RECLAIMING THE SOURCE OF PROGRAMMATIC POLICIES: PROGRAMMATIC VERSUS LATENT SPACES

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent works have introduced LEAPS and HPRL, systems that learn latent spaces of domain-specific languages, which are used to define programmatic policies for partially observable Markov decision processes (POMDPs). These systems induce a latent space while optimizing losses such as the behavior loss, which aim to achieve locality in program behavior, meaning that vectors close in the latent space should correspond to similarly behaving programs. In this paper, we show that the programmatic space, induced by the domain-specific language and requiring no training, presents values for the behavior loss similar to those observed in latent spaces presented in previous work. Moreover, algorithms searching in the programmatic space significantly outperform those in LEAPS and HPRL. To explain our results, we measured the “friendliness” of the two spaces to local search algorithms. We discovered that algorithms are more likely to stop at local maxima when searching in the latent space than when searching in the programmatic space. This implies that the optimization topology of the programmatic space, induced by the reward function in conjunction with the neighborhood function, is more conducive to search than that of the latent space. This result provides an explanation for the superior performance in the programmatic space.

1 INTRODUCTION

Programmatic representations of policies for solving reinforcement learning problems can offer important advantages over alternatives, such as neural representations. Previous work showed that due to the inductive bias of the language in which such policies are written, they tend to generalize better to unseen scenarios (Inala et al., 2020; Trivedi et al., 2021). The programmatic nature of policies also allows modularization and reuse of parts of programs (Ellis et al., 2020; Aleixo & Lelis, 2023), which can speed up learning. Previous work also showed that programmatic policies can be more amenable to verification (Bastani et al., 2018) and interpretability (Verma et al., 2018, 2019).

The main challenge with programmatic representations is that, in the synthesis process, one needs to search in very large and often discontinuous policy spaces. While some domain-specific languages are differentiable and gradient descent methods can be used (Qiu & Zhu, 2022; Orfanos & Lelis, 2023), more expressive languages that allow the synthesis of policies with internal states (Inala et al., 2020; Trivedi et al., 2021; Liu et al., 2023) are often full of discontinuities, and thus one must use combinatorial search algorithms to find suitable programs. In an attempt to ease the process of searching for policies, recent work introduced Learning Embeddings for Latent Program Synthesis (LEAPS) (Trivedi et al., 2021), a system that learns a latent space of a domain-specific language with locality in program behavior. That is, if two vectors are near each other in the latent space, then they should decode programs with similar behavior. Once the latent space is learned, LEAPS uses a local search algorithm to find a latent vector that is decoded into a program encoding a policy for a target task. Liu et al. (2023) extended LEAPS to propose a hierarchical framework, HPRL, to allow the synthesis of programs outside the distribution of programs used to learn the latent space.

In this paper, we evaluate local search algorithms operating in the programmatic space induced by the domain-specific language, and compare them with LEAPS and HPRL. Searching in the original programmatic space involves defining an initial candidate solution (i.e., a program) and a neighborhood function that returns the neighbor programs of a candidate solution. We generate neighbors by following a process similar to the one used in genetic programming algorithms (Koza, 1992), which
was previously used in the context of programmatic policies in multi-agent settings \cite{Medeiros et al., 2022, Aleixo & Lelis, 2023}. Namely, we generate a number of neighbor programs by modifying parts of the program that represents the candidate. We hypothesized that searching for good policies in the latent space is not easier than in the original programmatic space, as seen in the problems and latent spaces considered in previous work \cite{Trivedi et al., 2021, Liu et al., 2023}. Our rationale is that the latent spaces used in previous work are also high-dimensional and non-differentiable, given that the evaluation of latent vectors depends on the execution of the decoded program.

We tested our hypothesis using the same set of problems used to evaluate LEAPS and HPRL. We discovered that a hill-climbing algorithm in the programmatic space, HC, outperformed both LEAPS and HPRL. HC consistently matched or exceeded the performance of the two latent-based methods. To interpret our findings, we examined the value of the behavior loss, used to learn the latent space, within the latent and programmatic spaces, and found that they are comparable in each. Although loss values do not account for performance differences between the spaces, they suggest that optimizing solely for the behavior loss does not necessarily produce spaces conducive to search.

We then evaluated the “friendliness” of the two spaces for local search, which is formalized as the probability of a hill-climbing search, which is randomly initialized in the space, converging to a solution with at least a given target reward value. This probability is a measure of the topology of the search space for a given distribution of initial candidates, since it measures the likelihood that the search will be stuck in local maxima. We observed that the programmatic space is never worse and is often much superior to the latent space for a wide range of target reward values. These results not only support our hypothesis that searching in the latent space is not easier than searching in the programmatic space, but also suggest that the programmatic space can be more conducive to search.

We conjecture that the effectiveness of latent spaces in the context of the synthesis of programmatic policies depends on two properties: how much the latent space compresses the original space and how conducive to search the space is. Intuitively, by compressing the space, the search becomes easier as one has fewer programs to evaluate; by being more conducive to search, the search signal could directly guide the search toward high-return programs. Our empirical results suggest that current systems for learning latent spaces lack either or both of these properties, since the search in the original programmatic space is more effective than the search in latent spaces. The contribution of this paper is to highlight the importance of using a baseline that searches directly in the programmatic space in this line of research; such a baseline was missing in previous work. Our baseline allows us to better evaluate and understand the progress in systems that search in latent spaces.

1.1 Related Works

Most of the early work on programmatic policies considered stateless programs, such as decision tree. For example, \cite{Verma et al., 2018} and \cite{Verma et al., 2019} learn tree-like programs with no internal states. \cite{Bastani et al., 2018} use imitation learning to induce decision trees encoding policies. \cite{Qiu & Zhu, 2022} learn programmatic policies by using a language of differentiable programs, which are identical to oblique decision trees. Learning programmatic policies with internal states, such as programs with while-loops, can be more challenging. \cite{Inala et al., 2020} presented an algorithm for learning policies in the form of finite-state machines, which can represent loops. As in this paper, \cite{Trivedi et al., 2021} and \cite{Liu et al., 2023} also consider programmatic policies with internal states through loops. There is also work on programmatic policies in the multi-agent context, where one learns a sequence of policies within self-play algorithms \cite{Medeiros et al., 2022, Aleixo & Lelis, 2023}. Most of these previous works search in the space of programs with a local search algorithm.

Program synthesis problems also pose problems similar to the ones discussed in this paper \cite{Waldinger & Lee, 1969, Solar-Lezama et al., 2006}, where one must search in the programmatic spaces for a program that satisfies the user’s intent. These problems can be solved with brute-force search \cite{Udupa et al., 2013} or algorithms that are guided by a function, which is often learned \cite{Odena et al., 2021, Barke et al., 2020, Shi et al., 2022, Ellis et al., 2020, Wong et al., 2021}. A common method to learning such guiding functions is to use a self-supervised approach in which the learning system exploits the structure of the language to generate training data. Similar to these works, LEAPS can be seen as an attempt to learn a function a priori to help with the search in the programmatic space in the context of reinforcement learning.
We are interested in episodic partially observable Markov decision processes (POMDPs) with deterministic dynamics and undiscounted reward functions. This setting can be described by the tuple \((S, A, O, p, q, r, S_0)\). In this formulation, \(S\) is the set of states, \(A\) is the set of actions and \(O\) is the set of observations in the environment. The function \(p : S \times A \rightarrow S\) determines the state transition dynamic of the environment, \(q : S \rightarrow O\) the observation given a state and \(r : S \times A \rightarrow \mathbb{R}\) the reward given a state and action. Finally, \(S_0\) defines the distribution for the initial state of an episode.

We consider that agents can interact in the environment following policies with internal states. These are defined by the function \(\pi : O \times H \rightarrow A \times H\), where \(H\) represents the internal state of the policy, initialized as a constant \(h_0\). Given an initial state \(s_0 \sim S_0\) and following the state transition \(s_{t+1} = \rho(s_t, a_t)\) and the policy \((a_t, h_{t+1}) = \pi(q(s_t), h_t)\) to determine the next states, we can define the trajectory as a function of the policy and initial state \(\tau(\pi, s_0) = (s_1, s_2, \ldots, s_T)\) for an episode with \(T\) time steps. This can be rewritten as \(\tau(\pi, s_0) = (a_0, a_1, \ldots, a_T)\), as state transitions are uniquely defined by a tuple state-action.

The goal of an agent acting in a POMDP is to maximize the cumulative reward over an episode. As the rewards during the episode depend uniquely on the initial state \(s_0\) and the policy \(\pi\), we can define the return of an episode as \(g(s_0, \pi) = \sum_{t=0}^{T} r(s_t, a_t)\). Our objective is to find an optimal policy \(\pi^*\) given a policy class \(\Pi\).

\[
\pi^* = \arg \max_{\pi \in \Pi} \mathbb{E}_{s_0 \sim S_0}[g(s_0, \pi)] \tag{1}
\]

### 2.1 Programmatic Policies

A programmatic policy class \(\Pi_{\text{DSL}}\) defines the set of policies that can be represented by a program \(\rho\) within a domain-specific language (DSL). Figure 1 shows a context-free grammar that defines the DSL for KAREL THE ROBOT, the problem domain we use in our experiments. A context-free grammar is represented by the tuple \((\Sigma, V, R, I)\). Here, \(\Sigma\) and \(V\) are sets of terminal and non-terminal symbols of the grammar. \(R\) defines the set of production rules that can be used to transform a non-terminal symbol into a sequence of terminal and non-terminal ones. Finally, \(I\) is the initial symbol of \(G\). In Figure 1, the non-terminal symbols are \(\rho, s, b, n, h\) and \(a\), where \(I = \rho\); terminal symbols include \(\text{WHILE}, \text{REPEAT}, \text{IF}\), etc. An example of a production rule is \(a := \text{move}\), where the non-terminal \(a\) is replaced by the action \(\text{move}\). This grammar accepts strings defining functions with loops, if-statements, and Boolean functions, such as \(\text{frontIsClear}\), over the observation space \(O\). The DSL also includes instructions defining actions in the action space \(A\), such as \(\text{move}\) and \(\text{turnLeft}\). The policy class \(\Pi_{\text{DSL}}\) is defined as the set of all programs that the grammar accepts.

The problem is to search in the space of programmatic policies \(\Pi_{\text{DSL}}\) for a policy that solves Equation (1).

Programs are represented in memory as abstract syntax trees (ASTs). In an AST, each node represents a production rule. For example, for the AST shown in Figure 2, its root represents the production rule \(\rho := \text{DEF run m(s m)}\), whose child is generated by the rule

\[
\text{DEF run m(s m)}
\]
Under review as a conference paper at ICLR 2024

DEF run m(
    IF c( markersPresent c) i(
        pickMarker move
    i)
)m

ρ
If
h
MP
PM
s
PM
M

Figure 2: An example of a program defined in KAREL DSL (left) and its AST representation (right). In the AST, MP stands for markersPresent, PM for pickMarker, and M for move.

$s := \text{IF } c( b \ c) i( s \ i)$. Its conditional expression is generated by the rules $b := h$ and $h := \text{markersPresent}$, while its statement is generated by $s := s; s$, branching into $s := \text{pickMarker}$ and $s := \text{move}$.

A programmatic policy $\rho \in \Pi_{DSL}$ is a policy with internal state, where $h_t$ can be interpreted as the pointer in the program $\rho$ after the action taken at time step $t-1$. The internal state $h_t$ and the current observation $q(s_t)$ are sufficient to uniquely determine the action $a_t$ the programmatic policy returns, thus its trajectory given an initial state is also deterministic.

3 SEARCH SPACES FOR PROGRAMMATIC POLICIES

We formalize searching for a programmatic policy as a local search problem. This involves specifying a feasible set $\mathcal{R} \subseteq \Pi_{DSL}$ and a corresponding neighborhood function $N_K : \mathcal{R} \rightarrow \mathcal{R}^K$ which, given a feasible solution $\rho \in \mathcal{R}$, defines its $K$-neighborhood. Consequently, we can interpret the search space as a graph $(V, E)$, where $V = \mathcal{R}$ and $E = \{(u, v) \mid u \in \mathcal{R}, v \in N_K(u)\}$.

In this work, we evaluate two search spaces: PROGRAMMATIC SPACE, which uses the DSL directly, and LATENT SPACE, which uses a learned embedding of the DSL.

3.1 PROGRAMMATIC SPACE

In this formulation, $\mathcal{R}^{prog}$ is a subset of the programs the DSL accepts. We define the subset $\mathcal{R}^{prog}$ by imposing constraints on the size of the programs. In particular, we limit the number of times the production rule $s := s; s$ (statement chaining) can be used, and we also limit the height of the abstract syntax tree (AST) of every program. These constraints are the ones used to determine the distribution of programs used to train the latent space of LEAPS, which we describe in Appendix A.

The $K$-neighborhood of a program $\rho \in \mathcal{R}^{prog}$, $N_K^{prog}(\rho)$, consists of $K$ samples of a mutation applied in $\rho$. A mutation is defined by uniformly sampling a node in the AST of $\rho$ and deleting one of its children, also chosen from a uniform distribution. In the newly created hole, we generate a new sub-tree by sequentially sampling a suitable production rule following the probability distribution used to generate the programs to train the latent space of LEAPS. We continue to sample production rules from the grammar until the newly created sub-tree does not have any non-terminal symbols and the neighbor program of $\rho$ is a valid program. We ignore programs that are not in $\mathcal{R}^{prog}$ through a sample rejection scheme. That is, if the neighbor of $\rho$ is not in $\mathcal{R}^{prog}$, we sample a new neighbor until we obtain one that is $\mathcal{R}^{prog}$. We present the probability distribution in Appendix B.

3.2 LATENT SPACE

LEAPS (Trivedi et al., 2021) and HPRL (Liu et al., 2023) introduce a method for defining a continuous search space for programmatic policies. This is defined by a variational auto-encoder (VAE), with encoder $Q : \Pi_{DSL} \rightarrow \mathbb{R}^d$ and decoder $P : \mathbb{R}^d \rightarrow \Pi_{DSL}$, trained to reconstruct the text representation of a program $\rho \in \Pi_{DSL}$. In addition to minimizing a reconstruction loss, the authors
additionally minimize two losses related to program behavior, which ensure that programs that yield similar trajectories over a set of initial states are embedded close together in the Latent Space.

As the Latent Space is constructed in a supervised manner, it is trained on a set of programs from a predetermined distribution. For the VAE reconstruction loss, the LEAPS authors used the same constraints as described for the Programmatic Space and the same probabilistic DSL rules to generate a program training set. For training the VAE behavior losses, they generate trajectories by running each program from the training set on a set of random initial states from the environment.

Given a trained Latent Space, its feasible set \( \mathcal{R}^{\text{lat}} \subseteq \Pi_{\text{DSL}} \) is the set of all programs that \( P \) can generate. Meanwhile, given a program \( \rho = P(z) \in \mathcal{R}^{\text{lat}}, z \in \mathbb{R}^d \), each program in its \( K \)-neighborhood \( N_K^{\text{lat},\sigma}(\rho) \) is given by decoding \( z + \epsilon \), where \( \epsilon \sim \mathcal{N}(0, \sigma I_d) \) and \( I_d \) represents the \( d \times d \) identity matrix. The standard deviation of the noise \( \sigma \) and the dimension \( d \) of the Latent Space are hyperparameters. Similarly to the Programmatic Space, we define the Latent Space as a graph \((V, E)\) by fixing the seed used to obtain the neighbors \( N_K^{\text{lat},\sigma} \) and the initial candidate \( z \).

4 LOCAL SEARCH ALGORITHMS

Once the Latent Space is learned, LEAPS relies on the Cross Entropy Method (CEM) [Rubinstein, 1999] to search the Latent Space for a vector that will decode into a program that approximates a solution to Equation 1. In addition to CEM, we also consider Cross Entropy Beam Search (CEBS), a method inspired in CEM that retains information of the best candidate solutions from a population. We also consider Hill Climbing (HC), as it is an algorithm that does not offer any mechanism for escaping local minima, and thus can be used to measure properties related to the space topology. In all search algorithms, we break ties arbitrarily.

Hill Climbing (HC) This algorithm starts by sampling a candidate solution, using the probabilistic context-free grammar from Appendix [B] in the case of a search in the Programmatic Space, or a vector from distribution \( \mathcal{N}(0, I_d) \) in the case of a search in the Latent Space. HC evaluates the \( K \)-neighborhood set of this initial candidate. If the neighborhood contains another candidate that yields a greater episodic return on the evaluated task than the initial candidate, then this process is repeated from that neighbor. Otherwise, the algorithm returns the best-seen candidate and its episodic return.

Cross Entropy Method (CEM) CEM randomly generates a set of \( M \) candidate solutions from \( \mathcal{N}(0, I_d) \) and evaluates all of them in terms of episodic return. CEM then calculates the mean of the latent vectors of the candidate solutions that yield the top \( E \) episodic return. This process is then repeated by defining the \( K \)-neighborhood of the mean latent vector as the new set of candidate solutions until an execution budget is exhausted.

Cross Entropy Beam Search (CEBS) CEBS maintains a set of promising candidates, called a beam. Starting from an initial candidate, we generate its \( K \) neighbors and select the best \( E \) candidates with respect to their episodic return as the beam of the search. Then we form the next beam by selecting the top \( E \) candidates from the pool given by all \( K \) neighbors of the candidates in the beam. This process continues until the mean of the episodic rewards seen in the beam is unchanged.

We present a pseudo-code implementation of HC and CEBS in Appendix [C].

5 EXPERIMENTS

In this section, we describe our empirical methodology for comparing the Programmatic Space and the Latent Space with respect to how conducive they are to local search algorithms. We have two sets of experiments. In the first set we compare CEM searching in the Latent Space, as presented in its original paper, with CEBS also searching in the Latent Space, and HPRL, which implements a hierarchical method over Latent Space, and HC in the Programmatic Space. In the second set, we compare the spaces in a more controlled experiment, where we fix the search algorithm to HC for both spaces. We start by introducing Karel the Robot, the domain in which we carried out our experiments.
5.1 KAREL THE ROBOT DOMAIN

KAREL THE ROBOT was firstly introduced as a programming learning environment (Pattis, 1994) and, due to its simplified structure, it has recently been adopted as a test-bed for program synthesis and reinforcement learning (Bunel et al., 2018; Chen et al., 2018; Shin et al., 2018; Trivedi et al., 2021). KAREL is a grid environment with local Boolean perceptions and discrete navigation actions.

To define the programmatic policy class for KAREL, we adopt the DSL introduced by the work of Bunel et al. (2018), described in Figure 1, which is presented as a context-free grammar. This DSL represents a subset of the original KAREL language. Namely, it does not allow the creation of subroutines or variable assignments. The language allows the agent to observe the presence of walls in the immediate neighborhood of the robot, with the perceptions \{\text{front}\left|\text{right}\right.\text{IsClear}, and the presence of markers in the current robot location with markersPresent and noMarkersPresent. The agent can then move the robot with the actions move and turn\{\text{Left}\mid\text{Right}\}, and interact with the markers with \{\text{put}\mid\text{pick}\}Marker.

We consider the KAREL and KAREL-HARD problem sets to define tasks. The KAREL set contains the tasks StairClimber, Maze, FourCorners, TopOff, Harvester and CleanHouse, all introduced by Trivedi et al. (2021). The KAREL-HARD problem set includes the tasks DoorKey, OneStroke, Seeder and Snake, designed by Liu et al. (2023) as more challenging problems to better outline the capacity of a programmatic solution. Trivedi et al. (2021) showed that these domains are challenging for reinforcement learning algorithms using neural representations, so LEAPS and HPRL represent the current state of the art in these problems. A detailed description of each task in both sets is available in Appendix D.

5.2 FIRST SET: REWARD-BASED EVALUATION

Our first evaluation reproduces the experiments of Trivedi et al. (2021) and Liu et al. (2023), where we add the results of HC searches in the PROGRAMMATIC SPACE. We use $K = 250$ as the neighborhood parameter for the HC search in the PROGRAMMATIC SPACE. For CEBS, we set the dimension of the latent vector $d = 256$, the neighborhood size $K = 64$, the elite size $E = 16$, and the noise $\sigma = 0.25$. We use the hyperparameters for CEM and HPRL exactly as described in their papers.

For each method, estimating an episodic return is done by calculating the mean of the task return function on a set of states sampled from the task’s initial state distribution. In this experiment, we consider a set of 16 initial states of each problem. We limit the execution of each method to a budget of $10^6$ program evaluations. If an algorithm fails to converge but its execution is still within the budget, we re-sample an initial program and restart the search. We report results over 32 seeds.

Table 1 summarizes our results in the KAREL and KAREL-HARD problem sets, comparing them to the results reported by the authors of LEAPS and HPRL. To better outline the performance of each algorithm, we plot the reached episodic return as a function of the number of episodes in Figure 3 and analyze differences in running time in Appendix E. We also present representative examples of programs by HC and CEBS synthesize in Appendix F. We further evaluate every method on a harder version of the environments, which were not used in previous work, in Appendix G.

We see that HC, based on PROGRAMMATIC SPACE, achieves the highest episodic return on every task compared to all methods based on LATENT SPACE. Furthermore, the plots show that, although HC and CEBS achieve the same mean episodic return at the end of the search process for Seeder, HC generally does so with a smaller number of samples.

We highlight the results observed on DoorKey. This is a two-stage task that requires the agent to pick up a marker in a room, yielding a 0.5 reward, which opens a second room that contains a goal square, which yields an extra 0.5 reward upon reaching it. LEAPS, HPRL and CEBS are only able to find programmatic policies that achieve 0.5 episodic return in this task, suggesting that their search procedures reach a local maximum that does not lead to a general solution. Meanwhile, HC achieves a mean episodic return that is higher than 0.5, suggesting that, in some cases, it is able to escape such local maxima and find policies that reach the final goal. We hypothesize that this is a property of the search space itself, since HC does not employ a mechanism to escape local maxima.

The CEM curve in this plot is based on our implementation of the algorithm, thus it diverges slightly from the results reported by the LEAPS authors in a few cases.
Table 1: Mean and standard error of final episodic returns of our proposed methods in KAREL and KAREL-HARD problem sets within a budget of $10^6$ program evaluations, compared to the reported results from baselines. LEAPS, HPRL, and CEBS search in LATENT SPACES, while HC searches in PROGRAMMATIC SPACES. HC and CEBS results are estimated over 32 seeds.

![Figure 3](image-url)  
Figure 3: Episodic return performance of all methods in KAREL and KAREL-HARD problem sets. Reported mean and 95% confidence interval over 32 seeds. The x-axis is represented in log scale.

5.3 Second Set: Topology-Based Evaluation

To better understand the discrepancy between HC and the algorithms searching in the LATENT SPACES, we analyze the PROGRAMMATIC and LATENT SPACE while controlling for the search algorithm.

5.3.1 Local Behavior Similarity Analysis

We first analyze whether the properties of the behavior loss used to train the LATENT SPACE can be seen throughout both search spaces. We define a metric that measures such loss in the neighborhood of randomly sampled programs in a search space, described as behavior-similarity below.

We first define the similarity between two programmatic policies $\rho$ and $\rho'$, with trajectories from an initial state $s_0$ given by $\tau(\rho, s_0) = (a_0, \ldots, a_T)$ and $\tau(\rho', s_0) = (a'_0, \ldots, a'_T)$, as

$$
\rho\text{-similarity}(\rho, \rho', s_0) = \frac{\max\{0 \leq t \leq T \mid a_{t+1} = a'_{t+1}\}}{L},
$$

(2)
where \( l = \min\{T, T'\} \) and \( L = \max\{T, T'\} \), and by \( x_{0:t} \) we mean \((x_0, \ldots, x_t)\). The \( \rho \)-similarity returns the normalized length of the longest common prefix of the action sequences produced by \( \rho \) and \( \rho' \) when starting at state \( s_0 \).

The behavior-similarity of a search space defined by the neighborhood function \( N_K \) with initial program distribution \( P_0 \) and initial state distribution \( S_0 \) is described as

\[
\text{behavior-similarity}(N_1, n_{\text{mutations}}) = \mathbb{E}_{\rho_{01} \sim P_0, s_0 \sim S_0} [\rho \text{-similarity}(\rho_{01}, \rho_{n_{\text{mutations}}}, s_0)],
\]

where \( \rho_{n_{\text{mutations}}} \) is the outcome of the neighborhood function \( N_1 \) with \( K = 1 \) recursively applied \( n_{\text{mutations}} \) times in \( \rho_{01} \), thus producing a path in the underlying search graph.

We observe that measuring behavior-similarity alone can be misleading due to the possibility of observing a neighbor that provides no change to the original program. We propose the metric identity-rate to complement the analysis, which measures the probability of observing a program in its own candidate neighborhood. The identity-rate is defined as follows.

\[
\text{identity-rate}(N_1, n_{\text{mutations}}) = \mathbb{E}_{\rho_{01} \sim P_0, s_0 \sim S_0} [\mathbb{I}\{\rho_{01} = \rho_{n_{\text{mutations}}}\}].
\]

To estimate the metrics given by Equations (3) and (4), we sample a set of 32 initial states from a distribution \( S_0 \) and a set of 1,000 initial programs from \( P_0 \). \( S_0 \) is composed of random KAREL maps unrelated to any task in the problem sets, and we set \( P_0 \) differently for each search space. For PROGRAMMATIC SPACE, it is given by the probabilistic DSL rules from Appendix B and for LATENT SPACE, it is given by \( \mathcal{N}(0, I_d) \). We run the metrics estimations as a function of \( n_{\text{mutations}} \in [1, 10] \) on PROGRAMMATIC SPACE and three specifications of LATENT SPACE, setting \( \sigma = \{0.1, 0.25, 0.5\} \) – hyperparameters commonly used by LEAPS and HPRL. The results are presented in Figure 4.

Although this was the objective of constructing the LATENT SPACE, we see that the PROGRAMMATIC SPACE achieves comparable behavior-similarity metrics – even though this space requires no prior training. We also see that although the setting \( \sigma = 0.1 \) on LATENT SPACE achieves high behavior-similarity, it is not more conducive to search due to its higher identity-rate.

However, the observed result does not provide evidence of the performance discrepancy that we observe while searching for policies that solve tasks.

5.3.2 CONVERGENCE ANALYSIS

Next, we now look at the topology of each search space with respect to the return functions of the tasks we want to solve. Specifically, we want to measure how conducive a given space is to search. To do so, we use HC in both LATENT SPACE and PROGRAMMATIC SPACE to estimate the chances that it has of converging to solutions of a given quality.

We define the convergence rate of a search space given by the neighborhood function \( N_K \), with initial program distribution \( P_0 \) and initial state distribution \( S_0 \) of a given POMDP. The initial program distribution for the PROGRAMMATIC SPACE is given by the LEAPS probabilistic context-free grammar (Appendix B); for the LATENT SPACE, it is given by the programs one decodes after sampling a latent vector from \( \mathcal{N}(0, I_d) \). The convergence rate is measured in terms of \( g_{\text{target}} \in [0, 1] \) as follows.

\[
\text{convergence-rate}(N_K, g_{\text{target}}) = \mathbb{E}_{\rho_0 \sim P_0, s_0 \sim S_0} [\mathbb{I}\{g_{\text{HC}}(\rho_0, s_0) \geq g_{\text{target}}\}],
\]
Figure 5: Convergence rate of PROGRAMMATIC SPACE and LATENT SPACE with neighborhood size $K = 250$, guided by hill-climbing. Reported mean and 95% confidence interval of estimation over a set of 10,000 seeds. The plots for DoorKey and Snake show a zoomed-in region highlighting runs of the search that are able to achieve reward values larger than 0.5 in these two domains.

where $g_{\text{HC}}(\rho_0, s_0)$ is the return of the best-performing program as a result of applying HC in the space defined by $N_K$ starting on $\rho_0$ while evaluating on $s_0$, and $K$ is the neighborhood size.

As the HC return depends on the task we are solving, we estimate different values from Equation 5 for each task. The estimation involves sampling a set of 32 states given by the task’s initial state distribution, and sampling 250 initial programs to start each execution of HC. The estimation of convergence-rate of PROGRAMMATIC SPACE and LATENT SPACE, both set to a neighborhood size $K = 250$, for every task in our problem sets is shown in Figure 5 as a function of $g_{\text{target}} \in [0, 1]$.

In the figure, we show a zoomed-in plot for the tasks DoorKey and Snake to better visualize cases with low convergence rate. Table 1 and Figure 3 show that the search in the programmatic space achieves reward values larger than 0.5 for DoorKey and Snake; the zoomed-in regions in Figure 5 show that these are rare events, but possible to be observed with a reasonable search budget.

We also present further analyses with different neighborhood sizes in Appendix H and we evaluate convergence with CEM and CEBS as search algorithms in Appendix I.

The plots show that, even in tasks where HC only matched or performed marginally better than latent methods, the PROGRAMMATIC SPACE is more likely to yield policies with greater episodic return. This indicates that this search space is more conducive to search than the LATENT SPACE, regardless of the search algorithm.

6 CONCLUSION

In this paper, we showed that despite the recent efforts in learning latent spaces to replace programmatic spaces, the latter is still more conducive to search. Empirical results in KAREL THE ROBOT showed that a simple hill-climbing algorithm searching in the programmatic space can significantly outperform the current state-of-the-art algorithms that search in latent spaces. We measured both the learned latent space and the programmatic space in terms of the loss function used to train the former. We discovered that both have similar loss values, despite the fact that the programmatic space does not require training. We also compared the topology of the two spaces through the probability of a hill-climbing search being stuck at local maxima in the two spaces, and found that the programmatic space is more conducive to search. Our results suggest that learning latent spaces for programming languages is still an open and challenging research question.

Here, $\sigma$ follows the values that the LEAPS authors chose for each task: $\sigma = 0.5$ for FourCorners and Harvester, $\sigma = 0.1$ for Maze, and $\sigma = 0.25$ for all other tasks.
REFERENCES


Pr(s := WHILE) = 0.15; Pr(s := IF) = 0.08; Pr(s := IFELSE) = 0.04;
Pr(s := REPEAT) = 0.03; Pr(s := s;s) = 0.5; Pr(s := a) = 0.2;
Pr(b := h) = 0.9; Pr(b := not ( h )) = 0.1;
Pr(n := 0) = Pr(n := 1) = ··· = Pr(n := 19) = 1/20;
Pr(h := frontIsClear) = 0.5; Pr(h := leftIsClear) = 0.15;
Pr(h := rightIsClear) = 0.15; Pr(h := markersPresent) = 0.1;
Pr(h := noMarkersPresent) = 0.1;
Pr(a := move) = 0.5; Pr(a := turnLeft) = 0.15; Pr(a := turnRight) = 0.15;
Pr(a := pickMarker) = 0.1; Pr(a := putMarker) = 0.1.

Figure 6: Adopted probabilities for the KAREL THE ROBOT DSL as a probabilistic context-free grammar.

A SETTINGS FOR GENERATING PROGRAMS

In this work, we use the same constraints for generating programs as the LEAPS project (Trivedi et al., 2021), as described below.

• Maximum AST height: 4;
• Maximum statement chaining (s := s; s rule): 6;
• Maximum program length (in number of symbols in the program text representation): 45.

B PROBABILITIES FOR DSL PRODUCTION RULES

We adopt a fixed probability for each DSL production rule, described in Figure 6 as a probabilistic context-free grammar. The adopted probabilities are based on the LEAPS project specifications (Trivedi et al., 2021).

C ALGORITHM DETAILS OF SEARCH ALGORITHMS

We present pseudo-code implementations of HC and CEBS as described in Section 4 in Algorithms 1 and 2 respectively.

Algorithm 1 Hill Climbing for Programmatic Policies

Require: N, the neighborhood function; K, the neighborhood size; P0, the initial program distribution; g: the task return function; S0, the set of task initial states.
Ensure: ρ, best-seen program with respect to highest episodic return estimate; ˆg, estimated episodic return of best-seen program.

1: ˆρ ∼ P0
2: ˆg ← 1 |S0| ∑ s0∈S0 g(πρ, s0)
3: repeat
4:   in_local_maximum ← true
5:   candidates ← NK(ˆρ)
6:   for each ρ in candidates do
7:       g ← 1 |S0| ∑ s0∈S0 g(πρ, s0)
8:       if g > ˆg then
9:           ˆρ ← ρ
10:          ˆg ← g
11:     end if
12: end for
13:   in_local_maximum ← false
14: until in_local_maximum = true
Algorithm 2 Cross-Entropy Beam Search for Programmatic Policies in LATENT SPACES

Require: $N_{\text{lat}}$, the neighborhood function; $K$, the neighborhood size; $E$, the size of the beam; $\sigma$, the noise parameter for the LATENT SPACE; $P_0$, the initial program distribution; $g$: the task return function; $S_0$, the set of task initial states.
Ensure: $\tilde{\rho}$, best-seen program with respect to highest episodic return estimate; $\bar{g}$, estimated episodic return of best-seen program.

1: $\tilde{\rho} \sim P_0$
2: $\bar{g} \leftarrow 1$
3: $S_0 \in S_0 g(\pi_{\tilde{\rho}}, s_0)$
4: $\text{best_mean_elite_return} \leftarrow -\infty$
5: $\text{candidates} \leftarrow N_{\text{lat}}, \sigma K(\tilde{\rho})$
6: repeat
7: in_local_maximum $\leftarrow$ true
8: for each $\rho$ in candidates do
9: $g$.append$(\frac{1}{|S_0|} \sum_{s_0 \in S_0} g(\pi_{\rho}, s_0))$
10: if $g[-1] > \bar{g}$ then $\triangleright$ Index -1 represents the last element in the list.
11: $\tilde{\rho} \leftarrow \rho$
12: $\bar{g} \leftarrow g[-1]$
13: $\text{elite_indices} \leftarrow \text{argtop-E}(g)$
14: if $\frac{1}{\text{elite_count}} \sum_{i \in \text{elite_indices}} g[i] > \text{best_mean_elite_return}$ then
15: $\text{best_mean_elite_return} \leftarrow \frac{1}{E} \sum_{i \in \text{elite_indices}} g[i]$
16: in_local_maximum $\leftarrow$ false
17: candidates $\leftarrow \bigcup_{i \in \text{elite_indices}} N_{\text{lat}}, \sigma K/E(\rho[i])$ $\triangleright$ Aggregates as a $K$-neighborhood of the elite.
18: until in_local_maximum = true

D KAREL problem sets

In this Section, we specify the initial state and return function of every task in KAREL and KAREL-HARD problem sets. Further details of each task are present in LEAPS (Trivedi et al., 2021) and HPRL (Liu et al., 2023), works that introduced KAREL and KAREL-HARD, respectively.

D.1 KAREL

STAIRCLIMBER This environment is given by a $12 \times 12$ grid with stairs formed by walls. The agent starts on a random position on the stairs and its goal is to reach a marker that is also randomly initialized on the stairs. If the agent reaches the marker, the agent receives 1 as an episodic return and 0 otherwise. If the agent moves to an invalid position, i.e. outside the contour of the stairs, the episode terminates with a $-1$ return.

MAZE A random maze is initialized on an $8 \times 8$ grid, and a random marker is placed on an empty square as a goal. The agent starts on a random empty square of the grid and its goal is to reach the marker goal, which yields a 1 episodic return. Otherwise, the agent receives 0 as a return.

TOP-OFF Markers are placed randomly on the bottom row of an empty $12 \times 12$ grid. The goal of the agent, initialized on the bottom left of the map, is to place one extra marker on top of every marker on the map. The return of the episode is given by the number of markers that have been topped off divided by the total number of markers.

FOUR-CORNERS Starting on a random cell on the bottom row of an empty $12 \times 12$ grid, the goal of the agent is to place one marker in each corner of the map. Return is given by the number of corners with one marker divided by four.

HARVESTER The agent starts on a random cell on the bottom row of an $8 \times 8$ grid, that starts with a marker on each cell. The goal of the agent is to pick up every marker on the map. Return is given by the number of picked-up markers divided by the total number of markers.
Under review as a conference paper at ICLR 2024

<table>
<thead>
<tr>
<th></th>
<th>PROGRAMMATIC SPACE</th>
<th>LATENT SPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed time (seconds)</td>
<td>0.0021 ± 0.0002</td>
<td>0.0293 ± 0.0004</td>
</tr>
</tbody>
</table>

Table 2: Time for generating one neighbor from a given candidate, measured over 1,000 initial random candidates. Reported mean and 95% confidence interval.

CLEANHOUSE  In this task, the agent starts on a fixed cell of a complex 14 × 22 grid environment made of many connected rooms, with ten markers randomly placed adjacent to the walls. The goal of the agent is to pick up every marker on the map and the return is given by the number of picked-up markers divided by the total number of markers.

D.2 KAREL-HARD

DOORKEY  The agent starts on a random position on the left side of an 8 × 8 grid that is vertically split into two chambers. The agent goal is to pick up a marker on the left chamber, which opens a door connecting both chambers and allows the agent to reach a goal marker. Picking up the first marker yields a 0.5 reward, and reaching the goal yields an additional 0.5.

ONESTROKE  Starting on a random position of an empty 8 × 8 grid, the goal of the agent is to visit every grid cell without repeating. Visited cells become a wall that terminates the episode upon touching. The episodic return is given by the number of visited cells divided by the total number of cells in the initial state.

SEEDER  The environment starts as an empty 8 × 8 grid, with the agent placed randomly in any square. The agent’s goal is to place one marker in every empty cell of the map. The return is given by the number of cells with one marker divided by the total number of empty cells at the start of the episode.

SNAKE  In this task, the agent and one marker are randomly placed on an empty 8 × 8 grid. The agent acts like the head of a snake, whose body grows each time a marker is collected. The goal of the agent is to touch the marker on the map without colliding with the snake’s body, which terminates the episode. Each time the marker is collected, it is placed in a new random location, until 20 markers are collected. The episodic return is given by the number of collected markers divided by 20.

E  RUNNING TIME COMPARISON OF PROGRAMMATIC AND LATENT SPACES

In this section, we compare the neighborhood generation process of each search space in terms of running time. We do this by measuring the time the PROGRAMMATIC SPACE and the LATENT SPACE take to generate one neighbor from a given candidate program, sampled from the initial distribution, and present the results in Table 2. We see that sampling from the programmatic space is more than 10 times faster than sampling from the latent space.

F  EXAMPLES OF OBTAINED SOLUTIONS

In this section, we show representative examples of programmatic policies from HC and CEBS across some relevant tasks. We selected programs that yield the highest return for each algorithm. Results are presented in Tables 3 and 4 for HC and CEBS, respectively.

G  EVALUATION ON CRASHABLE KAREL

To further evaluate the search algorithms, we propose a modification of the KAREL environment. In this version, which we name CRASHABLE, invalid actions terminate episodes. This change implies that the same tasks from KAREL and KAREL-HARD become more difficult to solve, as the valid
**Task** | **Solution** | **Return**  
--- | --- | ---  
Harvester | DEF run m( WHILE c( leftIsClear c) w( WHILE c( leftIsClear c) w( REPEAT R=14 r( move pickMarker r) turnRight w) WHILE c( rightIsClear c) w( pickMarker turnRight move turnLeft w) WHILE c( frontIsClear c) w( move w) w) m) | 1.0  
CleanHouse | DEF run m( WHILE c( leftIsClear c) w( move turnRight move move w) WHILE c( frontIsClear c) w( turnRight w) WHILE c( noMarkersPresent c) w( move REPEAT R=7 r( turnLeft move pickMarker r) w) move turnLeft m) | 1.0  
DoorKey | DEF run m( WHILE c( frontIsClear c) w( move w) turnLeft move WHILE c( noMarkersPresent c) w( turnRight move move w) IF c( leftIsClear c) i( pickMarker move move WHILE c( noMarkersPresent c) w( move turnRight move w) putMarker i) m) | 1.0  
OneStroke | DEF run m( IF c( frontIsClear c) i( turnRight i) WHILE c( noMarkersPresent c) w( WHILE c( frontIsClear c) w( turnRight move w) turnLeft IFELSE c( frontIsClear c) i( move turnRight pickMarker move move move i) ELSE e( turnRight move e) w) m) | 0.953  
Seeder | DEF run m( turnLeft WHILE c( noMarkersPresent c) w( putMarker REPEAT R=10 r( move r) REPEAT R=5 r( WHILE c( markersPresent c) w( turnLeft move turnRight w) pickMarker r) w) WHILE c( frontIsClear c) w( turnLeft w) m) | 1.0  
Snake | DEF run m( turnLeft WHILE c( frontIsClear c) w( move w) WHILE c( rightIsClear c) w( WHILE c( rightIsClear c) w( move turnLeft move IF c( frontIsClear c) i( move move i) turnLeft w) putMarker WHILE c( rightIsClear c) w( putMarker w) turnRight w) m) | 1.0  

Table 3: Representative high-return solutions from HC searches in the PROGRAMMATIC SPACE.
<table>
<thead>
<tr>
<th>Task</th>
<th>Solution</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>HARVESTER</td>
<td>DEF run m( WHILE c( leftIsClear c) w( move pickMarker move turnLeft pickMarker move pickMarker move turnLeft move pickMarker move turnLeft move pickMarker move w) m)</td>
<td>1.0</td>
</tr>
<tr>
<td>CLEANHOUSE</td>
<td>DEF run m( WHILE c( noMarkersPresent c) w( move pickMarker turnLeft w) WHILE c( leftIsClear c) w( move move w) WHILE c( frontIsClear c) w( move move w) m)</td>
<td>1.0</td>
</tr>
<tr>
<td>DOORKEY</td>
<td>DEF run m( WHILE c( rightIsClear c) w( turnLeft pickMarker w) WHILE c( noMarkersPresent c) w( turnRight move w) pickMarker WHILE c( noMarkersPresent c) w( turnRight move w) putMarker m)</td>
<td>0.6875</td>
</tr>
<tr>
<td>ONESTROKE</td>
<td>DEF run m( WHILE c( noMarkersPresent c) w( turnLeft move turnLeft WHILE c( frontIsClear c) w( turnLeft move w) pickMarker move move move w) WHILE c( noMarkersPresent c) w( turnLeft move w) pickMarker move move move move m)</td>
<td>0.9288</td>
</tr>
<tr>
<td>SEEDER</td>
<td>DEF run m( WHILE c( noMarkersPresent c) w( putMarker move move WHILE c( markersPresent c) w( turnRight move w) w) m)</td>
<td>1.0</td>
</tr>
<tr>
<td>SNAKE</td>
<td>DEF run m( WHILE c( noMarkersPresent c) w( REPEAT R=13 r( IFEELSE c( frontIsClear c) i( turnRight move pickMarker i) ELSE e( move pickMarker REPEAT R=13 r( turnRight move r) REPEAT R=13 r( pickMarker r) move e) pickMarker r) w) m)</td>
<td>0.4375</td>
</tr>
</tbody>
</table>

Table 4: Representative high-return solutions from CEBS search in LATENT SPACE.
trajectories are stricter. Figure 7 outlines the episodic return performance of CEM, CEBS, and HC on the CRASHABLE version of every task of the problem sets.

The results show a greater discrepancy between the programmatic and latent methods. We conjecture that this is because the LATENT SPACE was trained with trajectories obtained from the original environment setting and it is unable to generalize to the CRASHABLE environment. Since the PROGRAMMATIC SPACE does not require any training, it generalizes better to the CRASHABLE setting.

H CONVERGENCE RATE FOR DIFFERENT NEIGHBORHOOD SIZES

We expand the convergence rate analysis by adopting neighborhood functions $N_K$ with different neighborhood sizes $K$ in Equation (5). The convergence-rate analysis on a space given by a lower $K$ is related to how robust the search space is to conduct a search process, given that it relies on a small number of samples. On the other hand, the result of convergence-rate on higher $K$ gives us information about how capable the search space is, as it can expand a search state further to find the better candidate. Figure 8 compares the convergence-rate estimation of both PROGRAMMATIC and LATENT SPACE adopting $K = \{10, 250, 1000\}$, and suggests that the PROGRAMMATIC SPACE is both more robust, evidenced by the higher convergence-rate with $K = 10$, and more capable, shown by the superior convergence-rate with $K = 1,000$.

I CONVERGENCE RATE OF CEM AND CEBS

We further analyze the convergence rate of the LATENT SPACE by using CEM and CEBS as the search algorithm in Equation (5). Figure 9 compares the convergence rate obtained with HC (original setting), CEM, and CEBS, with $K = 64$ (neighborhood size). Although CEM and HC have a similar convergence rate across all tasks, we see that CEBS outperforms both in HARVESTER, DOORKEY, ONESTROKE and SEEDER. These results highlight the ability of CEBS to escape local optima. Despite the superior performance of CEBS, HC searching in the PROGRAMMATIC SPACE performs better than CEBS searching in the LATENT SPACE (see Figure 5).
Figure 8: Convergence rate of PROGRAMMATIC SPACE and LATENT SPACE with neighborhood sizes $K = 10$ (a), $K = 250$ (b) and $K = 1,000$ (c), guided by hill-climbing. Reported mean and 95% confidence interval of estimation over a set of 250 search initializations.
Figure 9: Convergence rate of \textsc{Latent Space} with neighborhood size $K = 64$, guided by HC, CEM, and CEBS. Reported mean and 95\% confidence interval over 1,000 seeds.