Program Synthesis Guided Reinforcement Learning for Partially Observed Environments

Anonymous Author(s)
Affiliation
Address
email

Abstract

A key challenge for reinforcement learning is solving long-horizon planning problems. Recent work has leveraged programs to guide reinforcement learning in these settings. However, these approaches impose a high manual burden on the user since they must provide a guiding program for every new task. We propose a novel approach that uses program synthesis to automatically generate the guiding program from a high-level goal specification. A key challenge is handling partially observable environments. We propose *model predictive program synthesis*, which trains a generative model to predict the unobserved portions of the world, and then synthesizes a program based on samples from this model in a way that is robust to its uncertainty. In our experiments, we show that our approach significantly outperforms prior approaches on a set of challenging benchmarks, including a 2D Minecraft-inspired environment where the agent must complete a complex sequence of subtasks to achieve its goal. Our results demonstrate that our approach can obtain the benefits of program-guided reinforcement learning without requiring the user to provide a new guiding program for each new task.

1 Introduction

Reinforcement learning is a prominent technique for solving challenging planning and control problems [23, 3]. Despite significant recent progress, solving long-horizon problems remains a significant challenge due to the combinatorial explosion of possible strategies. One promising approach to addressing these issues is to leverage *programs* to guide the behavior of the agents [2, 31, 17]. In this paradigm, the user typically provides two pieces of information:

- **Domain-specific language (DSL):** For each domain, a set of *components* c that encode intermediate subgoals that are useful for that domain (e.g., "gather wood" or "build bridge"), but leaving out how exactly to achieve these subgoals.
- **Task-specific program:** A sequence of components that, if followed, enable the agent to achieve its goal in a specific task (e.g., "gather wood"; "build bridge"; "get gem").

For a given domain, the reinforcement learning algorithm learns an option [32] that implements each component (i.e., achieves the subgoal specified by that component). Then, to solve a new task in that domain, the user provides a program in the DSL to solve that task. The agent then deploys the sequence of options that correspond to the components in that program.

A key drawback of this approach is user burden: for every new task (consisting of an instantiation of the environment and a goal), the user needs to analyze the environment, design a strategy to achieve the goal, and encode the strategy into a program. Furthermore, a poorly written program may produce a suboptimal agent. We propose an alternative strategy in which the user only needs to provide a high-level specification for the goal of the task; then, the agent automatically *synthesize* a suitable

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

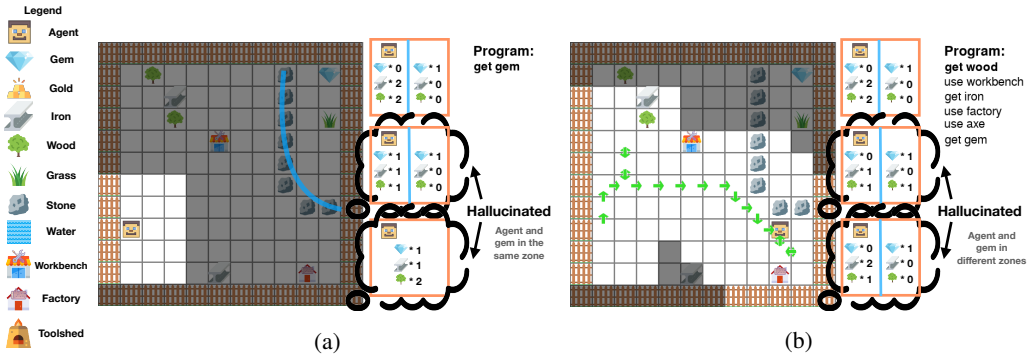


Figure 1: (a) The initial state of an example task for the craft environment. Bright regions are observed and dark ones are unobserved. This particular map has two *zones* separated by a stone boundary (blue line). The first zone contains the agent, 2 irons, and 2 woods; the second contains 1 grass and 1 gem (the goal). The agent represents the high-level structure of the map (e.g., resources in each zone) using state features. The ground truth features are in the top-right; we only show the counts of gems, irons, and woods in each zone and the zone containing the agent. The two thought bubbles below are features hallucinated by the agent based on the observed parts of the map. In both, the zone that the agent is in contains a gem, so the synthesized program is "get gem" (b) The state after the agent took 20 steps (green arrows), failed to obtain the gem, and is now re-synthesizing the program. Having explored more of the map, it predicts that the gem is in a different zone, indicated by its two hallucinations. As a result, it synthesizes a program that includes building and using an axe to break the stone, which leads to successful completion of the task.

program that is used to guide the policy. That is, the agent automatically reasons about how to solve the new task, thereby significantly reducing user burden.

A key challenge is handling partially observable environments. In the fully observed setting, the program synthesis problem reduces to STRIPS planning [7]—i.e., searching over possible plans to find one that achieves the goal. However, these techniques can be difficult to apply in settings where the environment is initially unknown.

To address this challenge, we propose a new approach called *model predictive program synthesis (MPPS)*. MPPS synthesizes the program based on a conditional generative model of the environment, in a way that is robust to the uncertainty in this model. In particular, given a goal specification ϕ, the agent chooses its actions using the following three steps:

- **Hallucinator:** First, inspired by world-models [9], the agent keeps track of a conditional generative model g over possible realizations of the unobserved portions of the environment.
- **Synthesizer:** Next, the agent synthesizes a program p that achieves ϕ assuming the hallucinator g is accurate. Since world predictions are stochastic in nature, it samples multiple predicted worlds and computes the program that maximizes the probability of success.
- **Executor:** Finally, the agent executes the options corresponding to the components in the program $p = c_1; ...; c_k$ for a fixed number of steps N.

If ϕ is not satisfied after N steps, then the above process is repeated. Since the hallucinator now has more information (because the agent has explored more of the environment), the agent now has a better chance of achieving its goal. Importantly, the agent is implicitly encouraged to explore since it must do so to discover whether the current program can successfully achieve the goal ϕ.

We instantiate our approach in the context of a 2D Minecraft-inspired environment [2, 26, 31], which we call the "craft environment", and a "box-world" environment [38]. We demonstrate that our approach significantly outperforms existing approaches for partially observable environments, while achieving a similar performance as using handcrafted programs to guide the agent. In addition, we demonstrate that the policy we learn can be transferred to a continuous variant of the craft environment, where the agent is replaced by a MuJoCo [33] Ant.

Related work. Most closely related is recent work that has demonstrated how programs can be used to guide reinforcement learning in the craft environment [31]. In their approach the user provides a DSL for the domain along with a program for each new task. Furthermore, their approach requires that the user includes conditional statements in the program to handle partial observability, which imposes an even greater burden on the user. In contrast, we only require the user to provide a specification encoding the goal for each new task, and automatically handle partial observability.

There has been work enabling users to write specifications in a high-level language based on temporal logic [17]; then, they translate these specifications into shaped rewards to guide learning. Furthermore, recent work has shown that even if the subgoal encoded by each component is omitted, the program (i.e., just a sequence of symbols) can still aid learning [2]. Unlike our approach, these work still requires the user to provide the guiding programs, and they do not handle partial observability.

More broadly, our work fits into the literature on combining high-level planning with reinforcement learning. In particular, there is a long literature on planning with options [32] (also known as *skills* [12]), including work on inferring options [30]. However, these approaches cannot be applied to MDPs with continuous state and action spaces or to partially observed MDPs. Recent work has addressed the former [1, 18, 11] by combining high-level planning with reinforcement learning to handle low-level control, but not the latter, whereas our work tackles both challenges.

Classical STRIPS planning [7] cannot handle uncertainty in the realization of the environment. Replanning [29] can be used to handle small changes to an initially known environment, but cannot handle environments that are initially completely unknown. There has been work on hierarchical planning in POMDPs [5, 34], but they do not incorporate predicate abstractions (i.e., state features); for instance, these can be used to handle continuous state and action spaces. Generalized planning [8, 14, 28] can be used to compute a plan that is valid in multiple environments. However, in our setting, oftentimes no such plan exists; instead, we need to synthesize a plan that is valid in a maximal number of environments. There is work along these lines for handling partial observability [13], but it does not handle predicate abstractions. We leverage program synthesis [27] with the world models approach [9] to address these issues; generally speaking, our solver-aided plan synthesis approach is more flexible than existing planning algorithms that target more narrow problem settings.

Finally, there has broadly been recent interest in using program synthesis to learn programmatic policies that are more interpretable [37, 16], verifiable [4, 36], and generalizable [15]. In contrast, we are not directly synthesizing the policy, but a program to guide the policy.

2 Motivating Example

Figure 1a shows a 2D Minecraft-inspired crafting game. In this grid world, the agent can navigate and collect resources (e.g., wood), build tools (e.g., a bridge) at workshops using collected resources, and use the tools to achieve subtasks (e.g., use a bridge to cross water). The agent can only observe the 5×5 grid around its current position; since the environment is static, it also memorizes locations it has seen before. A single task consists of a randomly generated map (i.e., the environment) and goal (i.e., obtain a certain resource or build a certain tool).

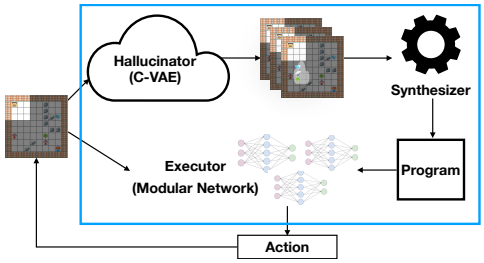
DSL. We provide a DSL (shown in Figure 2a) of components that encode subgoals such as "get wood". For each component, we provide the subgoal as a logical predicate; it describes what the component is expected to achieve. To deal with high-dimensional state spaces, the logical predicates are expressed over features $\alpha(s)$ of the state—e.g., the logical predicate for "get wood" is

$$\forall i, j \,.\, (z^- = i \wedge z^+ = j) \Rightarrow (b^-_{i,j} = \text{connected}) \wedge (\rho^+_{j,\text{wood}} = \rho^-_{j,\text{wood}} - 1) \wedge (\iota^+_{\text{wood}} = \iota^-_{\text{wood}} + 1).$$

This predicate is over two sets of features: (i) features $\alpha(s^-)$, denoted by a $-$, of the initial state s^- (i.e., where execution of the component starts), and (ii) features $\alpha(s^+)$, denoted by a $+$, of the final state s^- (i.e., where the subgoal is achieved and execution of the component terminates). The first feature is the categorical feature z that indicates the zone containing the agent. In particular, we divide the map into zones that are regions separated by obstacles such as water and stone—e.g., the map in Figure 1a has two zones: (i) the region containing the agent, and (ii) the region blocked off by stones. Now, the feature $b_{i,j}$ indicates whether zones i and j are connected, $\rho_{i,r}$ denotes the count of resource r in zone i, and ι_r denotes the count of resource r in the agent's inventory.

$$
\begin{array}{lcl}
C & := & \text{get } R \mid \text{use } T \mid \text{use } W \\
R & := & \text{wood} \mid \text{iron} \mid \text{grass} \mid \text{gold} \mid \text{gem} \\
T & := & \text{bridge} \mid \text{axe} \mid \text{ladder} \\
W & := & \text{factory} \mid \text{workbench} \mid \text{toolshed}
\end{array}
$$

(a)



(b)

Figure 2: (a) DSL of components for the craft environment; the three kinds of components are get resource (R), use tool (T), and use workshop (W). (b) Architecture of our agent (the blue box).

Thus, this formula says that (i) the agent goes from zone i to j, (ii) i and j are connected, (iii) the count of wood in the agent's inventory increases by one, and (iv) the count of wood in zone j decreases by one. All of the components we use are summarized in Appendix A.1.

Approach. Before solving any tasks, for each component c, our algorithm uses reinforcement learning to train an option $\tilde{c}$ that attempts to achieve the subgoal encoded by c. Next, to solve a new task, the user provides a specification ϕ, which is a logical predicate encoding the goal of this task. Then, the agent acts in the environment to try to achieve ϕ. Encoding the goal is typically simple; for example, the goal of the task in Figure 1a is getting gem, which is encoded as $\phi := \iota_{\text{gem}} \geq 1$.

First, based on the observations so far, the agent π uses the hallucinator g to predict multiple potential worlds, each of which represents a possible realization of the full map. Rather than predicting concrete states, it suffices to predict the state features. For instance, Figure 1a shows two samples of the world predicted by g; here, the only values it predicts are the number of zones in the map, the type of the boundary between the zones, and the counts of the resources and workshops in each zone. In this example, the first predicted world contains two zones, and the second contains one zone. Note that in both predicted worlds, there is a gem located in same zone as the agent.

Next, the agent π synthesizes a program p that achieves the goal in the maximum possible number of predicted worlds. The synthesized program in Figure 1a is a single component "get gem", which is an option that searches the current zone (or zones already connected with the current zone) for a gem. Note that this program achieves the goal for the predicted worlds shown in Figure 1a.

Finally, the agent executes the program $p = c_1; ...; c_k$ for a fixed number N of steps. In particular, it executes the policy π_τ of option $\tilde{c}_\tau = (\pi_\tau, \beta_\tau)$ corresponding to c_τ until the termination condition β_τ holds, upon which it switches to executing $\pi_{\tau+1}$. In our example, there is only one component "get gem", so it executes the policy for this component until the agent finds a gem.

In this case, the agent fails to achieve its goal ϕ since there is no gem in its current zone. Thus, it repeats the above process. Since it now has more observations, g more accurately predicts the world—e.g., Figure 1b shows the intermediate step when the agent re-plans. Note that it now correctly predicts that the only gem is in the second zone. Thus, the newly synthesized program is

$$p = \underbrace{\text{get wood; use workbench; get iron; use factory;}}_{\text{for building axe}} \text{ use axe; get gem.}$$

That is, it builds an axe to break the stone so it can get to the zone containing the gem. Finally, the agent executes this new program, which successfully finds the gem.

3 Problem Formulation

POMDP. We consider a partially observed Markov decision process (POMDP) with states $\mathcal{S} \subseteq \mathbb{R}^n$, actions $\mathcal{A} \subseteq \mathbb{R}^m$, observations $\mathcal{O} \subseteq \mathbb{R}^q$, initial state distribution $\mathcal{P}_0$, observation function $h : \mathcal{S} \to \mathcal{O}$, and transition function $f : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. Given initial state $s_0 \sim \mathcal{P}_0$, policy $\pi : \mathcal{O} \to \mathcal{A}$, and time horizon $T \in \mathbb{N}$, the generated trajectory is $(s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$, where $o_t = h(s_t)$, $a_t = \pi(o_t)$, and $s_{t+1} = f(s_t, a_t)$. We assume the state includes the unobserved parts of the environment—e.g., in the craft environment, it represents both the entire map and the agent's current position.

Programs. We consider programs $p = c_1; ...; c_k$ composed of *components* $c_\tau \in C$. Each component c corresponds to an option $\tilde{c} = (\pi, \beta)$, where $\pi : \mathcal{O} \to \mathcal{A}$ is a policy and $\beta : \mathcal{O} \to \{0, 1\}$ is a termination condition. To execute p, the agent uses the options $\tilde{c}_1, ..., \tilde{c}_k$ in sequence; to use $\tilde{c}_\tau = (\pi_\tau, \beta_\tau)$, it takes actions $\pi_\tau(o)$ until $\beta_\tau(o) = 1$, at which point it switches to option $\tilde{c}_{\tau+1}$.

User-provided components. We assume the user provides the components c in terms of their desired behaviors. Importantly, these components only need to be provided once for a domain; they are shared across all tasks in this domain. Each component is a logical predicate that encodes a subgoal. More precisely, c is a logical predicate over s^- and s^+, where s^- denotes the initial state before executing c and s^+ denotes the final state after executing c. For instance, the component

$$c \equiv (s^- = s_0 \Rightarrow s^+ = s_1) \vee (s^- = s_2 \Rightarrow s^+ = s_3)$$

says that if the POMDP is currently in state s_0, then c should transition it to s_1, and if it is currently in state s_2, then c should transition it to s_3. Rather than define c over the concrete states, we can define it over features $\alpha(s^-)$ and $\alpha(s^+)$ of the states.

User-provided goal specification. The goal of each task is specified with a logical predicate ϕ over the final state s, encoding what should be achieved; as with components, ϕ maybe be specified over features $\alpha(s)$ instead of concrete states. Our goal is to design an agent π that can achieve any given specification ϕ (i.e., act in the POMDP to reach a state that satisfies ϕ) as quickly as possible.

4 Model Predictive Program Synthesis

We describe the architecture of our agent, depicted in Figure 2b. It is composed of three parts: the *hallucinator* g, which predicts possible worlds; the *synthesizer*, which generates a program p that succeeds with high probability according to worlds sampled from g; and the *executor*, which uses p to act in the POMDP. These parts are run once every N steps to generate a program p to execute for the subsequent N steps, until the user-provided specification ϕ is achieved.

Hallucinator. First, the hallucinator is a conditional generative model trained to predict the environment given the observation so far. For simplicity, we assume the observation o on the current step already encodes all observations so far. To be precise, the hallucinator g encodes a distribution $g(s \mid o)$, which is trained to approximate the actual distribution $P(s \mid o)$. Then, at each iteration (i.e., once every N steps), our agent samples m worlds $\hat{s}_1, ..., \hat{s}_m \sim g(\cdot \mid o)$. Our technique can work with any type of conditional generative model as the hallucinator; in our experiments, we use a conditional variational auto-encoder (CVAE) [25].

When using state features, we can have g directly predict the features; this approach works since as described below, the synthesizer only needs to know the values of the features to generate a program.

Synthesizer. The synthesizer computes a program that maximizes the probability of satisfying ϕ:

$$p^* = \arg\max_p \mathbb{E}_{P(s|o)}[p \text{ solves } \phi \text{ for } s] \approx \arg\max_p \frac{1}{m} \sum_{j=1}^{m} \mathbb{1}[p \text{ solves } \phi \text{ for } \hat{s}_j], \tag{1}$$

where the $\hat{s}_j$ are samples from g. The objective (1) can be expressed as a MaxSAT problem [22]. In particular, suppose for now that we are searching over programs $p = c_1; ...; c_k$ of fixed length k. Then, consider the constrained optimization problem

$$\arg\max_{\xi_1,...,\xi_k} \frac{1}{m} \sum_{j=1}^{m} \exists s_1^-, s_1^+, ..., s_k^-, s_k^+ \,.\, \psi_j, \tag{2}$$

where ξ_τ and s_τ^δ (for $\tau \in \{1, ..., k\}$ and $\delta \in \{-, +\}$) are the optimization variables. Here, $\xi_1, ..., \xi_k$ encodes the program $p = c_1; ...; c_k$, and ψ_j encodes the event that p solves ϕ for world $\hat{s}_j$—i.e.,

$$\psi_j \equiv \psi_{j,\text{start}} \wedge \left[\bigwedge_{\tau=1}^{k} \psi_{j,\tau}\right] \wedge \left[\bigwedge_{\tau=1}^{k-1} \psi'_{j,\tau}\right] \wedge \psi_{j,\text{end}},$$

where (i) $\psi_{j,\text{start}} \equiv (s_1^- = \hat{s}_j)$ encodes that the initial state is $\hat{s}_j$, (ii) $\psi_{j,\tau} \equiv \big((\xi_\tau = c) \Rightarrow c(s_\tau^-, s_\tau^+)\big)$ encodes that if the the τth component of p is $c_\tau = c$, then the transition from s_τ^- to s_τ^+ on step

τ satisfies $c(s_\tau^-, s_\tau^+)$, (iii) $\psi'_{j,\tau} \equiv (s_\tau^+ = s_{\tau+1}^-)$ encodes that the final state of the τth step equals the initial state the $(\tau+1)$th step, and (iv) $\psi_{j,\text{end}} \equiv \phi(s_j^+)$ encodes that the final state of the last component should satisfy the user-provided goal ϕ. We use a MaxSAT solver to solve (2) [6]. Given a solution $\xi_1 = c_1, ..., \xi_k = c_k$, the synthesizer returns the corresponding program $p = c_1; ...; c_k$.

We incrementally search for longer and longer programs, starting from $k = 1$ and incrementing k until either we find a program that achieves at least a minimum objective value, or we reach a maximum program length k_{max}, at which point we use the best program found so far.

Executor. The executor runs the synthesized program $p = c_1; ...; c_k$ for $t \in \{1, ..., N\}$ steps. It uses each component $c_\tau = (\pi_\tau, \beta_\tau)$, starting from $\tau = 1$. In particular, it uses action $a_t = \pi_\tau(o_t)$ at each time step t, where o_t is the observation on that step, until $\beta_\tau(o_t) = 1$, at which point it increments $\tau \leftarrow \tau + 1$. It continues until either it has completed running the program ($\beta_k(o_t) = 1$), or after N time steps. In the former case, by construction, the goal ϕ has been achieved, so the agent terminates. In the latter case, the agent iteratively reruns the hallucinator and the synthesizer based on the current observation to obtain a new program. At this point, the hallucinator likely has additional information about the environment, so the new program has a greater chance of achieving ϕ.

5 Learning Algorithm

Next, we describe our algorithm for learning the parameters of models used by our agent. In particular, there are two parts that need to be learned: (i) we need to learn parameters of the hallucinator g, and (ii) we need to learn the options $\tilde{c}$ based on the user-provided components c.

Hallucinator. The goal is to train the hallucinator $g(s \mid o)$ to approximate the actual distribution $P(s \mid o)$ of states s given the current observation o. First, we obtain samples (o_t, s_t) using rollouts collected using a random agent; then, we train $g_\theta(s \mid o)$ using supervised learning. In our experiments, we take g_θ to be a CVAE and train using the evidence lower bound (ELBo) on the log likelihood [20].

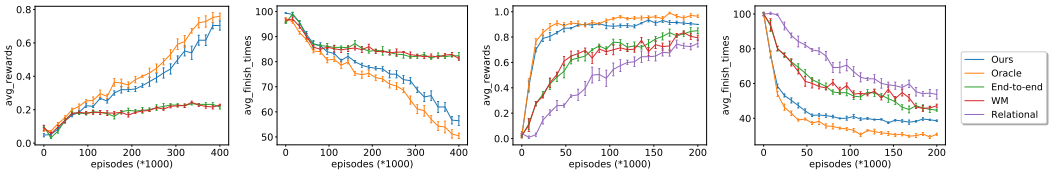
Executor. Our framework uses reinforcement learning to learn options $\tilde{c}$ that implement the user-provided components c; these options can be shared across multiple tasks. We use neural module networks [2] as the executor model. In particular, we take $\tilde{c} = (\pi, \beta)$, where $\pi : \mathcal{O} \to \mathcal{A}$ is a neural module and $\beta : \mathcal{O} \to \{0, 1\}$ checks when to terminate execution. First, β is constructed directly from c—i.e., it returns whether c is satisfied based on the current observation o. Next, to train π, we generate random initial states s and goal specifications ϕ. Just for training, we use the ground truth program p synthesized based on the fully observed environment (since we can explore the entire map and post-hoc generate p); this approach avoids the need to run the synthesizer repeatedly during training. Given p, we sample a rollout $\{((o_1, a_1, r_1), ..., (o_T, a_T, r_T))\}$ by using the executor with p and the current options $c_\tau = (\pi_\tau, \beta_\tau)$ (where π_τ is randomly initialized). We give the agent a reward $\tilde{r}$ at each time step when it achieves the subgoal of the component c_τ. Then, we use actor-critic reinforcement learning [21] to update π. Finally, we use curriculum learning to speed up training—i.e., we train using tasks that can be solved with shorter programs first [2].

6 Experiments

We empirically show that our approach significantly outperforms prior approaches that do not leverage programs, and furthermore achieves similar performance as an oracle given the ground truth program.

6.1 Benchmarks

2D-craft. We consider a 2D Minecraft-inspired game [2] (Figure 1a). A map is a 10×10 grid, where each grid cell is either empty or contains a resource (e.g., wood), obstacle (e.g., water), or workshop. Each task consists of a randomly sampled map, initial position, and goal (one of 10 possibilities, either getting a resource or building a tool), which typically require agent to achieve several intermediate subgoals. In contrast to prior work, our agent does not initially observe the entire map; instead, they can only observe cells within two units. Since the environment is static, any previously visited cells remain visible. The actions are discrete: moving in one of the four directions, picking up a resource, using a workshop, or using a tool. The maximum episode length is $T = 100$.



(a) (b) (c) (d)

Figure 3: (a,b) Training curves for 2D-craft environment. (c,d) Training curves for the box-world environment. (a,c) The average reward on the test set over the course of training; the agent gets a reward of 1 if it successfully finishes the task in the time horizon, and 0 otherwise. (b,d) The average number of steps taken to complete the task on the test set. We run all the training with 5 different random seeds, and report the mean and standard error of each metric. We show our approach ("Ours"), the program guided agent ("Oracle"), the end-to-end neural policy ("End-to-end"), world models ("WM"), and relational reinforcement learning ("Relational").

	2D-craft		**Box-world**		**Ant-craft**	
	Reward	**Finish step**	**Reward**	**Finish step**	**Reward**	**Finish step**
End-to-end	0.22 (0.01)	82.3 (1.3)	0.85 (0.02)	44.7 (0.6)	0.12 (0.03)	93.1 (2.2)
World models [9]	0.23 (0.01)	81.2 (0.7)	0.80 (0.02)	47.2 (0.9)	0.13 (0.01)	91.3 (1.2)
Relational [38]	-	-	0.75 (0.03)	53.7 (2.7)	-	-
Ours	0.70 (0.03)	56.4 (2.0)	0.90 (0.00)	38.6 (0.4)	0.40 (0.01)	79.2 (1.7)
Oracle	0.76 (0.02)	50.4 (1.1)	0.97 (0.01)	30.8 (0.5)	0.43 (0.02)	77.2 (1.6)

Table 1: Average rewards and average completion times on the test set for each approach at the end of training. We report the mean and standard error (in parentheses) over 5 random seeds for training.

Box-world. Next, we consider box-world [38], which requires abstract reasoning. It is a 12×12 grid world with locks and boxes (Figure 5a). The agent is given a key to get started, and its goal is to unlock a white box. Each lock locks a box in the adjacent cell containing a key. Lock and boxes are colored; the key needed to open a lock is in the box of the same color. The actions are to move in one of the four directions; the agent opens a lock and obtains the key simply by walking over it. We assume that the agent can unlock multiple locks with each key. The agent can only observe grid cells within a distance of 3 (as well as the previously observed cells). Each task consists of a randomly sampled map and initial position, where the number of boxes in the path to the goal is randomly chosen between 1 to 4, and the number of "distractor branches" (i.e., boxes that the agent can open but does not help them reach the goal) is also randomly chosen between 1 to 4.

Ant-craft. Finally, to demonstrate that our approach can handle continuous control tasks, we consider a variant of 2D-craft where the agent is replaced by a MuJoCo ant [24] (Figure 5b). For simplicity, the ant automatically picks up resources in the grid cell it currently occupies.

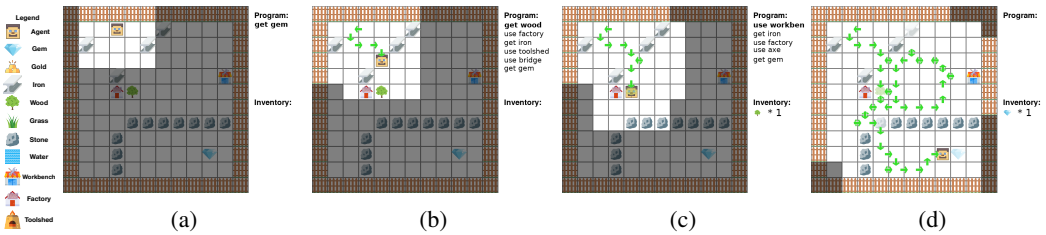
6.2 Baselines

End-to-end. A set of DNN policies trained using the same actor-critic algorithm and curriculum learning strategy as described in Section 5; it uses one DNN policy per goal.

World models [9]. This approach handles partial observability by using a generative model to predict the future. It trains a VAE model that encodes the current observation o_t into a latent vector z_t, and trains a recurrent model to predict z_{t+1} based on $z_1, ..., z_t$. Then, it trains a policy using the latent vectors from the VAE model and the recurrent model as inputs.

Relational reinforcement learning [38]. For box-world, we also compare with this approach, which uses a relational module based on the multi-head attention mechanism [35] for the policy network to facilitate relational reasoning.

Oracle. Finally, we compare to an oracle similar to program-guided agents [31], which is our approach but given the ground truth program (i.e., guaranteed to achieve ϕ). This baseline is an oracle since it strictly requires significantly more information as input from the user.



(a) (b) (c) (d)

Figure 4: Example behavior of our policy in a task with the goal of getting gem. (a) The start state. The agent initially hallucinates that there is a gem in the same zone, thus starts with a simple program "get gem". (b) After several steps, the agent observes a wood and a factory. Hallucinating based on these new observations, the agent synthesizes a new program that builds a bridge to cross some water and get gem. This is a reasonable guess since wood, iron and factory are part of the recipe to build a bridge, therefore the presence of them hints that the solution might be via building a bridge. (c) After the agent finishes the "get wood" component, it observes that there are stones in the map, for which bridge cannot be used. Hallucinating based on these new observations, the agent synthesizes a new program that builds an axe to cross the stone. This is a correct program for this task. (d) The final state. The agent executes the program and successfully gets the gem.

6.3 Implementation Details

2D-craft environment. For our approach, we use a CVAE hallucinator, with MLP (with 200 hidden units) encoder/decoder, trained on 20K (s, o) pairs collected by a random agent. We use the Z3 [6] solver to solve the MaxSAT problems. We use $m = 3$ hallucinated environments, and $N = 20$ steps before replanning for our main experiments. We use the same actor (resp., critic) network architecture across all approaches—i.e., an MLP with 128 (resp., 32) hidden units. We train each approach on 400K episodes over randomly sampled training tasks, and evaluate on a test set containing 50 tasks.

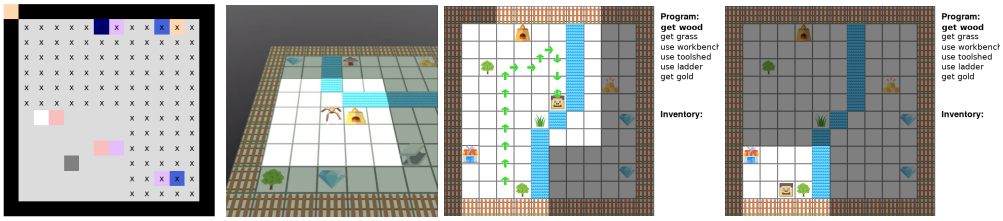
Box-world. Following [38], we use a one-layer CNN with 32 kernels of size 3×3 to preprocess the map across all approaches. For our approach, we have a component for each color where the subgoal is to get the key of that color; see Appendix A.2 for details. For the hallucinator, we use the same architecture as in the craft environment but with 300 hidden units, and trained with 100K (s, o) pairs. For the synthesizer, we use $m = 3$ and $N = 10$. We train each model for 200K episodes, and evaluate on a test set containing 40 tasks.

Ant-craft. We focus on transfer learning from the 2D-craft environment to this one. In particular, we pretrain a goal-reaching policy for the ant using soft actor-critic [10]: given a random goal position, this policy moves ant to that position. The actions output by each approach is translated into a goal position used as input to this goal-reaching policy. We initialize each policy with the corresponding model for 2D-craft, and fine-tune it on ant-craft for 40K episodes.

6.4 Results

Table 1 shows the performance of each approach at the end of training, including results for ant-craft after fine-tuning. Figure 3 shows the training curves for 2D-craft and box-world. Our approach significantly outperforms the non-program-guided baselines, both in terms of fraction of tasks solved and time taken to solve them; it also converges faster, demonstrating that program guidance makes learning significantly more tractable. Our approach also performs comparably to the oracle; thus, it achieves comparable performance with significantly less user guidance. Figure 4 shows the behavior of our policy in an example task in the 2D-craft environment; see Appendix C for more examples.

Effect of the learned hallucinator. Next, we study the benefit of the learned hallucinator to our approach. We compare to two ablations without a learned hallucinator: (i) an *optimistic* synthesizer that synthesizes the shortest possible program making best-case assumptions about the unobserved parts of the map, and (ii) a *random* hallucinator that randomly samples completions of the world. Table 6 shows the results on the 2D-craft environment. As can be seen, our approach significantly outperforms both alternatives. Figure 5c & 5d shows the difference in behavior between our approach and the optimistic strategy; by using a learned hallucinator, our approach is able to leverage the current observations effectively and synthesize a correct program sooner.

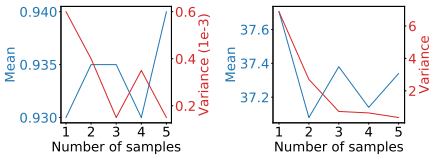


(a) (b) (c) (d)

Figure 5: (a) The box-world environment. The grey pixel denotes the agent. The goal is to get the white key. The unobserved parts of the map is marked with "x". The key currently held by the agent is shown in the top-left corner. In this map, the number of boxes in the path to the goal is 4, and it contains 1 distractor branch. (b) The ant-craft environment. The policy needs to control the ant to perform the crafting tasks. (c,d) Comparison of behaviors between the optimistic approach (left) and our MPPS approach (right), in a task where the goal is to get gold. (c) The state when the optimistic approach first synthesizes the correct program instead of the (incorrect) one "get gold". It only does so after observing all the squares in its current zone. (d) The initial state of our MPPS strategy. It directly synthesizes the correct program, since the hallucinator knows the gold is most likely in the other zone based on the observations. Thus, the agent completes the task much more quickly.

	Avg. reward	Avg. finish step
Ours	**0.70 (0.03)**	**56.4 (2.0)**
Optimistic	0.42 (0.02)	70.2 (1.2)
Random	0.48 (0.02)	72.6 (0.9)

Figure 6: Comparison to optimistic synthesis and random hallucination strategies on the 2D-craft environment.



(a) Reward (b) Finish time

Figure 7: Effect of varying the number of samples m on our approach, evaluated on the box-world over 5 random seeds. Mean and variance of (a) the average reward, (b) the average finishing time on the test tasks.

Effect of the number of hallucinator samples. We vary the number of hallucinator samples m on box-world. Figure 7 shows the results on the test set over 5 random seeds. As can be seen, varying m does not significantly affect the mean performance, but increasing m significantly reduces variance. Thus, increasing m makes the policy more robust to the uncertainty in the hallucinator. This shows the benefit of using multiple samples and MaxSAT synthesis.

7 Conclusion

We have proposed an approach that automatically synthesizes programs that are used to guide reinforcement learning for complex long-horizon tasks. Our algorithm, called model predictive program synthesis (MPPS), handles partially observed environments by leveraging an approach inspired by world models, where it learns a generative model over the remainder of the world conditioned on the observations so far, and then synthesizes a guiding program that accounts for the uncertainty in this model. Our experiments demonstrate that our approach significantly outperforms non-program-guided approaches, while performing comparably to an oracle given a ground truth guiding program. Our results demonstrate that our approach can obtain the benefits of program-guided reinforcement learning without requiring the user to provide a guiding program for every new task.

One limitation of our approach is that, as with existing program guided approaches, the user must provide a set of components for each domain. This process only needs to be completed once for each domain since the components can be reused across tasks; nevertheless, automatically inferring these components is an important direction for future work. Finally, we do not foresee any negative societal impacts or ethical concerns for our work (outside of generic risks in improving robotics capabilities).

References

[1] David Abel, Nate Umbanhowar, Khimya Khetarpal, Dilip Arumugam, Doina Precup, and Michael Littman. Value preserving state-action abstractions. In *International Conference on Artificial Intelligence and Statistics*, pages 1639–1650. PMLR, 2020.

[2] Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 166–175, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL `http://proceedings.mlr.press/v70/andreas17a.html`.

[3] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[4] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. *arXiv preprint arXiv:1805.08328*, 2018.

[5] Laurent Charlin, Pascal Poupart, and Romy Shioda. Automated hierarchy discovery for planning in partially observable environments. *Advances in Neural Information Processing Systems*, 19: 225, 2007.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787992.

[7] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[8] Edward Groshev, Maxwell Goldstein, Aviv Tamar, Siddharth Srivastava, and Pieter Abbeel. Learning generalized reactive policies using deep neural networks, 2018.

[9] David Ha and Jürgen Schmidhuber. World models. *CoRR*, abs/1803.10122, 2018. URL `http://arxiv.org/abs/1803.10122`.

[10] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL `http://proceedings.mlr.press/v80/haarnoja18b.html`.

[11] Mohammadhosein Hasanbeig, Natasha Yogananda Jeppu, Alessandro Abate, Tom Melham, and Daniel Kroening. Deepsynth: Program synthesis for automatic task segmentation in deep reinforcement learning. *CoRR*, abs/1911.10244, 2019. URL `http://arxiv.org/abs/1911.10244`.

[12] Karol Hausman, Jost Tobias Springenberg, Ziyu Wang, Nicolas Heess, and Martin Riedmiller. Learning an embedding space for transferable robot skills. In *International Conference on Learning Representations*, 2018.

[13] Yuxiao Hu and Giuseppe De Giacomo. Generalized planning: Synthesizing plans that work for multiple environments. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, page 918–923. AAAI Press, 2011. ISBN 9781577355144.

[14] León Illanes and Sheila A. McIlraith. Generalized planning via abstraction: Arbitrary numbers of objects. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):7610–7618, Jul. 2019. doi: 10.1609/aaai.v33i01.33017610. URL `https://ojs.aaai.org/index.php/AAAI/article/view/4754`.

[15] Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. Synthesizing programmatic policies that inductively generalize. In *International Conference on Learning Representations*, 2020.

[16] Jeevana Priya Inala, Yichen Yang, James Paulos, Yewen Pu, Osbert Bastani, Vijay Kumar, Martin Rinard, and Armando Solar-Lezama. Neurosymbolic transformers for multi-agent communication. *arXiv preprint arXiv:2101.03238*, 2021.

[17] Kishor Jothimurugan, Rajeev Alur, and Osbert Bastani. A composable specification language for reinforcement learning tasks. In *NeurIPS*, 2019.

[18] Kishor Jothimurugan, Osbert Bastani, and Rajeev Alur. Abstract value iteration for hierarchical reinforcement learning. In *AISTATS*, 2021.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[20] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[21] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12, pages 1008–1014. MIT Press, 2000. URL `https://proceedings.neurips.cc/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf`.

[22] M W Krentel. The complexity of optimization problems. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, page 69–76, New York, NY, USA, 1986. Association for Computing Machinery. ISBN 0897911938. doi: 10.1145/12130.12138. URL `https://doi.org/10.1145/12130.12138`.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[24] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[25] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. Learning structured output representation using deep conditional generative models. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28, pages 3483–3491. Curran Associates, Inc., 2015. URL `https://proceedings.neurips.cc/paper/2015/file/8d55a249e6baa5c06772297520da2051-Paper.pdf`.

[26] Sungryull Sohn, Junhyuk Oh, and Honglak Lee. Hierarchical reinforcement learning for zero-shot generalization with subtask dependencies. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 7156–7166, Red Hook, NY, USA, 2018. Curran Associates Inc.

[27] Armando Solar-Lezama. *Program synthesis by sketching*. Citeseer, 2008.

[28] Siddharth Srivastava. Foundations and applications of generalized planning. *AI Commun.*, 24 (4):349–351, December 2011. ISSN 0921-7126.

[29] Anthony Stentz et al. The focussed dˆ* algorithm for real-time replanning. In *IJCAI*, volume 95, pages 1652–1659, 1995.

[30] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pages 212–223. Springer, 2002.

[31] Shao-Hua Sun, Te-Lin Wu, and Joseph J. Lim. Program guided agent. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=BkxUvnEYDH`.

[32] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2): 181–211, 1999.

[33] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109.

[34] Marc Toussaint, Laurent Charlin, and Pascal Poupart. Hierarchical pomdp controller optimization by likelihood maximization. In *UAI*, volume 24, pages 562–570, 2008.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[36] Abhinav Verma. Verifiable and interpretable reinforcement learning through program synthesis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 9902–9903, 2019.

[37] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, pages 5045–5054. PMLR, 2018.

[38] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, Murray Shanahan, Victoria Langston, Razvan Pascanu, Matthew Botvinick, Oriol Vinyals, and Peter Battaglia. Deep reinforcement learning with relational inductive biases. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=HkxaFoC9KQ`.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See Section 7
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Section 7
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] In supplemental materials
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Section 6
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Appendix B
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]