the square set of tiles visible to the agent
        # Facing right
        if self.agent_dir == 0:
            topX = self.agent_pos[0]
            topY = self.agent_pos[1] - view_size // 2
        # Facing down
        elif self.agent_dir == 1:
            topX = self.agent_pos[0] - view_size // 2
            topY = self.agent_pos[1]
        # Facing left
        elif self.agent_dir == 2:
            topX = self.agent_pos[0] - view_size + 1
            topY = self.agent_pos[1] - view_size // 2
        # Facing up
        elif self.agent_dir == 3:
            topX = self.agent_pos[0] - view_size // 2
            topY = self.agent_pos[1] - view_size + 1
        else:
            assert False, "invalid agent direction"

        fov = np.full((view_size, view_size), ObjectTypes.wall,
        ↪  dtype=self.grid.dtype)

        # Compute the overlapping region in the grid.
        gx0 = max(topX, 0)
        gy0 = max(topY, 0)
        gx1 = min(topX + view_size, self.grid.shape[0])
        gy1 = min(topY + view_size, self.grid.shape[1])

        # Determine where the overlapping region goes in the padded array.
        px0 = max(0, -topX)
        py0 = max(0, -topY)

        # Copy the overlapping slice.
        fov[px0 : px0 + (gx1 - gx0), py0 : py0 + (gy1 - gy0)] = self.grid[
            gx0:gx1, gy0:gy1
```

```
        ]

        for _ in range(self.agent_dir + 1):
            # Rotate left
            fov = np.rot90(fov.T, k=1).T

        agent_pos = (self.grid.shape[0] // 2, self.grid.shape[1] - 1)
        self.grid[agent_pos] = ObjectTypes.agent

        return fov
```

D.4 Spot exploration

```
import copy
import logging
import math
import random
from dataclasses import dataclass, field
from enum import IntEnum
from typing import Any, List, Optional, Tuple

import numpy as np
from numpy.typing import NDArray
from scipy.spatial.transform import Rotation as R

import uncertain_worms.environments.spot.pb_utils as pbu
from uncertain_worms.environments.spot.spot_constants import *
from uncertain_worms.structs import Observation, State

log = logging.getLogger(__name__)


NAVIGATION_STEP_SIZE = 5  # size of each step in the navigation
FRUSTUM_DEPTH = 3.0
ROTATION_ANGLE = [i * np.pi / 4.0 for i in range(8)]  # Angles for the robot to
↪  rotate
PICKUP_DISTANCE_THRESHOLD = 2.0  # Adjust this value as needed


class SpotActions(IntEnum):
    move_left = 0
    move_right = 1
    move_forward = 2
    move_backward = 3
    rotate_left = 4
    rotate_right = 5
    pickup = 6  # pick up the object if the object is in the camera's view


ARM_CONF = "ARM_STOW"


@dataclass
class AABB:
    lower: List[float, float, float]
    upper: List[float, float, float]


def pose_to_se2(pose):
    return [pose[0][0], pose[0][1], pbu.euler_from_quat(pose[1])[2]]
```

```python
def se2_to_pose(se2):
    return pbu.Pose(point=pbu.Point(x=se2[0], y=se2[1]),
    ↪  euler=pbu.Euler(yaw=se2[2]))


def transformation_matrix(
    translation: NDArray[np.float64], quat: NDArray[np.float64]
) -> NDArray[np.float64]:
    r = R.from_quat(quat)
    rotation_matrix = r.as_matrix()
    T = np.eye(4)
    T[:3, :3] = rotation_matrix
    T[:3, 3] = translation
    return T

class SpotActions(IntEnum):
    move_left = 0
    move_right = 1
    move_forward = 2
    move_backward = 3
    rotate_left = 4
    rotate_right = 5
    arm_stow = 6
    arm_left = 7
    arm_right = 8
    arm_down = 9
    pickup = 10  # pick up the object if the object is in the camera's view


@dataclass
class SceneObject:
    name: str
    location: List[int]
    aabb: AABB = None

    def __hash__(self) -> int:
        return hash((self.name, tuple(self.location)))

    def __eq__(self, other: Any) -> bool:
        return hash(other) == hash(self)

    def __repr__(self) -> str:
        return (
            'SceneObject(name="'
            + str(self.name)
            + '", location='
            + str(self.location)
            + ", aabb="
            + str(self.aabb)
            + ")"
        )


@dataclass
class SpotState(State):
    body_location: List[
        int
    ]  # x voxel index, y voxel index, rotation index into ROTATION_ANGLE
    occupancy_grid: OccupancyGrid
    visibility_grid: VisibilityGrid
    movable_objects: List[SceneObject] = field(default_factory=list)
    fixed_objects: List[SceneObject] = field(default_factory=list)
    carry_object: Optional[SceneObject] = None
```

```python
    ons: List[Tuple[str, str]] = field(
        default_factory=list
    )  # what object is on what other object

    def __repr__(self) -> str:
        return f"SpotState(body_location={self.body_location},
        ↪  movable_objects={str([o for o in self.movable_objects])},
        ↪  carry_object={self.carry_object}, ons={self.ons},
        ↪  fixed_objects={str([o for o in self.fixed_objects])})"

    @property
    def camera_pose(self):
        return pbu.multiply(
            se2_to_pose(self.occupancy_grid.to_world(self.body_location)),
            CAMERA_POSES[ARM_CONF],
        )

@dataclass
class SpotObservation(Observation):
    body_location: List[
        int
    ]  # x voxel index, y voxel index, rotation index into ROTATION_ANGLE
    visible_movable_objects: List[SceneObject] = field(default_factory=list)
    carry_object: Optional[SceneObject] = None

    def __repr__(self) -> str:
        return f"SpotObservation(body_location={self.body_location},
        ↪  carry_object={self.carry_object}, visible_movable_objects={str([o for
        ↪  o in self.visible_movable_objects])})"

    @property
    def camera_pose(self):
        return CAMERA_POSES[ARM_CONF]


class OccupancyGrid:
    def check_collision(self, body_location: Tuple[int, int, int]) -> bool:
        """Returns the collision result for the given robot body location."""
        ...

    def from_world(
        self, world_state: Tuple[float, float, float]
    ) -> Tuple[int, int, int]:
        """Converts a world state (x, y, theta) into a discrete occupancy grid
        state (row, col, theta_index)."""
        ...

    def to_world(
        self, occupancy_grid_state: Tuple[int, int, int]
    ) -> Tuple[float, float, float]:
        """Converts an occupancy grid state (row, col, theta_index) to world
        coordinates."""
        ...

    @property
    def grid_size(self) -> Tuple[int, int]:
        """Returns the size of the occupancy grid."""
        ...


class VisibilityGrid:
    def from_world(
        self, world_state: Tuple[float, float, float]
    ) -> Tuple[int, int, int]:
```

```
        """Converts a world coordinates (x, y, theta) into a discrete
        visibility grid state (row, col, theta_index)."""
        ...

    def to_world(self, grid_state: Tuple[int, int, int]) -> Tuple[float, float,
    ↪   float]:
        """Converts a visibility grid state (row, col, theta_index) to world
        coordinates."""
        ...

    def get_voxels_above_aabb(self, aabb: AABB) -> NDArray[np.int64]:
        """Returns the indices of voxels in the visibility grid whose centers
        are directly above the given aabb.

        IMPORTANT: You must use this function to get the location of a goal that
        ↪   is on top of a fixed object.
        """
        ...

    @property
    def grid_size(self) -> Tuple[int, int, int]:
        """Returns the size of the occupancy grid."""
        ...
```

E Hyperparameters

Table E shows the hyperparameters used per domain for both planning and learning. The planning hyperparameters (in grey) were tuned to work best for the ground truth models used in oracle. With the exception of Thompson smoothing coefficient which was selected based on Tang et al. (2024), the other hyperparameters were selected to be as large as possible under computational and budgetary constraints.

Hyperparameter	Classical	MiniGrid	Spot Robot
Action cost penalty	0.01	0.01	0.01
α (Entropy coefficient)	0.0	0.0	1.0
λ (log-prob reward shaping)	0.1	0.1	0.1
Rollouts per stochastic model query	5	1	1
H (Max expansions)	50	5000	5000
N (Num initial particles),	50	10	10
Max particle rejuvenations,	2,500,00	500,000	25,000
M Max refinements	25	25	25
C Thompson smoothing	25	25	25
ND (# datapoints shown - initial, F.2)	5	5	5
NC (# conditions shown - refinement, F.3)	5	5	5
NS (# samples per condition - refinement, F.3)	5	5	5

F Prompts

F.1 Function Templates

```
def initial_func(empty_state:MiniGridState):
    """
    Input:
        empty_state (MiniGridState): An empty state with only the walls filled
        ↪   into the grid
```

```
    Returns:
        state (MiniGridState): the initial state of the environment
    """
    raise NotImplementedError
```

```
def observation_func(state, action, empty_obs):
    """
    Args:
        state (MiniGridState): the state of the environment
        action (int): the previous action that was taken
        empty_obs (MiniGridObservation): an empty observation that needs to be
        ↪  filled and returned
    Returns:
        obs (MiniGridObservation): observation of the agent
    """
    raise NotImplementedError
```

```
def reward_func(state, action, next_state):
    """
    Args:
        state (MiniGridState): the state of the environment
        action (int): the action to be executed
        next_state (MiniGridState): the next state of the environment
    Returns:
        reward (float): the reward of that state
        done (bool): whether the episode is done
    """
    raise NotImplementedError
```

```
def transition_func(state, action):
    """
    Args:
        state (MiniGridState): the state of the environment
        action (int): action to be taken in state `state`
    Returns:
        new_state (MiniGridState): the new state of the environment
    """
    raise NotImplementedError
```

501 ### F.2 Initial Prompt

```
You are a robot exploring its environment.

Environment Description: {env_description}
Goal Description: {goal_description}

Your goal is to model the {what_to_model}.
You need to implement the python code to model the world, as seen in the provided
↪  experiences.
```

````
Please follow the template to implement the code.
The code needs to be directly runnable {model_input} and return {model_output}.


Below are a few samples from the environment distribution. These are only samples
↪  from a larger distribution that your should model.

{exp}

Here is the template for the {model_name} function. Please implement
the reward function following the template. The code needs to be directly
runnable.

‘‘‘
{code_api}

{code_template}
‘‘‘

Explain what you believe is the {what_to_model} in english.
Additionally, please implement code to model the logic of the world. Please
↪  implement the
code following the template. Only output the definition for ‘ {model_name} ‘.
You must implement the ‘ {model_name} ‘ function.
Create any helper function inside the scope of ‘ {model_name} ‘.
Do not create any helper function outside the scope of ‘ {model_name} ‘.
Do not output examples usage.
Do not create any new classes.
Do not rewrite existing classes.
Do not import any new modules from anywhere.
Do not overfit to the specific samples.
Put the ‘ {model_name} ‘ function in a python code block.
Implement any randomness with `pyro.sample`
````

F.3 Refinement Prompt

````
You are a robot exploring its environment.

{env_description}

Your goal is to model {what_to_model} of the world in python.

You have tried it before and came up with one partially correct solution, but it
↪  is not perfect.

The observed distribution disagrees with the generated model in several cases.
You need to improve your code to come closer to the true distribution.

Environment Description: {env_description}
Goal Description: {goal_description}

Here is a solution you came up with before.

```
{code_api}

{code}
```
````

```
{experiences}

Explain what you believe is {what_to_model} in english, then improve your code to
↪  better model the true distribution.

Please implement the code for the following the template.
You must implement the ‘ {model_name} ‘ function.

The code needs to be directly runnable {model_input} and return {model_output}.

Do not output examples.
Do not create any new classes.
Do not rewrite existing classes.
Do not import any new modules from anywhere.
Do not list out specific indices that overfit to the examples, but include ranges.
Put the ‘ {model_name} ‘ function in a python code block.
Implement any randomness with `pyro.sample`
```

The experiences in the refinement prompt are structured as follows. First, we sample conditions for which there is coverage less than 1. For example, in the transition model $P(s_{t+1}|s_t, a_t)$, we search through the set of (s_t, a_t) tuples in the database and we find the set of those tuples where the distribution over (s_{t+1}) contains at least 1 element that can not be achieved by the LLM-generated model. We select out NC of those conditions. Then, given those conditions, we select NS samples that were not covered by the model to show as examples to the LLM. An example template for a single condition and an $NS = 3$ is shown below. The values we used for NC, NS are in Table E.

```
Here are some samples from the real world that were impossible under your model
{condition} -> {dataset_outcome}
{condition} -> {dataset_outcome}
{condition} -> {dataset_outcome}

And here are some samples from your code under the same conditions
{condition} -> {model_outcome}
{condition} -> {model_outcome}
{condition} -> {model_outcome}
```

F.4 Direct LLM Baseline Prompt

```
You are a robot exploring its environment.

{env_description}

Your goal is to predict the next best action to take to reach the goal and
↪  maximize reward.

Here is the template for the reward function. Please implement
the reward function following the template. The code needs to be directly
runnable on the inputs of (state) and return (reward) in python.

‘‘‘
{code_api}
‘‘‘

Here are some example rollouts from the environment

{exp}

Here is the current episode history for the task that you are doing right now
```

```
{current_episode}

Output the next aciton in the form where you fill in <action-here> with the action
↪  that is best for reaching the goal and maximizing reward.
For example, your code will look like this:

```
next_action:int = 0
```

The action should be an integer with no additional code. Explan your reasoning in
↪ one sentence.
```

F.5 Runtime Statistics

Approach	Tiger	RockSample	Empty	Corners	Lava	Rooms	Unlock
Offline Transition	2.00 ± 0.00	2.60 ± 0.24	2.20 ± 0.20	2.00 ± 0.00	2.00 ± 0.00	2.60 ± 0.24	4.60 ± 0.81
Online Transition	0.00 ± 0.00	0.06 ± 0.06	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.40 ± 0.15
Offline Reward	2.40 ± 0.24	11.80 ± 4.33	2.00 ± 0.00	2.00 ± 0.00	2.20 ± 0.20	2.00 ± 0.00	2.20 ± 0.20
Online Reward	0.05 ± 0.05	0.05 ± 0.05	0.00 ± 0.00	0.00 ± 0.00	0.19 ± 0.09	0.00 ± 0.00	0.00 ± 0.00
Offline Observation	2.20 ± 0.20	5.80 ± 1.46	10.40 ± 4.01	7.80 ± 4.61	2.40 ± 0.24	17.25 ± 5.71	15.00 ± 3.63
Online Observation	0.00 ± 0.00	0.00 ± 0.00	0.05 ± 0.05	0.08 ± 0.06	0.00 ± 0.00	0.28 ± 0.17	0.05 ± 0.05
Offline Initial	2.00 ± 0.00	2.60 ± 0.24	2.00 ± 0.00	2.80 ± 0.37	22.60 ± 3.40	1.75 ± 0.25	18.80 ± 4.59
Online Initial	0.00 ± 0.00	0.02 ± 0.02	0.00 ± 0.00	0.17 ± 0.09	0.86 ± 0.21	0.06 ± 0.06	0.49 ± 0.15

Table 2: The average and standard deviation of the number of nodes, or LLM-generated candidate programs, sampled during the online and offline phases of model learning.

Approach	Tiger	RockSample	Empty	Corners	Lava	Rooms	Unlock
Transition Train	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
Transition Test	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
Reward Train	1.00 ± 0.00	0.99 ± 0.01	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
Reward Test	1.00 ± 0.00	0.99 ± 0.01	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
Observation Train	1.00 ± 0.00	1.00 ± 0.00	0.92 ± 0.08	0.81 ± 0.19	1.00 ± 0.00	0.94 ± 0.06	0.94 ± 0.06
Observation Test	1.00 ± 0.00	1.00 ± 0.00	0.92 ± 0.08	0.88 ± 0.12	1.00 ± 0.00	0.93 ± 0.06	0.94 ± 0.06
Initial Train	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	0.68 ± 0.16	0.75 ± 0.25	0.88 ± 0.12
Initial Test	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	0.56 ± 0.18	0.75 ± 0.25	0.24 ± 0.19

Table 3: The average and standard deviation of coverages achieved after the offline model learning step has completed running split into training and testing coverages.

G Example Rollout

Below is an example of a full rollout from the unlock minigrid environment for the version of our method that includes both offline and online learning of the initial state distribution. This rollout is only a single leaf node in the tree that is formed by Algorithm 2. This particular rollout required three iterations to reach full coverage and did not require additional online learning iterations. We remove duplicate code api definitions for clarity.

G.1 Iteration 0 Input

```
#define system
You are a robot exploring its environment.

Environment Description: Unlock door with key to reach the goal square
```

```
Goal Description:

Your goal is to model the the distribution of initial states .
You need to implement the python code to model the world, as seen in the provided
↪  experiences.
Please follow the template to implement the code.
The code needs to be directly runnable an empty state with the walls of the grid
↪  pre-filled and return a sample initial state.


Below are a few samples from the environment distribution. These are only samples
↪  from a larger distribution that your should model.

Output MinigridState:
agent_pos=(2, 3)
agent_dir=3
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  7,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(1, 1)
agent_dir=2
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  7,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(3, 2)
agent_dir=3
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  7,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
```

```
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(4, 3)
agent_dir=3
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  7,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(1, 4)
agent_dir=2
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  7,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]




Here is the template for the initial_func function. Please implement
the reward function following the template. The code needs to be directly
runnable.

‘‘‘
# type: ignore
from __future__ import annotations

from dataclasses import dataclass
from enum import IntEnum
from typing import Any, List, Optional, Tuple

import numpy as np
```

```
from numpy.typing import NDArray

AGENT_DIR_TO_STR = {0: ">", 1: "V", 2: "<", 3: "^"}
DIR_TO_VEC = [
    # Pointing right (positive X)
    np.array((1, 0)),
    # Down (positive Y)
    np.array((0, 1)),
    # Pointing left (negative X)
    np.array((-1, 0)),
    # Up (negative Y)
    np.array((0, -1)),
]

SEE_THROUGH_WALLS = True


class ObjectTypes(IntEnum):
    unseen = 0
    empty = 1
    wall = 2
    open_door = 4
    closed_door = 5
    locked_door = 6
    key = 7
    ball = 8
    box = 9
    goal = 10
    lava = 11
    agent = 12


class Direction(IntEnum):
    facing_right = 0
    facing_down = 1
    facing_left = 2
    facing_up = 3


class Actions(IntEnum):
    left = 0  # Turn left
    right = 1  # Turn right
    forward = 2  # Move forward
    pickup = 3  # Pick up an object
    drop = 4  # Drop an object
    toggle = 5  # Toggle/activate an object
    done = 6  # Done completing the task


@dataclass
class MinigridState(State):
    """An agent exists in an indoor multi-room environment represented by a
    grid."""

    grid: NDArray[np.int8]
    agent_pos: Tuple[int, int]
    agent_dir: int
    carrying: Optional[int]

    def __hash__(self) -> int:
        return hash(
            (
                tuple(self.agent_pos),
                self.agent_dir,
```

```
            self.carrying,
            self.grid.tobytes(),
        )
    )

def __eq__(self, other: object) -> bool:
    return (
        isinstance(other, MinigridState)
        and np.allclose(self.grid, other.grid)
        and tuple(self.agent_pos) == tuple(other.agent_pos)
        and self.agent_dir == other.agent_dir
        and self.carrying == other.carrying
    )

@property
def front_pos(self) -> Tuple[int, int]:
    """Get the position of the cell that is right in front of the agent."""

    return (
        np.array(self.agent_pos) + np.array(DIR_TO_VEC[self.agent_dir])
    ).tolist()

@property
def width(self) -> int:
    return self.grid.shape[0]

@property
def height(self) -> int:
    return self.grid.shape[1]

def get_type_indices(self, type: int) -> List[Tuple[int, int]]:
    idxs = np.where(self.grid == type)  # Returns (row_indices, col_indices)
    return list(zip(idxs[0], idxs[1]))  # Combine row and column indices

def get_field_of_view(self, view_size: int) -> NDArray[np.int8]:
    """Returns the field of view in front of the agent.

    DO NOT modify this function.
    """

    # Get the extents of the square set of tiles visible to the agent
    # Facing right
    if self.agent_dir == 0:
        topX = self.agent_pos[0]
        topY = self.agent_pos[1] - view_size // 2
    # Facing down
    elif self.agent_dir == 1:
        topX = self.agent_pos[0] - view_size // 2
        topY = self.agent_pos[1]
    # Facing left
    elif self.agent_dir == 2:
        topX = self.agent_pos[0] - view_size + 1
        topY = self.agent_pos[1] - view_size // 2
    # Facing up
    elif self.agent_dir == 3:
        topX = self.agent_pos[0] - view_size // 2
        topY = self.agent_pos[1] - view_size + 1
    else:
        assert False, "invalid agent direction"

    fov = np.full((view_size, view_size), ObjectTypes.wall,
    ↪  dtype=self.grid.dtype)

    # Compute the overlapping region in the grid.
```

```
        gx0 = max(topX, 0)
        gy0 = max(topY, 0)
        gx1 = min(topX + view_size, self.grid.shape[0])
        gy1 = min(topY + view_size, self.grid.shape[1])

        # Determine where the overlapping region goes in the padded array.
        px0 = max(0, -topX)
        py0 = max(0, -topY)

        # Copy the overlapping slice.
        fov[px0 : px0 + (gx1 - gx0), py0 : py0 + (gy1 - gy0)] = self.grid[
            gx0:gx1, gy0:gy1
        ]

        for _ in range(self.agent_dir + 1):
            # Rotate left
            fov = np.rot90(fov.T, k=1).T

        return fov

    def __repr__(self) -> str:
        """Returns a string representation of the grid with agent position."""

        print_agent = False
        print_state = "agent_pos={}\n".format(self.agent_pos)
        print_state += "agent_dir={}\n".format(self.agent_dir)
        print_state += "carrying={}\n".format(self.carrying)
        print_state += "grid=[\n"
        for x in range(self.width):
            row = "["
            for y in range(self.height):
                if [x, y] == list(self.agent_pos) and print_agent:
                    row += f" {AGENT_DIR_TO_STR[self.agent_dir]}, "
                else:
                    row += f"{self.grid[x, y]:2d}, "
            row += "],\n"
            print_state += row
        print_state += "]\n"
        return print_state


@dataclass
class MinigridObservation(Observation):
    """
    Represents the non-centered field of view of the agent.
    The agent is NOT in the center of the observation grid.
    Observation grids are always square-sizes (i.e. 3x3, 5x5, 7x7).
    The width and height of the observation grid are called view size.
    The agent is ALWAYS in the observation and ALWAYS at the same spot
    in the observation `image`, independent of the observation.
    The experiences are printed through the `__repr__` function.
    Args:
        `image`: field of view in front of the agent.

        `agent_pos`: agent's position in the real world. It differs from the
        ↪  position
                     in the observation grid.
        `agent_dir`: agent's direction in the real world. It differs from the
        ↪  direction
                     of the agent in the observation grid.
        `carrying`: what the agent is carrying at the moment.
    """

    image: NDArray[np.int8]
```

```
    agent_pos: Tuple[int, int]
    agent_dir: int
    carrying: Optional[int] = None

    def __eq__(self, other: object) -> bool:
        return (
            isinstance(other, MinigridObservation)
            and np.allclose(self.image, other.image)
            and tuple(self.agent_pos) == tuple(other.agent_pos)
            and self.agent_dir == other.agent_dir
            and self.carrying == other.carrying
        )

    def __hash__(self) -> int:
        return hash(
            (self.image.tobytes(), tuple(self.agent_pos), self.agent_dir,
            ↪  self.carrying)
        )

    def __repr__(self) -> str:
        return str(
            MinigridState(
                self.image,
                agent_pos=self.agent_pos,
                agent_dir=self.agent_dir,
                carrying=self.carrying,
            )
        )

def initial_func(empty_state:MinigridState):
    """
    Input:
        empty_state (MinigridState): An empty state with only the walls filled
        ↪  into the grid
    Returns:
        state (MinigridState): the initial state of the environment
    """
    raise NotImplementedError
‘‘‘

Explain what you believe is the the distribution of initial states  in english.
Additionally, please implement code to model the logic of the world. Please
↪  implement the
code following the template. Only output the definition for ‘ initial_func ‘.
You must implement the ‘ initial_func ‘ function.
Create any helper function inside the scope of ‘ initial_func ‘.
Do not create any helper function outside the scope of ‘ initial_func ‘.
Do not output examples usage.
Do not create any new classes.
Do not rewrite existing classes.
Do not import any new modules from anywhere.
Do not overfit to the specific samples.
Put the ‘ initial_func ‘ function in a python code block.
```

519
G.2 Iteration 0 Output

```
The distribution of initial states describes an environment where the agent is
↪  placed on a grid surrounded predominantly by walls. There is a locked door
↪  that blocks the path, and a key is placed somewhere in the grid for the agent
↪  to collect. The initial states vary primarily in the starting position and
↪  direction of the agent, and the position of the key. The door is initially
↪  locked, and the agent must pick up the key before it can pass through the door
↪  to reach the goal.

The observed patterns in the sample outputs:

1. The grid is 11x6 in size with outer walls filling the entire perimeter,
↪  represented by '2'.
2. Within the grid:
   - The locked door is always at position (5,3), represented by '6'.
   - The goal is always at position (8,3), represented by '10'.
   - The goal is behind the locked door.
   - The key can be at any of the positions marked as '7'. Mainly, rows containing
   ↪  the agent.
3. The agent's initial position and direction vary but are within the traversable
↪  area within the outer walls.
4. The agent is always initially not carrying any object (carrying=None).

The below code `initial_func` aims to reflect this analysis by placing the agent,
↪  door, goal, and key consistent with the described logic.

```python
def initial_func(empty_state: MinigridState):
    """
    Input:
        empty_state (MinigridState): An empty state with only the walls filled
        ↪  into the grid
    Returns:
        state (MinigridState): the initial state of the environment
    """

    # Create the grid with walls, locked door, key, and goal
    grid = np.array([
        [2, 2, 2, 2, 2, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 2, 2, 6, 2, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 10, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 2, 2, 2, 2, 2],
    ], dtype=np.int8)

    # Random choice for the agent's position within the traversable area (ensuring
    ↪  no overlapping with door and goal)
    possible_initial_positions = [(1, 1), (1, 2), (1, 3), (1, 4),
                                  (2, 1), (2, 2), (2, 3), (2, 4),
                                  (3, 1), (3, 2), (3, 3), (3, 4),
                                  (4, 1), (4, 3)]
    agent_pos =
    ↪  possible_initial_positions[np.random.choice(len(possible_initial_positions))]

    # Random choice for the agent's initial direction (facing one of four possible
    ↪  directions)
    agent_dir = np.random.choice([0, 1, 2, 3])

    # Place the key randomly in one of the valid initial positions
```
```

~~~~
    key_position_options = [(1, 1), (1, 2), (1, 3), (1, 4),
                            (2, 1), (2, 2), (2, 3), (2, 4),
                            (3, 1), (3, 2), (3, 3), (3, 4),
                            (4, 1), (4, 3)]
    key_position =
    ↪  key_position_options[np.random.choice(len(key_position_options))]
    grid[key_position] = ObjectTypes.key

    # Return the initial state with the agent randomly positioned on the grid,
    ↪  initially carrying nothing
    return MinigridState(
        grid=grid,
        agent_pos=agent_pos,
        agent_dir=agent_dir,
        carrying=None
    )
```
~~~~

G.3 Iteration 1 Input

~~~~
#define system
You are a robot exploring its environment.

Unlock door with key to reach the goal square

Your goal is to model the distribution of initial states  of the world in python.

You have tried it before and came up with one partially correct solution, but it
↪  is not perfect.

The observed distribution disagrees with the generated model in several cases.
You need to improve your code to come closer to the true distribution.

Environment Description: Unlock door with key to reach the goal square
Goal Description:

Here is a solution you came up with before.

```
{code_api}

def initial_func(empty_state: MinigridState):
    """
    Input:
        empty_state (MinigridState): An empty state with only the walls filled
        ↪  into the grid
    Returns:
        state (MinigridState): the initial state of the environment
    """

    # Create the grid with walls, locked door, key, and goal
    grid = np.array([
        [2, 2, 2, 2, 2, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 2, 2, 6, 2, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
~~~~

```
        [2, 1, 1, 10, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 2, 2, 2, 2, 2],
    ], dtype=np.int8)

    # Random choice for the agent's position within the traversable area (ensuring
    ↪  no overlapping with door and goal)
    possible_initial_positions = [(1, 1), (1, 2), (1, 3), (1, 4),
                                  (2, 1), (2, 2), (2, 3), (2, 4),
                                  (3, 1), (3, 2), (3, 3), (3, 4),
                                  (4, 1), (4, 3)]
    agent_pos =
    ↪  possible_initial_positions[np.random.choice(len(possible_initial_positions))]

    # Random choice for the agent's initial direction (facing one of four possible
    ↪  directions)
    agent_dir = np.random.choice([0, 1, 2, 3])

    # Place the key randomly in one of the valid initial positions
    key_position_options = [(1, 1), (1, 2), (1, 3), (1, 4),
                            (2, 1), (2, 2), (2, 3), (2, 4),
                            (3, 1), (3, 2), (3, 3), (3, 4),
                            (4, 1), (4, 3)]
    key_position =
    ↪  key_position_options[np.random.choice(len(key_position_options))]
    grid[key_position] = ObjectTypes.key

    # Return the initial state with the agent randomly positioned on the grid,
    ↪  initially carrying nothing
    return MinigridState(
        grid=grid,
        agent_pos=agent_pos,
        agent_dir=agent_dir,
        carrying=None
    )

```

```
Here are some samples from the real world that were impossible under your model
Output MinigridState:
agent_pos=(1, 1)
agent_dir=2
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  7,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]




And here are some samples from your code under the same conditions
Output MinigridState:
agent_pos=(1, 3)
```

```
agent_dir=1
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  7,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(3, 3)
agent_dir=1
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  7,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(1, 2)
agent_dir=1
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  7,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(1, 2)
agent_dir=2
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
```

```
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  7,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(2, 1)
agent_dir=1
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  7,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]




Explain what you believe is the distribution of initial states  in english, then
↪  improve your code to better model the true distribution.

Please implement the code for the following the template.
You must implement the ‘ initial_func ‘ function.

The code needs to be directly runnable an empty state with the walls of the grid
↪  pre-filled and return a sample initial state.

Do not output examples.
Do not create any new classes.
Do not rewrite existing classes.
Do not import any new modules from anywhere.
Do not list out specific indices that overfit to the examples, but include ranges.
Put the ‘ initial_func ‘ function in a python code block.
```

521 ## G.4 Iteration 1 Output

```
The distribution of initial states in this environment should consider the grid's
↪  layout and the constraints on various object placements. The initial state
↪  should not only randomly position the agent within specific traversable tiles,
↪  but it should also place the key within valid tiles, excluding the agent's
↪  initial position. My previous implementation missed this exclusion condition.
↪  The initial world state should allow the key to be anywhere in the agent's
↪  reachable area, except where the agent itself is positioned.

Below is the improved code addressing the observed discrepancies:

```python
def initial_func(empty_state: MinigridState):
    """
    Input:
        empty_state (MinigridState): An empty state with only the walls filled
        ↪  into the grid
    Returns:
        state (MinigridState): the initial state of the environment
    """

    # Create the grid with walls, locked door, key, and goal
    grid = np.array([
        [2, 2, 2, 2, 2, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 2, 2, 6, 2, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 10, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 2, 2, 2, 2, 2],
    ], dtype=np.int8)

    # Generate possible positions for the agent and key
    possible_positions = [(x, y) for x in range(1, 5) for y in range(1, 5)]
    possible_positions.remove((4, 2))  # Remove fixed key position for example
    ↪  compatibility

    # Randomly choose the agent's position
    agent_pos = possible_positions[np.random.choice(len(possible_positions))]

    # If the agent's position overlaps with the key's recognizable positions,
    ↪  choose again for the key
    key_position_options = [pos for pos in possible_positions if pos != agent_pos]
    key_position =
    ↪  key_position_options[np.random.choice(len(key_position_options))]
    grid[key_position] = ObjectTypes.key

    # Random choice for the agent's initial direction (facing one of four possible
    ↪  directions)
    agent_dir = np.random.choice([0, 1, 2, 3])

    # Return the initial state with the agent randomly positioned on the grid,
    ↪  initially carrying nothing
    return MinigridState(
        grid=grid,
        agent_pos=agent_pos,
        agent_dir=agent_dir,
        carrying=None
    )
```
```

```
This code improves on the initial model by correctly simulating the distribution
↪  of initial states, ensuring that the key is placed at a viable location in the
↪  environment, distinct from the agent's initial position.
```

G.5 Iteration 2 Input

```
#define system
You are a robot exploring its environment.

Unlock door with key to reach the goal square

Your goal is to model the distribution of initial states  of the world in python.

You have tried it before and came up with one partially correct solution, but it
↪  is not perfect.

The observed distribution disagrees with the generated model in several cases.
You need to improve your code to come closer to the true distribution.

Environment Description: Unlock door with key to reach the goal square
Goal Description:

Here is a solution you came up with before.

```
{code_api}

def initial_func(empty_state: MinigridState):
    """
    Input:
        empty_state (MinigridState): An empty state with only the walls filled
        ↪ into the grid
    Returns:
        state (MinigridState): the initial state of the environment
    """

    # Create the grid with walls, locked door, key, and goal
    grid = np.array([
        [2, 2, 2, 2, 2, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 2, 2, 6, 2, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 10, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 2, 2, 2, 2, 2],
    ], dtype=np.int8)

    # Generate possible positions for the agent and key
    possible_positions = [(x, y) for x in range(1, 5) for y in range(1, 5)]
    possible_positions.remove((4, 2)) # Remove fixed key position for example
    ↪ compatibility

    # Randomly choose the agent's position
    agent_pos = possible_positions[np.random.choice(len(possible_positions))]

    # If the agent's position overlaps with the key's recognizable positions,
    ↪ choose again for the key
```
```

```
    key_position_options = [pos for pos in possible_positions if pos != agent_pos]
    key_position =
    ↪  key_position_options[np.random.choice(len(key_position_options))]
    grid[key_position] = ObjectTypes.key

    # Random choice for the agent's initial direction (facing one of four possible
    ↪  directions)
    agent_dir = np.random.choice([0, 1, 2, 3])

    # Return the initial state with the agent randomly positioned on the grid,
    ↪  initially carrying nothing
    return MinigridState(
        grid=grid,
        agent_pos=agent_pos,
        agent_dir=agent_dir,
        carrying=None
    )

```
```

```
Here are some samples from the real world that were impossible under your model
Output MinigridState:
agent_pos=(1, 1)
agent_dir=2
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  7,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



And here are some samples from your code under the same conditions
Output MinigridState:
agent_pos=(2, 2)
agent_dir=1
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  7,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]
```
```

```
Output MinigridState:
agent_pos=(4, 1)
agent_dir=2
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  7,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(3, 2)
agent_dir=1
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  7,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(2, 1)
agent_dir=1
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  7,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(4, 4)
agent_dir=0
carrying=None
```

```
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  7,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]




Explain what you believe is the distribution of initial states  in english, then
↪  improve your code to better model the true distribution.

Please implement the code for the following the template.
You must implement the ‘ initial_func ‘ function.

The code needs to be directly runnable an empty state with the walls of the grid
↪  pre-filled and return a sample initial state.

Do not output examples.
Do not create any new classes.
Do not rewrite existing classes.
Do not import any new modules from anywhere.
Do not list out specific indices that overfit to the examples, but include ranges.
Put the ‘ initial_func ‘ function in a python code block.
```

G.6 Iteration 2 Output

````
#define system
You are a robot exploring its environment.

Unlock door with key to reach the goal square

Your goal is to model the distribution of initial states  of the world in python.

You have tried it before and came up with one partially correct solution, but it
↪  is not perfect.

The observed distribution disagrees with the generated model in several cases.
You need to improve your code to come closer to the true distribution.

Environment Description: Unlock door with key to reach the goal square
Goal Description:

Here is a solution you came up with before.

```
{code_api}

def initial_func(empty_state: MinigridState):
    """
````
````

````
    Input:
        empty_state (MinigridState): An empty state with only the walls filled
        ↪  into the grid
    Returns:
        state (MinigridState): the initial state of the environment
    """

    # Create the grid with walls, locked door, key, and goal
    grid = np.array([
        [2, 2, 2, 2, 2, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 2, 2, 6, 2, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 10, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 2, 2, 2, 2, 2],
    ], dtype=np.int8)

    # Generate possible positions for the agent and key
    possible_positions = [(x, y) for x in range(1, 5) for y in range(1, 5)]
    possible_positions.remove((4, 2))  # Remove fixed key position for example
    ↪  compatibility

    # Randomly choose the agent's position
    agent_pos = possible_positions[np.random.choice(len(possible_positions))]

    # If the agent's position overlaps with the key's recognizable positions,
    ↪  choose again for the key
    key_position_options = [pos for pos in possible_positions if pos != agent_pos]
    key_position =
    ↪  key_position_options[np.random.choice(len(key_position_options))]
    grid[key_position] = ObjectTypes.key

    # Random choice for the agent's initial direction (facing one of four possible
    ↪  directions)
    agent_dir = np.random.choice([0, 1, 2, 3])

    # Return the initial state with the agent randomly positioned on the grid,
    ↪  initially carrying nothing
    return MinigridState(
        grid=grid,
        agent_pos=agent_pos,
        agent_dir=agent_dir,
        carrying=None
    )

```


Here are some samples from the real world that were impossible under your model
Output MinigridState:
agent_pos=(1, 1)
agent_dir=2
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  7,  1,  1,  2, ],
````
````

```
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



And here are some samples from your code under the same conditions
Output MinigridState:
agent_pos=(2, 2)
agent_dir=1
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  7,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(4, 1)
agent_dir=2
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  7,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(3, 2)
agent_dir=1
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  7,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
```

```
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(2, 1)
agent_dir=1
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  7,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]



Output MinigridState:
agent_pos=(4, 4)
agent_dir=0
carrying=None
grid=[
[ 2,  2,  2,  2,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  7,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  6,  2,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  1,  1, 10,  1,  2, ],
[ 2,  1,  1,  1,  1,  2, ],
[ 2,  2,  2,  2,  2,  2, ],
]






Explain what you believe is the distribution of initial states  in english, then
↪  improve your code to better model the true distribution.

Please implement the code for the following the template.
You must implement the ‘ initial_func ‘ function.

The code needs to be directly runnable an empty state with the walls of the grid
↪  pre-filled and return a sample initial state.

Do not output examples.
Do not create any new classes.
Do not rewrite existing classes.
Do not import any new modules from anywhere.
```

```
Do not list out specific indices that overfit to the examples, but include ranges.
Put the ‘ initial_func ‘ function in a python code block.
```

G.7 Iteration 2 Output

````
The initial distribution of states in the environment involves a grid with walls
↪  enclosing the playing area and a locked door blocking access to the goal.
↪  Inside the grid, the agent and the key are randomly placed, with the
↪  stipulation that they don't overlap. The agent's position is chosen randomly
↪  from the walkable area within the grid, and its facing direction can be any
↪  one of the four cardinal directions (right, down, left, up). The key's
↪  position is also selected randomly but must be different from the agent's
↪  position.

Here's the improved code to better model this distribution:

```python
def initial_func(empty_state: MinigridState) -> MinigridState:
    """
    Input:
        empty_state (MinigridState): An empty state with only the walls filled
        ↪  into the grid
    Returns:
        state (MinigridState): the initial state of the environment
    """

    # Create the grid with walls, locked door, key, and goal
    grid = np.array([
        [2, 2, 2, 2, 2, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 2, 2, 6, 2, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 1, 1, 10, 1, 2],
        [2, 1, 1, 1, 1, 2],
        [2, 2, 2, 2, 2, 2],
    ], dtype=np.int8)

    # Generate possible positions for the agent and key
    possible_agent_positions = [(x, y) for x in range(1, 5) for y in range(1, 5)]

    # Randomly choose the agent's position
    agent_pos =
    ↪  possible_agent_positions[np.random.choice(len(possible_agent_positions))]

    # Generate possible positions for the key (must not overlap with agent)
    possible_key_positions = [(x, y) for x in range(1, 5) for y in range(1, 5) if
    ↪  (x, y) != agent_pos]

    # Randomly choose the key's position
    key_pos =
    ↪  possible_key_positions[np.random.choice(len(possible_key_positions))]
    grid[key_pos] = ObjectTypes.key

    # Random choice for the agent's initial direction (facing one of four possible
    ↪  directions)
    agent_dir = np.random.choice([0, 1, 2, 3])
````
````

````
    # Return the initial state with the agent randomly positioned on the grid,
    ↪  initially carrying nothing
    return MinigridState(
        grid=grid,
        agent_pos=agent_pos,
        agent_dir=agent_dir,
        carrying=None
    )
```
````
````