Common Benchmarks Undervalue the Generalization Power of Programmatic Policies

Amirhossein Rajabpour[1] **Kiarash Aghakasiri**[1] **Sandra Zilles**[2] **Levi H. S. Lelis**[1]
[1]Amii, Department of Computing Science, University of Alberta
[2]Department of Computer Science, University of Regina

Abstract

Algorithms for learning programmatic representations for sequential decision-making problems are often evaluated on out-of-distribution (OOD) problems, with the common conclusion that programmatic policies generalize better than neural policies on OOD problems. In this position paper, we argue that commonly used benchmarks undervalue the generalization capabilities of programmatic representations. We analyze the experiments of four papers from the literature and show that neural policies, which were shown not to generalize, can generalize as effectively as programmatic policies on OOD problems. This is achieved with simple changes in the neural policies training pipeline. Namely, we show that simpler neural architectures with the same type of sparse observation used with programmatic policies can help attain OOD generalization. Another modification we have shown to be effective is the use of reward functions that allow for safer policies (e.g., agents that drive slowly can generalize better). Also, we argue for creating benchmark problems highlighting concepts needed for OOD generalization that may challenge neural policies but align with programmatic representations, such as tasks requiring algorithmic constructs like stacks.

1 Introduction

Deep reinforcement learning (RL) has led to remarkable successes in domains ranging from games to robotics, largely by representing policies as highly parametrized neural networks and optimizing them end-to-end [Lillicrap *et al.*, 2019; Schulman *et al.*, 2017]. However, neural policies often struggle to generalize outside the distribution of their training environments, exhibiting brittle behavior when confronted with out-of-distribution (OOD) scenarios. In contrast, a growing literature on *programmatic* policies, where decision–making rules are expressed in a domain-specific language, claims superior OOD generalization [Verma *et al.*, 2018, 2019; Trivedi *et al.*, 2021; Inala *et al.*, 2020].

We argue that commonly used benchmarks undervalue the generalization power of programmatic representations. Previous work on programmatic policies has observed a substantial gap in terms of OOD generalization between programmatic and neural representations. We revisit these OOD generalization claims and show that, in some cases, the apparent gap between programmatic and neural representations arises not from an inherent limitation of neural representations but from variables we failed to control in evaluating OOD generalization with neural models. Namely, the input observation of the neural agent must be as sparse as the observation the programmatic agent considers. Sparse observations automatically remove distractions that can improve OOD generalization, especially when used with simpler models that are easier to train, such as fully connected networks. Moreover, neural policies tend to be more sensitive to the reward function because they tend to optimize it better than programmatic ones. As a result, a policy that is "too specialized" in one setting might perform poorly in OOD problems. As we show in our experiments, simple changes to the reward function can dramatically enhance OOD generalization.

RLC 2025 Workshop on Programmatic Reinforcement Learning.

We demonstrate how some of these ideas improve the OOD generalization on benchmark problems commonly used in the literature, including a car racing environment (TORCS) [Verma *et al.*, 2018, 2019], grid-world planning problems (KAREL) [Trivedi *et al.*, 2021], and continuous control with repetitive behavior (PARKING) [Inala *et al.*, 2020]. Given our observation that neural policies can generalize to OOD problems in these benchmarks, we suggest creating problems that showcase the OOD generalization of programmatic representations by requiring learning structures that neural networks fail to master, such as stacks [Joulin and Mikolov, 2015]. As an illustrative example, we suggest a problem that requires the agent to use memory through a stack or a queue to solve.

Focusing on benchmark problems that require features beyond the reach of neural models will help us better understand where programmatic representations are most needed. This understanding can help us develop novel representations that combine the flexibility of highly parameterized models with the desired properties of symbolic programs, such as sparsity and the usage of complex data structures.

The code used to run our experiments is publicly available online.

2 Problem Definition

We consider sequential decision-making problems as Markov decision processes (MDPs) $\mathcal{M} = (S, A, p, r, \mu, \gamma)$. Here, S and A are the sets of states and actions. The function $p : S \times A \to S$ is the transition model, which returns the state s_{t+1} reached once the agent takes action a_t in state s_t at time step t. The agent observes a reward value of $R_{t+1} = r(s_t, a_t)$ when transitioning to s_{t+1}; such values are given by the reward function $r : S \times A \to \mathbb{R}$. The MDP's initial states are determined by the distribution μ, with states sampled from μ denoted as s_0. Finally, $\gamma \in [0, 1]$ is the discount factor. A policy $\pi : S \times A \to [0, 1]$ receives a state s and action a and returns the probability of taking a at s. Given a class of policies Π, the goal is to find a policy π within Π that maximizes the return:

$$\arg\max_{\pi \in \Pi} \mathbb{E}_{\pi,p,\mu}[\sum_{k=0}^{\infty} \gamma^k R_{k+1}] \tag{1}$$

The class Π determines the biases of the policies we consider. For example, Π could be an architecture of a neural network, and the policies π within this class are the different weights we can assign to the connections of the neural network. We consider classes Π determined by a domain-specific language, so programs written in the language form Π. A language is defined with a context-free grammar $(\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{I})$, where $\mathcal{N}$, $\mathcal{T}$, $\mathcal{R}$, $\mathcal{I}$ are the sets of non-terminals, terminals, the production rules, and the grammar's initial symbol, respectively. Figure 1 (a) shows an example of a context-free grammar encoding a language for TORCS policies. The grammar's initial symbol $\mathcal{I}$ is E. It accepts strings such as the one shown in Figure 1 (b), which is obtained through a sequence of production rules applied to the initial symbol: $E \to \textbf{if } B \textbf{ then } E \textbf{ else } E \to \textbf{if } B \textbf{ and } B \textbf{ then } E \textbf{ else } E \to \cdots$.

We empirically compare solutions to Equation 1 when the class Π is defined with pre-defined neural network architectures and domain-specific languages. We call the former neural and the latter programmatic policies. We consider the following problem domains in our experiments: TORCS [Verma *et al.*, 2018, 2019], KAREL [Trivedi *et al.*, 2021], and PARKING [Inala *et al.*, 2020].

3 Background: Searching for Programmatic Policies

This section describes the algorithms used to synthesize programmatic policies for solving TORCS (Section 3.1), KAREL (Section 3.2), and PARKING (Section 3.3). We aim to provide enough information so the reader understands our results in Section 4. We do not intend to detail the original algorithms. For full method descriptions, see the cited papers in each subsection.

3.1 Neurally Directed Program Search (NDPS)

Verma *et al.* [2018] introduced Neurally Directed Program Search (NDPS), a method that uses imitation learning through the DAGGER algorithm [Ross *et al.*, 2011] to learn programmatic policies. Figure 1 (a) shows the domain-specific language Verma *et al.* [2018] considered in their experiments on the TORCS benchmark. The **peek** function reads the value of a sensor. For example, $\textbf{peek}(h_{\texttt{RPM}}, -1)$ reads the latest value (denoted by the parameter -1) of the rotation-per-minute

(a) Domain-Specific Language

$$
\begin{array}{rcl}
P & ::= & \mathbf{peek}((\epsilon - h_i), -1) \\
I & ::= & \mathbf{fold}(+, \epsilon - h_i) \\
D & ::= & \mathbf{peek}(h_i, -2) - \mathbf{peek}(h_i, -1) \\
C & ::= & c_1 * P + c_2 * I + c_3 * D \\
B & ::= & c_0 + c_1 * \mathbf{peek}(h_1, -1) + \ldots \\
 & & \cdots + c_k * \mathbf{peek}(h_m, -1) > 0 \mid \\
 & & B \textbf{ or } B \mid B \textbf{ and } B \\
E & ::= & C \mid \textbf{if } B \textbf{ then } E \textbf{ else } E.
\end{array}
$$

(b) Example Policy

$$
\begin{array}{l}
\textbf{if } (0.001 - \mathbf{peek}(h_{\texttt{TrackPOS}}, -1) > 0) \\
\quad \textbf{and } (0.001 + \mathbf{peek}(h_{\texttt{TrackPOS}}, -1) > 0) \\
\quad\quad \textbf{then } 3.97 * \mathbf{peek}((0.44 - h_{\texttt{RPM}}), -1) \\
\quad\quad\quad\quad +0.01 * \mathbf{fold}(+, (0.44 - h_{\texttt{RPM}})) \\
\quad\quad\quad\quad +48.79 * (\mathbf{peek}(h_{\texttt{RPM}}, -2) - \mathbf{peek}(h_{\texttt{RPM}}, -1)) \\
\quad\quad \textbf{else } \ 3.97 * \mathbf{peek}((0.40 - h_{\texttt{RPM}}), -1) \\
\quad\quad\quad\quad +0.01 * \mathbf{fold}(+, (0.40 - h_{\texttt{RPM}})) \\
\quad\quad\quad\quad +48.79 * (\mathbf{peek}(h_{\texttt{RPM}}, -2) - \mathbf{peek}(h_{\texttt{RPM}}, -1))
\end{array}
$$

Figure 1: (a) Context-free grammar specifying a domain-specific language for TORCS, a racing car domain [Verma *et al.*, 2018]. The initial symbol of the language is E, ϵ is a pre-defined constant, and $\{h_i\}_{i=1}^{m}$ is a set of m sensors from which the agent can read. The grammar allows programs that switch between different PID controllers. (b) Example of a policy written in the language.

sensor ($h_{\texttt{RPM}}$); $\mathbf{peek}(h_{\texttt{RPM}}, -2)$ would read the second latest value of the sensor. The $\mathbf{fold}(+, \epsilon - h_i)$ operation adds the difference $\epsilon - h_i$ for a fixed number of steps of the past readings of sensor h_i.

The non-terminal symbols P, I, and D in Figure 1 (a) form the operations needed to learn PID controllers, with programs that switch between different PID controllers, as shown in Figure 1 (b).

NDPS uses a neural policy as an oracle to guide the NDPS's synthesis. Given a set of state-action pairs H, where the actions are given by the neural oracle, NDPS evaluates a program ρ by computing the action agreement of ρ with the actions in H. NDPS runs a brute force search algorithm [Albarghouthi *et al.*, 2013; Udupa *et al.*, 2013], to generate a set of candidate programs C. Then, it learns the parameters of the programs (c_1, c_2, and c_3 in Figure 1) with Bayesian optimization [Snoek *et al.*, 2012] such that the programs mimic H. Once NDPS determines the parameters of programs C, it selects the candidate c in C that maximizes the agent's return; c is the starting point of a local search that optimizes a mixture of the action agreement function and the agent's return.

Verma *et al.* [2019] introduced Imitation-Projected Programmatic Reinforcement Learning (PROPEL), an algorithm that also synthesizes a program for solving control problems. PROPEL is similar to NDPS in that it relies on a neural policy to guide its search through the space of programs. The difference between PROPEL and NDPS is that the neural policy of the former is trained so that it does not become "too different" from what the programmatic learner can express—the inability to represent the teacher's policy is known as the representation gap in the literature [Qiu and Zhu, 2021]. The programmatic policies of both NDPS and PROPEL are called for every state the agent encounters.

3.2 Learning Embeddings for Latent Program Synthesis (LEAPS)

Trivedi *et al.* [2021] introduced Learning Embeddings for Latent Program Synthesis (LEAPS), a system that learns a latent representation of the space of programs a language induces. When given an MDP $\mathcal{M}$, LEAPS searches in the learned latent space for a vector decoded into a program encoding a policy that maximizes the agent's return at $\mathcal{M}$. LEAPS's premise is that searching in the learned latent space is easier than searching in the space of programs, as NDPS and PROPEL do.

Figure 2 (a) shows the context-free grammar specifying the language used to encode policies for KAREL. The language accepts programs with conditionals and loops. It also includes a set of perception functions, such as `frontIsClear`, which verifies whether the cell in front of the agent is clear. Further included are action instructions such as `move` and `turnLeft`. The set of perception functions is important because it defines what the agent can observe. As we show in Section 4.2, having access to less information allows the agent to generalize to OOD problems. Figure 2 (b) shows an example of a KAREL program. Here, the agent will perform two actions, `pickMarker` and `move`, if a marker is present in its current location; otherwise it will not perform any action.

To learn its latent space, LEAPS generates a data set of programs P by sampling a probabilistic version of the context-free grammar defining the domain-specific language. That is, each production of a non-terminal can be selected with a given probability. A program can be sampled from this probabilistic grammar by starting at the initial symbol and randomly applying production rules until

(a) Domain-Specific Language

$\rho :=$ **def** `run m(`s `m)`
$s :=$ **while** `c(`b `c) w(` s `w)` | **if** `c(`b `c) i(`s `i)` |
ifelse `c(`b `c) i(`s `i)` **else** `e(`s `e)` |
repeat `R=`n `r(`s `r)` | $s; s$ | a
$b :=$ h | **not** (h)
$n :=$ $0, 1, \cdots, 19$
$h :=$ `frontIsClear` | `leftIsClear` | `rightIsClear` |
`markersPresent` | `noMarkersPresent`
$a :=$ `move` | `turnLeft` | `turnRight` |
`putMarker` | `pickMarker`

(b) Example Policy

```
def run m(
  if c( markersPresent c) i(
    pickMarker move
  i)
m)
```

Figure 2: (a) Context-free grammar specifying a domain-specific language for KAREL. The programs written in this language accept conditional statements and loops. There is a set of perception functions (h) and functions that return actions (a). (b) Example of a policy for a KAREL task.

we obtain a program with only terminal symbols. This set of programs is used to train a Variational Auto-Encoder (VAE) [Kingma and Welling, 2014], with its usual reconstruction loss. However, in addition to learn spaces that are more friendly to search algorithms, LEAPS uses two additional losses that attempt to capture the semantics of the programs. These two losses incentivize latent vectors that decode into programs with similar agent behavior to be near each other in the latent space. The intuition is that this behavior locality can render optimization landscapes easier to search.

Once the latent space is trained, it is used to solve MDPs. Given an MDP, LEAPS uses the Cross-Entropy Method (CEM) [Mannor *et al.*, 2003] to search for a vector that decodes into a program that maximizes the return. The rollouts of the decoded policies are used to inform the CEM search.

3.3 Programmatic State Machine Policies (PSM)

Inala *et al.* [2020] introduced Programmatic State Machine Policies, which we refer to as PSM, a system that learns a policy as a finite-state machine. A finite state machine policy for an MDP $\mathcal{M}$ is a tuple $(M, S, A, \delta, m_0, F, \alpha)$ where M is a finite set of modes. The sets S and A are the sets of states and actions from $\mathcal{M}$. The function $\delta : M \times S \to M$ is the transition function, m_0 in M is the initial mode, and $F \subseteq S$ is the set of modes in which the policy terminates. The transition function δ defines the next mode given the current mode and input state s in S. Finally, $\alpha : M \times S \to A$ determines the policy's action when in mode m and the agent observes state s.

In the PARKING environment, Inala *et al.* [2020] considered a domain-specific language for the transition function δ and constant values for α. The grammar defining the language δ is the following.

$$B \quad ::= \quad \{s[i] \geq v\}_{i=1}^{n} \mid \{s[i] \leq v\}_{i=1}^{n} \mid B \wedge B \mid B \vee B$$

Here, the values v are constants that need to be learned, $s[i]$ is the i-th entry of the state s the agent observes at a given time step, and n is the dimensionality of the observation.

Figure 3 shows an example of the type of policy PSM learns. In this example, the policy is for PARKING, a domain where the agent must learn how to exit a parking spot with a car in front of the agent's car (car_f) and another at the rear (car_b). The policy uses the following state features: the distance between the agent's car and car_f (d_f) and car_b (d_b), the x coordinate of the car, and the angle θ of the car. A solution involves the agent moving forward to the left (mode m_1) and then back to the right (mode m_2), until the agent has cleared car_f (transitioning to mode m_3).

The agent solves the problem if it straightens the car after clearing car_f, thus transitioning from m_3 to m_f. PSM's policies are called only once for the initial state; the policy returns only at the end of the episode.

PSM learns policies with a teacher-student scheme, where the student is a finite state machine encoding the policy. The teacher is a loop-free learner that finds state-action pair sequences that optimize for two objectives. Specifically, they maximize the agent's return and minimize how much they deviate from the student's policy. Optimizing for the latter avoids sequences that cannot be encoded in a finite-state machine. After optimizing the teacher, the student updates its policy based on the teacher's sequence. The student's policy is updated through a clustering scheme on the teacher's

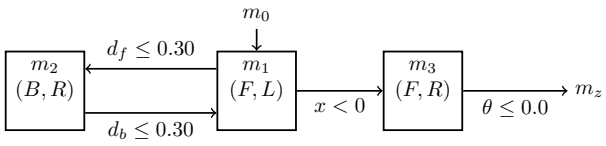

Figure 3: Example of a state machine policy, where m_0 is the initial mode and m_z is an accepting mode. The tuples inside each mode specify the agent's action when in that mode (e.g., (F, L) means "move forward and steer to the left". The transitions from one mode to another are triggered by a Boolean expression shown in the arrows. For example, if the car is too close to the car in front of it ($d_f \leq 0.30$), then the policy moves from m_1 to m_2. The agent remains in the current mode if no outgoing Boolean expression is triggered. This policy is based on an example by Inala *et al.* [2020].

TRACKS	NDPS LAP TIME	DRL ($\beta = 1.0$) LAP TIME	DRL ($\beta = 0.5$) LAP TIME
G-TRACK-1	1:01	54	1:17
G-TRACK-2 (OOD)	1:40	CR 1608M	1:48 (0.76)
E-ROAD (OOD)	1:51	CR 1902M	1:54 (0.69)
AALBORG	2:38	1:49	2:24
ALPINE-2 (OOD)	3:16	CR 1688M	3:13 (1.00)
RUUDSKOGEN (OOD)	3:19	CR 3232M	2:46 (1.00)

Table 1: For DRL ($\beta = 0.5$), we trained 30 models (seeds) for G-TRACK-1 and 15 for AALBORG. Each cell shows the average lap time (mm:ss) over three laps per model, then averaged across models; 13 models learned to complete G-TRACK-1 and four models learned to complete AALBORG. Values in parentheses for DRL ($\beta = 0.5$) show the fraction of seeds that successfully generalized to the test track (out of 13 and 4 for G-TRACK-1 and AALBORG, respectively). For NDPS and DRL ($\beta = 1.0$), we used the data from [Verma *et al.*, 2018], which is over three models. "CR" indicates that all three models crashed, and the number reported is the average distance at which the agent crashed the car.

sequence. The Boolean expressions denoting transitions between modes are found through discrete search. The process is repeated, and the teacher's sequence-based policy is optimized.

4 Experiments

In this section, we revisit the experiments of Verma *et al.* [2018] and Verma *et al.* [2019] on TORCS (Section 4.1 and Appendix A), of Trivedi *et al.* [2021] on KAREL (Section 4.2 and Appendix B), and of Inala *et al.* [2020] on PARKING (Section 4.3 and Appendix C).

4.1 TORCS

Verma *et al.* [2018] and Verma *et al.* [2019] showed that programmatic policies written in the language from Figure 1 generalize better to OOD problems than neural policies in race tracks of the Open Racing Car Simulator (TORCS) [Wymann *et al.*, 2000]. The results of Verma *et al.* [2018] also showed that neural policies better optimize the agent's return than programmatic policies, as the former complete laps more quickly than the latter on the tracks on which they are trained. We hypothesized that the programmatic policies generalize better not because of their representation, but because the car moves more slowly, thus making it easier to generalize to tracks with sharper turns.

We test our hypothesis by training models with two different reward functions: the original function used in previous experiments ($\beta = 1.0$ in Equation 2), which we refer to as "original", and a function that makes the agent more cautious about speeding ($\beta = 0.5$), which we refer to as "cautious".

$$\beta \times V_x \cos(\theta) - |V_x \sin(\theta)| - V_x |d_l| \ . \tag{2}$$

Here, V_x is the speed of the car along the longitudinal axis of the car, θ is the angle between the direction of the car and the direction of the track axis, and d_l is the car's lateral distance from the center of the track. The first term of the reward measures the velocity along the central line of the track, while the second is the velocity moving away from the central line. Maximizing the first term

		StairClimber	Maze	TopOff	FourCorner	Harvester
Leaps†	Small	1.00 (0.00)	1.00 (0.00)	0.81 (0.07)	0.45 (0.40)	0.45 (0.28)
	100×100	1.00 (0.00)	1.00 (0.00)	0.21 (0.03)	0.45 (0.37)	0.00 (0.00)
PPO with ConvNet†	Small	1.00 (0.00)	1.00 (0.00)	0.32 (0.07)	0.29 (0.05)	0.90 (0.10)
	100×100	0.00 (0.00)	0.00 (0.00)	0.01 (0.01)	0.00 (0.00)	0.00 (0.00)
PPO with LSTM†	Small	0.13 (0.29)	1.00 (0.00)	0.63 (0.23)	0.36 (0.44)	0.32 (0.18)
	100×100	0.00 (0.00)	0.04 (0.05)	0.15 (0.12)	0.37 (0.44)	0.02 (0.01)
PPO with a_{t-1}	Small	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)	0.59 (0.05)
	100×100	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)	0.04 (0.00)

Table 2: Generalization results on Karel, where cells show the average return and standard deviation. "PPO with ConvNet" observes the entire state and employs a convolutional network to learn its representation. "PPO with LSTM" uses an LSTM layer for both actor and critic, while "PPO with a_{t-1}" uses a fully connected network with the observation space augmented with the agent's last action. "Small" refers to the problems in which the models were trained, which were of size either 8×8 or 12×12. Rows marked with a † are from Trivedi *et al.* [2021]. The results for PPO with a_{t-1} are over 30 seeds, and each seed is evaluated on 10 different initial states; the results for Leaps and PPO with a ConvNet and with an LSTM are over five seeds and 10 different initial states.

minus the second allows the agent to move fast without deviating from the central line. The last term also contributes to having the agent follow the center of the track. Once we set $\beta = 0.5$, the agent will learn policies where the car moves more slowly, which allows us to test our hypothesis.

Following Verma *et al.* [2018], we use the Deep Deterministic Policy Gradient (DDPG) algorithm [Lillicrap *et al.*, 2019] and TORCS's practice mode, which includes 29 sensors as observation space and the actions of accelerating and steering. We considered two tracks for training the agent: G-Track-1 and Aalborg. The first is considered easier than the second based on the track's number of turns, length, and width. The models trained on G-Track-1 were tested on G-Track-2 and E-Road, while the models trained on Aalborg were tested on Alpine-2 and Ruudskogen.

Table 1 presents the results. NDPS can generalize to the test problems in all three seeds evaluated. DRL with $\beta = 1.0$ does not generalize to the test tracks, with the numbers in the table showing the average distance at which the agent crashes the car in all three seeds. For DRL ($\beta = 0.5$) we trained 30 models (seeds) for G-Track-1 and 15 for Aalborg. Then, we verified that 13 of the 30 models learned how to complete laps of the G-Track-1 track, and 4 of the 15 models learned to complete laps of the Aalborg track; these models were evaluated on the OOD tracks.

The results support our hypothesis that by changing the reward function, we would allow the agent to generalize. On the training tracks, the lap time increases as we reduce β. Most models trained with $\beta = 0.5$ generalize from the G-Track-1 to G-Track-2 (76% of the models) and E-Road (69%) tracks; all models that learned to complete a lap on Aalborg generalized to the other two tracks.

4.2 Karel

Trivedi *et al.* [2021] showed that programs Leaps synthesized in the language shown in Figure 2 (a) generalized better than deep reinforcement learning baselines to problem sizes much larger than those the agent encountered during training. In our experiments, we consider the fully observable version of Karel, where the agent has access to the entire grid, and the partially observable version, where the agent can only perceive the cells around it, as shown by the non-terminal h in Figure 2 (a).

In the partially observable case, the problem cannot, in principle, be solved with fully connected neural networks. Consider the two states shown in Figure 4. In one, the agent is going downstairs; in the other, it is going upstairs. Yet, the observation is the same for both states. Trivedi *et al.* [2021] used LSTMs [Hochreiter and Schmidhuber, 1997] to deal with the partial observability problem. Instead of using LSTMs, which tend to be more complex to train than fully connected networks,

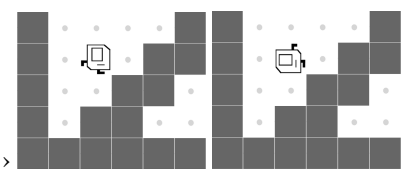

Figure 4: Different states but same observation.

	PSM		DQN	
	Successful-on-100	Success Rate	Successful-on-100	Success Rate
Training	0.06	0.26	0.40	0.86
Test	0.06	0.16	0.00	0.18

Table 3: Evaluation of 30 seeds of PSM and 15 seeds of DQN on the PARKING domain. Each model trained was evaluated on 100 different initial states of both training and testing settings. The columns "Successful-on-100" report the fraction of models trained that successfully solved all 100 initial states. The columns "Success Rate" reports the average number of initial states solved across different seeds.

we add the last action the agent has taken as part of the observation. For the fully observable case, we report the results of Trivedi *et al.* [2021], which used a convolutional network on the input.

We trained policies for the following problems, which were chosen to match the design of Trivedi *et al.* [2021]: STAIRCLIMBER, MAZE, TOPOFF, FOURCORNER, and HARVESTER. The grid size of these problems was either 8×8 or 12×12. After learning to solve these small problems, we evaluated them on grids of size 100×100, also following Trivedi *et al.* [2021]. In the MAZE problem, the agent learns to escape a small maze and is evaluated on a larger one. Table 2 shows the results.

Our results show that partial observability combined with a simpler model can generalize to larger grids. Namely, "PPO with a_{t-1}", which uses a fully connected network with the observation augmented with the agent's last action, generalizes to larger problems. This contrasts with "PPO with ConvNet", which operates in the fully observable setting, and "PPO with LSTM", which operates in the partially observable setting but uses a more complex neural model. To illustrate, in MAZE, if the agent can only see the cells around itself, it can learn strategies such as "follow the right wall", which is challenging to learn in the fully observable setting. The LSTM agent fails not only to generalize to larger problems, but it often also fails to learn how to solve even the smaller problems.

4.3 PARKING

In the PARKING domain, an agent must get out of a parking spot. During training, the distance between the two parked cars is sampled uniformly from the range $[12.0, 13.5]$. In contrast, the test environment uses a narrower and more challenging range of $[11.0, 12.0]$, requiring the agent to generalize to tighter parking scenarios.

We evaluate both programmatic policies, as described by Inala *et al.* [2020], and neural policies trained using Deep Q-Networks (DQN) [Mnih *et al.*, 2015]. Preliminary experiments showed that DQN performed better than the PPO and DDPG algorithms considered in our other experiments. For each policy type, we trained 30 independently seeded models and evaluated each one on 100 test episodes, where the test gap was sampled uniformly from the range $[11.0, 12.0]$.

Table 3 shows the results. We trained 30 independent models of PSM and 15 of DQN. Each model was evaluated on 100 different initial states. The columns "Successful-on-100" refer to the ratio of models that could solve all 100 initial states. For example, 0.06 for PSM means that two of the 30 models solved all initial states on training and test. The "Successful Rate" column shows the ratio of times across all models and initial states that the learned policy could solve the problem. For example, 0.86 for DQN in training means that DQN models solved 86% of the $15 \times 100 = 1500$ initial states.

Our results suggest that the PSM policies generalize better than the DQN policies, as two out of 30 models could solve all 100 test initial states. Looking at the difference between the "Success Rate" of PSM and DQN in training and test also suggests that PSM's policies generalize better, as the gap between the two scenarios is small for PSM: $0.26 - 0.16 = 0.10$ versus $0.86 - 0.18 = 0.68$ for DQN. However, looking at the test "Success Rate" alone suggests that DQN is the winner, as DQN policies can solve more test initial states on average than PSM can. Independent of the metric considered, our results show that PARKING is a challenging domain for both types of representation.

4.4 Discussion

Our experiments showed that neural models can also generalize to OOD problems commonly used in the literature. One key aspect of programmatic solutions is the policy's sparsity. For example, the

mode transitions in Figure 3 use a single variable in the Boolean expression. By contrast, neural networks typically use all variables available while defining such transitions, often by encountering spurious correlations between input features and the agent's action. That is why providing fewer input features, combined with a simpler neural model, helped with generalization in KAREL—we remove features that could generate spurious correlations with the model's actions. These results on reducing input features to enhance generalization align with other studies involving the removal of visual distractions that could hamper generalization [Bertoin *et al.*, 2022; Grooten *et al.*, 2024].

In the case of TORCS, OOD generalization was possible due to a "safer" reward function. If the agent learns on a track that allows it to move fast and never slow down, then it is unlikely to generalize to race tracks with sharp turns that require the agent to slow down. In this case, generalization or lack thereof is not caused by the representation, but by how well the agent can optimize its return while using that representation. We conjecture that NDPS and PROPEL would not generalize to OOD problems if they could find better optimized policies for the agent's return in the training tracks.

PARKING was the most challenging benchmark we considered in our experiments, and we believe it points in the direction of benchmarks that could value the generalization power of programmatic representations. Recurrent neural networks such as LSTMs can, in principle, represent the solution shown in Figure 3. In fact, due to the loop of the agent interacting with the environment, the solution to PARKING does not even require loops. If we augment the agent's observation with its last action, a decision tree could encode the repetitive behavior needed to solve the problem. Yet, we could not find a neural policy that reliably generalizes to OOD problems in this domain. By reliably we mean that if the agent learns how to solve the training setting, it automatically generalizes to the test setting.

4.5 Beyond Generalization

Our analysis has focused on generalizing to OOD problems. However, there are other important dimensions to consider when considering programmatic representations. The most common are interpretability and verifiability [Bastani *et al.*, 2018], as one can choose a language that results in programs that are easier for us to understand and verify. Intuitively, the policies of NDPS, LEAPS, and PSM tend to be more interpretable than neural policies we learned in our experiments.

Another important dimension is sample efficiency. A programming language's inductive bias can make the problem easier to solve. For example, we could add to the language used to define the Boolean expressions of the PSM's policies, an expression that verifies whether the agent is close to an object. Such an expression could be reused and potentially make the approach more sample-efficient. The idea of composing a solution from existing programs underlies library-learning approaches [Ellis *et al.*, 2023; Cao *et al.*, 2023; Bowers *et al.*, 2023; Rahman *et al.*, 2024; Palmarini *et al.*, 2024]. Programmatic solutions tend to be more composable than neural ones, although recent work has investigated the decomposition of reusable pieces of neural networks [Alikhasi and Lelis, 2024].

5 Valuing the Generalization Power of Programmatic Policies

If commonly used problems undervalue the generalization power of programmatic policies, what properties of problems could showcase how programmatic policies can generalize? We propose an illustrative benchmark problem that requires computations that neural networks struggle to learn from data. Although recurrent models are, in theory, computationally universal [Siegelmann and Sontag, 1994, 1995], they are more limited in practice [Weiss *et al.*, 2018]. We consider a problem that requires a stack or a queue, which neural models can struggle with [Joulin and Mikolov, 2015].

We consider finding the shortest paths on a grid. Suppose the agent can only sense the cells around itself, as in the KAREL problem. If the environment is not dense in walls, such that the agent can use simple strategies such as "follow the right wall", it needs to remember the cells it has visited to find the shortest path from its initial location to a goal location. Iterative-Deepening Depth-First Search (IDDFS) uses a stack to solve shortest-path problems. Dijkstra's algorithm [Dijkstra, 1959] could also be used, but it requires that the agent "jumps around" the state space as states far from each other can be expanded from one time step to the next based on the priority of the algorithm's queue.

The maze environment from the KAREL benchmark is similar to the problem we consider, which we call SparseMaze; see Figure 5. What makes the KAREL mazes easier than what we propose is that the agent always has a wall as a reference, favoring strategies such as "follow the right wall" that do not

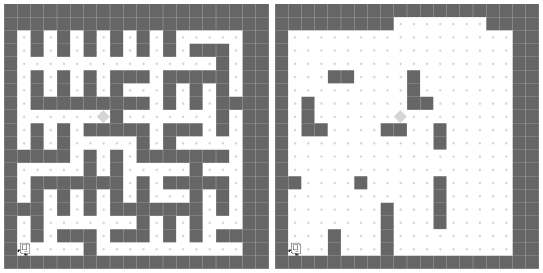

Figure 5: Maze problem from KAREL (left) and our proposed sparse maze (right).

Model		Return (std)
FUNSEARCH	Original	**1.00 (0.00)**
	100×100	**1.00 (0.00)**
PPO with GRU	Original	0.09 (0.29)
	100×100	0.11 (0.32)

Table 4: Results on SPARSEMAZE. The return is an average over 10 initial states. Results for PPO are averaged over 30 seeds; results for FUNSEARCH are a single run of the system.

require memory use. If the map is sparse, as in Figure 5 (right), finding any solution, let alone finding the shortest one, becomes challenging due to the model's inability of learning stacks and queues.

Table 4 presents the generalization results of neural and programmatic policies in SPARSEMAZE. As neural policies, we considered PPO with a GRU [Chung *et al.*, 2014]. As for a programmatic policy, we used FUNSEARCH [Romera-Paredes *et al.*, 2024] with Qwen 2.5-Coder (32B) [Bai *et al.*, 2023] to synthesize Python programs encoding policies to SPARSEMAZE. We use the return of a rollout of the policies as the evaluation function in FUNSEARCH. Appendix D provides the training information of the neural policies and the prompt used in FUNSEARCH. Each approach was trained on maps of size 20×20 ("Original" in Table 4), and evaluated on maps of size 100×100. While PPO could not learn a good policy even for the smaller map, FUNSEARCH synthesized the breadth-first search (BFS) algorithm after 21 iterations of evolution, which generalizes to maps of any size (see Appendix D for FUNSEARCH's policy). Similarly to Dijkstra's algorithm, BFS also uses a queue and thus assumes that the agent can "jump around" the state space. Nevertheless, this proof-of-concept experiment shows an example where programmatic representations can generalize to OOD problems, while neural policies are unlikely to generalize.

6 Conclusion

In this paper, we argued that commonly used benchmarks undervalue the generalization capabilities of programmatic representations. We empirically showed that the OOD generalization gap between programmatic and neural policies on commonly used benchmarks is not as large as we previously thought. We showed that simple neural networks can generalize to OOD problems in KAREL problems when using the sparse observations of programmatic representations. We also showed that, due to neural policies' ability to optimize the agent's return, they might become too specialized in a problem to generalize OOD. This can be fixed in TORCS by using a more cautious reward function that allows the car to move more slowly and thus generalize to unseen and possibly more challenging race tracks. We also evaluated the PARKING problem, where the agent must learn repetitive behaviors. While programmatic policies generalized slightly better than neural policies in this domain, both representations struggled to learn policies that generalize reliably, thus suggesting that more research is needed to understand what is required to attain OOD generalization in this type of problem. Finally, we argued for benchmarks focusing on the weaknesses of neural networks. As an illustrative example, we suggested a benchmark that requires the agent to use data structures that neural networks struggle to learn. By focusing on the weaknesses of neural networks, we will better understand the scenarios in which programmatic representations are needed and, importantly, how programmatic and neural representations can be combined into representations that inherit the best properties of both worlds.

References

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *International Conference Computer Aided Verification, CAV*, pages 934–950, 2013.

Mahdi Alikhasi and Levi Lelis. Unveiling options with neural network decomposition. In *The Twelfth International Conference on Learning Representations*, 2024.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report, 2023.

Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Proceedings of the International Conference on Neural Information Processing Systems*, pages 2499–2509, 2018.

David Bertoin, Adil Zouitine, Mehdi Zouitine, and Emmanuel Rachelson. Look where you look! saliency-guided q-networks for generalization in visual reinforcement learning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proceedings of the ACM on Programming Languages*, 2023.

David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. Babble: Learning better abstractions with e-graphs and anti-unification. *Proceedings of the ACM on Programming Languages*, 2023.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

Kevin Ellis, Lionel Wong, Maxwell Nye, Mathias Sabl-Meyer, Luc Cary, Lore Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua Tenenbaum. Dreamcoder: growing generalizable, interpretable knowledge with wake?sleep bayesian program learning. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 381, 06 2023.

Bram Grooten, Tristan Tomilin, Gautham Vasan, Matthew E. Taylor, A. Rupam Mahmood, Meng Fang, Mykola Pechenizkiy, and Decebal Constantin Mocanu. Madi: Learning to mask distractions for generalization in visual deep reinforcement learning. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, page 733?742. International Foundation for Autonomous Agents and Multiagent Systems, 2024.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

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

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 28, 2015.

Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In *International Conference on Learning Representations*, 2014.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

Shie Mannor, Reuven Y. Rubinstein, and Yohai Gat. The cross entropy method for fast policy search. In *Proceedings of the International Conference on Machine Learning*, pages 512–519, 2003.

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.

Alessandro B. Palmarini, Christopher G. Lucas, and N. Siddharth. Bayesian program learning by decompiling amortized knowledge. In *Proceedings of the International Conference on Machine Learning*, 2024.

Wenjie Qiu and He Zhu. Programmatic reinforcement learning without oracles. In *International Conference on Learning Representations*, 2021.

Habibur Rahman, Thirupathi Reddy Emireddy, Kenneth Tjhia, Elham Parhizkar, and Levi Lelis. Synthesizing libraries of programs with auxiliary functions. *Transactions on Machine Learning Research*, 2024.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 627–635. PMLR, 2011.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Hava T. Siegelmann and Eduardo D. Sontag. Analog computation via neural networks. *Theoretical Computer Science*, 131(2):331–360, 1994.

Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, 1995.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, volume 25, pages 2951–2959, 2012.

Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J. Lim. Learning to synthesize programs as interpretable and generalizable policies. In *Advances in Neural Information Processing Systems*, pages 25146–25163, 2021.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. Transit: Specifying protocols with concolic snippets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–296. ACM, 2013.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, volume 80, pages 5045–5054. PMLR, 2018.

Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. *Advances in Neural Information Processing Systems*, 32, 2019.

Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision rnns for language recognition. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pages 740–745. Association for Computational Linguistics, 2018.

Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. Torcs, the open racing car simulator. *Software available at http://torcs. sourceforge. net*, 4(6):2, 2000.

A TORCS Details

We use the hyperparameters in Table 5 with DDPG [Lillicrap *et al.*, 2019].

Hyperparameter	Selected Value
Actor's learning rate	0.0003
Critic's learning rate	0.001
Batch size	64
Buffer size	100000
τ	0.005
L1 regularization	0.00001
Max steps	20000
Training episodes	600

Table 5: Hyperparameter Configuration Used for TORCS

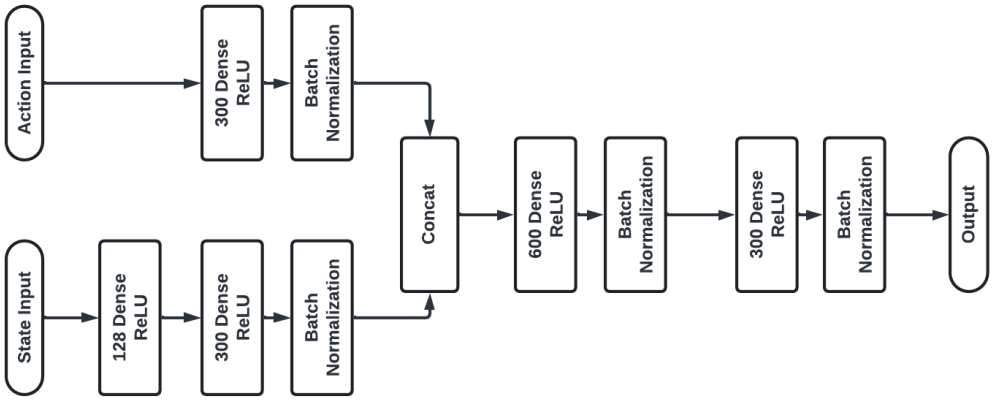

Figure 6: Architecture of the critic network used in DDPG for the TORCS environment.

B KAREL Details

We used Proximal Policy Optimization (PPO) [Schulman *et al.*, 2017] with the agent's previous action appended to the observation vector. A comprehensive hyperparameter sweep was conducted over the values from Table 6.

Hyperparameter	Values Tested
Learning rate	{0.001, 0.0001, 0.00001}
Clipping coefficient	{0.01, 0.1, 0.2}
Entropy coefficient	{0.001, 0.01, 0.1}
L1 regularization	{0.0, 0.0001, 0.0005, 0.001}
Actor's hidden layer size	{32, 64}
Training time steps	{2 million}
Seeds per config	{3}

Table 6: PPO + Last Action: Hyperparameter Sweep Configuration for Karel

The max steps used for training and testing different Karel tasks are shown in Table 7.

	StairClimber	Maze	TopOff	FourCorner	Harvester
Training	50	100	100	100	200
Test	1000	100000	1000	1000	10000

Table 7: Max steps of episodes for each Karel task during training and test.

Table 8 shows the best-performing configuration across all five tasks for final evaluation. The selection was based on the agent's average return across the three seeds after two million time steps of training.

Hyperparameter	**Selected Value**
Learning rate	`0.001`
Clipping coefficient	`0.1`
Entropy coefficient	`0.1`
L1 regularization	`0.0`
Actor's hidden layer size	`32`

Table 8: Best Hyperparameter Configuration Used for Final Training of Karel

The best configuration was then trained with 30 random seeds, and evaluation results were averaged over 10 distinct initial configurations per seed. Additionally, four other hyperparameter configurations achieved 100% generalization on four out of five tasks: StairClimber, Maze, and TopOff.

Training used diverse initial state configurations. Whenever feasible, we enumerated all combinations of agent and goal placements. Specifically:

- For StairClimber, TopOff, and FourCorner, all possible agent-goal placements were used.
- For Maze, where full enumeration was computationally infeasible, we sampled 5 random mazes and placed the goal at every position on the grid.

The training grid sizes for each task were:

- 12×12 for StairClimber, TopOff, and FourCorner
- 8×8 for Maze and Harvester

C Parking Details

We used a single-hidden-layer DQN architecture with 64 units for the neural baseline. The agent operated over a discretized action space, where continuous actions were mapped onto n equally spaced values using a fixed action resolution. We performed a grid search over the hyperparameter values listed in Table 9. The selected hyper-parameters are shown in Table 10.

Hyperparameter	**Values Tested**
Learning rate	{`0.01`, `0.001`, `0.0001`}
Batch size	{`64`, `128`, `256`}
Target update frequency	{`100`, `500`, `1000`}
ϵ	{`0.1`, `0.01`}
Replay buffer size	{`1 million`, `2 million`}
Action resolution	{`3`, `5`, `7`}
Seeds per config	{`10`}

Table 9: DQN: Hyperparameter Sweep Configuration for Parking Domain

Hyperparameter	Selected Value
Learning rate	0.0001
Batch size	64
Target update frequency	1000
ϵ	0.01
Action resolution	2 million

Table 10: Best Hyperparameter Configuration Used for Final Training

The original PARKING benchmark introduced by Inala *et al.* [2020] was not designed with reinforcement learning in mind—it provides "safety check" to invalidate policies that crash the car or get out of boundaries. We define both a shaped reward function and a termination condition to adapt it for RL. If the agent successfully reaches the parking exit, the episode ends with a large positive reward (2 × max episode length); if it takes an unsafe action, it terminates immediately with a large negative penalty (–2 × max episode length). Otherwise, at each timestep the agent receives $r_t = -\big(2|x_{\text{agent}} - x_{\text{goal}}| + |y_{\text{agent}} - y_{\text{goal}}|\big) - 1$, i.e., the (weighted) negative Manhattan distance minus an extra step penalty of 1, encouraging the car to move closer to the exit.

D SPARSEMAZE Details

We trained agents using Proximal Policy Optimization (PPO) [Schulman *et al.*, 2017] with the previous action included in the observation vector. A hyperparameter sweep was conducted with the values in Table 11.

Hyperparameter	Values Tested
Learning rate	{0.001, 0.0001, 0.00001}
Clipping coefficient	{0.01, 0.1, 0.2}
Entropy coefficient	{0.001, 0.01, 0.1}
L1 regularization	{0.0, 0.0001, 0.0005, 0.001}
Actor's hidden layer size	{32, 64}
Number of minibatches	{32, 64, 128}
Training time steps	{5 million}
Seeds per config	{5}

Table 11: PPO with a_{t-1} and GRU: Hyperparameter Sweep for SPARSEMAZE

As shown in Table 11, we evaluated a range of hyperparameters, and selected the best set listed in Table 12 for PPO with a_{t-1} and in Table 13 for PPO with GRU based on training AUC. This configuration was then trained with 30 seeds, and evaluation results were averaged over 10 distinct environment seeds per model. The environment grid size was set to 20×20 during training. A smaller grid, such as 8×8, would make the maze too sparse for effective learning in this domain. Results are available in Table 4.

Hyperparameter	Selected Value
Learning rate	0.0001
Actor's hidden layer size	64
Entropy coefficient	0.01
Clipping coefficient	0.1
Number of minibatches	32

Table 12: Best Hyperparameter Configuration of PPO with a_{t-1} for SPARSEMAZE

Listing 1 shows the policy that Funsearch [Romera-Paredes *et al.*, 2024] gave after 21 iterations and Listing 2 is the given prompt. We used Qwen 2.5-Coder, 32B variant [Bai *et al.*, 2023] as the LLM for this part.

Hyperparameter	Selected Value
Learning rate	0.0001
Actor's hidden layer size	32
Entropy coefficient	0.01
Clipping coefficient	0.2
Number of minibatches	32
Value learning rate	0.0005
GRU hidden layer size	64

Table 13: Best Hyperparameter Configuration of PPO with GRU for SPARSEMAZE

Listing 1: Funsearch Policy

```
def get_action(env: KarelGymEnv) -> Union[list[int], str]:
  """Creates a policy that returns a list of actions for the Karel agent to take in
      the environment."""

  def find_path_bfs(start_r, start_c, goal_r, goal_c, walls):
    from collections import deque

    queue = deque([(start_r, start_c, [])])
    visited = set((start_r, start_c))
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

    while queue:
      current_r, current_c, path = queue.popleft()

      if (current_r, current_c) == (goal_r, goal_c):
        return path

      for dr, dc in directions:
        new_r, new_c = current_r + dr, current_c + dc
        if 0 <= new_r < env.env_height and 0 <= new_c < env.env_width and not walls[
            new_r, new_c] and (new_r, new_c) not in visited:
          visited.add((new_r, new_c))
          queue.append((new_r, new_c, path + [(dr, dc)]))

    return []

  def convert_path_to_actions(start_r, start_c, start_d, path):
    actions = []
    current_r, current_c, d = start_r, start_c, start_d
    directions_map = {(0, 1): 1, (1, 0): 2, (0, -1): 3, (-1, 0): 0}

    for dr, dc in path:
      target_d = directions_map[(dr, dc)]

      while d != target_d:
        if (d + 1) % 4 == target_d: # Turn right
          actions.append(2)
        elif (d - 1) % 4 == target_d: # Turn left
          actions.append(1)
        else: # Turn around
          actions.extend([1, 1])

        d = target_d

      actions.append(0) # Move forward

    return actions

  state_arr = env.task.get_state()
```

```
  walls = state_arr[4].astype(bool)
  r, c, d = env.task.get_hero_pos()
  goal_r, goal_c = env.task_specific.marker_position

  path = find_path_bfs(r, c, goal_r, goal_c, walls)

  if not path:
    return [random.randint(0, 4) for _ in range(50)]

  actions = convert_path_to_actions(r, c, d, path)

  # Extend actions to ensure the policy has enough steps
  while len(actions) < 50:
      actions.append(random.choice([1, 2])) # Randomly turn left or right

  return actions
```

Listing 2: Funsearch Prompt

```
"""
Specification for the Karel SparseMaze environment.

We are searching for a function ‘get_action(env)‘ that returns a list of actions
    list[int] for the Karel environment.
get_action(env) should return a policy that can solve the maze all the time,
    regardless of the initial configuration. Then by calling this policy, it can
    get the actions for that specific initial configuration.

Input is a KarelGymEnv object.
- You can access the height and width of the env like this: env.env_width, env.
    env_height
- You can access walls like this:
  # static walls from feature index 4 of the Karel state
  state_arr = env.task.get_state() # shape: (features, H, W)
  self.walls = state_arr[4].astype(bool) # True where wall
- You can the row, column, and direction of the agenr like this:
  r, c, d = env.task.get_hero_pos()
- And the directions are like this:
  0: 'Karel facing North',
  1: 'Karel facing East',
  2: 'Karel facing South',
  3: 'Karel facing West',
- Access the goal marker position like this:
  goal_r, goal_c = env.task_specific.marker_position
- You can access the observation like this:
  obs = env._get_observation_dsl() # shape: (4,), [frontIsClear, leftIsClear,
      rightIsClear, markersPresent]

The actions are:
  0: move
  1: turnLeft
  2: turnRight
  3: pickMarker (not used in maze)
  4: putMarker (not used in maze)

In the maze task, the agent starts at a fixed position and must find its path to a
    goal marker. The environment uses a sparse reward: 1 when reaching the goal, 0
    otherwise.
The environment is a sparse maze (corridors are 2 cells wide) and has multiple
    initial configurations (both mazes and goal positions).

This specification describes the key classes, variables, and functions used to
    define a Gym compatible "Karel SparseMaze" task, where an agent navigates a
carved maze to reach a goal marker under sparse rewards.
```

```
Package Layout:
  project_root/
  |-- funsearch/
      |-- implementation/
      | |-- utils.py
      | |-- temp1.py # top-level script that calls evaluate and get_action
      |-- karel_wide_maze/
      | |-- __init__.py
      | |-- karel_wide_maze.py
      | |-- karel_wide_maze_prompt_spec.py
      | |-- gym_envs/
      | | |-- __init__.py
      | | |-- karel_gym.py # Defines KarelGymEnv
      | |-- karel_tasks/
      | | |-- __init__.py
      | | |-- maze.py # Defines Maze, MazeSparse, MazeWide, etc.
      | |-- karel/
      | | |-- __init__.py
      | | |-- environment.py # Defines KarelEnvironment and features
      | |-- base/
      | |-- __init__.py
      | |-- task.py # Defines BaseTask
      |...

Usage Summary:
  1. The FunSearch framework "evolves" a Python function `get_action(env)` to
       maximize `evaluate(n)`.
  2. `evaluate(n)` runs n episodes of the Karel SparseMaze environment, each seeded
       differently, with different locations for walls and goal.
  3. Each episode calls `run_episode()`, which repeatedly:
     - Queries `get_action(env)` to obtain actions: {0..4}.
     - Steps the Gym environment and accumulates sparse/dense rewards.
     - Terminates when Karel reaches the goal or max_steps is reached.
  4. Maze-classes in karel_tasks/maze.py carve out a random maze layout (via DFS),
       set a goal marker,
     and compute rewards either sparsely (1 upon reach) or densely (normalized
         distance progress).
  5. KarelGymEnv wraps these Tasks into a standard Gym API: it exposes `step()`, `
       reset()`, `render()`,
     `action_space`, `observation_space`, and handles "multiple initial
         configurations" if requested.
"""
```