Evaluating Search Spaces for Programmatic Policies in POMDPs

by

Tales Henrique Carvalho

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

Abstract

Searching for programmatic policies to solve a reinforcement learning problem can be challenging, particularly when dealing with domain-specific languages (DSLs) that define policies with internal states for partially observable Markov decision processes (POMDPs). This is because they lead to complex and discontinuous search spaces, often requiring combinatorial search processes. To avoid searching in the programmatic space, the recent work LEAPS and HPRL learn latent spaces of DSLs, which are used to define policies for POMDPs. Aside from reconstructing programs from their embedding representations, these spaces are trained to achieve locality in program behavior, expecting that vectors close in the latent space decode to programs that behave similarly. In this work, we show that searching using a hill-climbing process in the original programmatic space, induced by the DSL itself and requiring no learning, achieves a similar locality measure in program behavior and significantly outperforms LEAPS and HPRL in finding high-reward policies. We further analyze the optimization topology induced by the neighborhood function of each search space in conjunction with the reward function of the POMDP. We show that a local search algorithm is more likely to stop in local maxima regions when searching for high-reward policies in the latent space than when searching in the original programmatic space. This result implies that the programmatic space is more conducive to local search and explains its superior performance.

Preface

This dissertation is an original work of the author done in collaboration with Levi Lelis. This work is currently under review for publication. Due to the collaborative nature of this work, the pronoun "we" is used in this script. However, I remain solely responsible for any technical or presentational errors present in this document.

Tales Henrique Carvalho

December, 2023

Acknowledgements

I would like to acknowledge that this work was supported by NSERC Discovery, CIFAR through CCAI Chair funding and by the Alberta Machine Intelligence Institute (Amii).

I would like to thank my supervisor Levi Lelis for the constant support throughout my research work. Levi was a great mentor during my studies, and taught me a lot about his specialization and research in general.

I would also like to thank my colleagues from the University of Alberta for supporting me in my studies and for making my time in Edmonton a lot more enjoyable. In particular, I would like to thank, in no particular order, Saqib Ameen, Rubens Moraes, and Kenneth Tjhia, from my research group, and Justin Stevens, Michael Ogezi, Shreya Kannan, Deep Gandhi, and Tian Du, from the Computing Science department. Your support was incredibly important in every step of my journey.

Lastly, and no less important, I would like to deeply thank my parents, my friends from Brazil, and my partner Maria Gabriela Goulart Neves for their emotional support during my studies. I owe a lot of my work to their endless support.

Table of Contents

List of Tables

List of Figures

Abbreviations

AST abstract syntax tree.

CEBS cross-entropy beam search.

CEM cross-entropy method.

CFG context-free grammar.

DSL domain-specific language.

ELBO evidence lower bound.

GRU gated recurrent unit.

HC hill-climbing.

KL Kullback-Leibler.

MDP Markov decision process.

PCFG probabilistic context-free grammar.

POMDP partially observable Markov decision process.

VAE variational autoencoder.

Chapter 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 and 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 and Zhu, 2022, Orfanos and 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, Hierarchical Programmatic Reinforcement Learning (HPRL), to allow the synthesis of programs outside the distribution of programs used to learn the latent space. In this work, we compare the learned latent space with the original programmatic space in the context of local search.

1.1 Problem Formulation

We are interested in episodic partially observable Markov decision processs (POMDPs) with deterministic dynamics and undiscounted reward functions. This setting can be described by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{O}, p, q, r, S_0)$. In this formulation, $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions and $\mathcal{O}$ is the set of observations in the environment. The function $p : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ determines the state transition dynamic of the environment, $q : \mathcal{S} \to \mathcal{O}$ the observation given a state and $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ 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. We assume that the functions p, q, and r are hidden from the agent, and it only has access to the output of $q(s_t)$ and $r(s_t)$, given a state of the environment s_t. Policies with internal states are defined by the function $\pi : \mathcal{O} \times \mathcal{H} \to \mathcal{A} \times \mathcal{H}$, where $\mathcal{H}$ represents the set of possible internal states of the policy, initialized as a constant h_0. Given an initial state $s_0 \sim S_0$ and following the state transition $s_{t+1} = p(s_t, a_t)$ 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 state-action tuple.

We restrict the policy representations for this work with the policy class Π_{DSL}, the

set of all policies that can be represented by a program ρ within a domain-specific language (DSL). Figure 1.1 shows an example of a programmatic policy in KAREL THE ROBOT, the environment used in this work. In the case of the example and throughout this work, a programmatic policy ρ defined by the DSL is a policy with internal states, where h_t is the pointer in the program ρ 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 policy returns, thus its trajectory given an initial state is also deterministic.

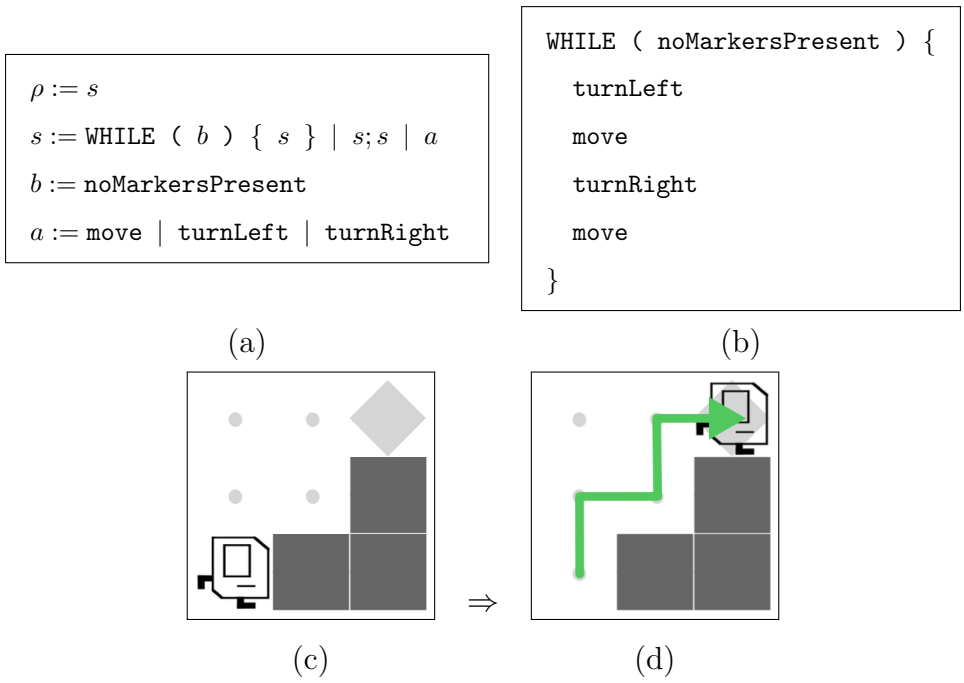


Figure 1.1: Example of a programmatic policy in a KAREL POMDP, indicating an example DSL to restrict the policy class as a context-free grammar (a), a programmatic policy represented in this DSL (b), an initial state of the environment (c), and the final state after executing the policy, with the trajectory highlighted in green (d).

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 programmatic policy ρ, we can define the return of an episode as $g(s_0, \rho) = \sum_{t=0}^{T} r(s_t, a_t)$. Our objective is to find an optimal programmatic policy ρ^*

given by

$$\rho^* = \arg\max_{\rho \in \Pi_{\text{DSL}}} \mathbb{E}_{s_0 \sim S_0}[g(s_0, \rho)], \tag{1.1}$$

restricted by the policy class Π_{DSL}.

1.2 Contributions

In our work, we evaluate local search algorithms operating in the programmatic space induced by the DSL, 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 [Medeiros et al., 2022, Aleixo and 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 searching in the original programmatic space. Our rationale is that, similarly to the programmatic space, the latent space is high-dimensional and non-differentiable, as the evaluation of latent vectors depends on executing 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 (HC) algorithm in the programmatic space 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 similar 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 an HC 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 programmatic space is easier than searching in the latent 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 work is to highlight the importance of using a baseline that searches directly in the programmatic space in this line of research. Our baseline allows us to better evaluate and understand the progress in systems that search in latent spaces.

Chapter 2

Background

In this Chapter, we present the background necessary to define our problem and to conduct our studies. We formally define programmatic policies in Section 2.1, we present how policies can be represented in a latent space in Section 2.2, and we formally define local search in Section 2.3.

2.1 Programmatic Policies

A programmatic policy class Π_{DSL} defines the set of policies that can be represented by a program ρ within a DSL. Figure 2.1 shows a context-free grammar (CFG) that defines a DSL for Karel the Robot [Bunel et al., 2018], the problem domain we use in our experiments. A CFG $\mathcal{G}$ is represented by the tuple (Σ, V, R, I). Here, Σ 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 $\mathcal{G}$. In Figure 2.1, the non-terminal symbols are ρ, s, b, n, h and a, where I is ρ; terminal symbols include `WHILE`, `REPEAT`, `IF`, etc. An example of a production rule is $a :=$ `move`, where the non-terminal a is replaced by the action `move`. This grammar accepts strings defining functions with loops, if-statements, and Boolean functions, such as `frontIsClear`, over the observation space $\mathcal{O}$ of the underlying POMDP. The DSL also includes instructions defining actions in the POMDP action space $\mathcal{A}$, such

as `move` and `turnLeft`.

Program $\rho :=$	`DEF run m(` s `m)`
Statement $s :=$	`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
Condition $b :=$	h | `not(` h `)`
Number $n :=$	0..19
Perception $h :=$	`frontIsClear` | `leftIsClear` | `rightIsClear` |
	`markersPresent` | `noMarkersPresent`
Action $a :=$	`move` | `turnLeft` | `turnRight` | `putMarker` | `pickMarker`

Figure 2.1: DSL for KAREL THE ROBOT as a context-free grammar.

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 2.2, its root represents the production rule $\rho :=$ `DEF run m(` s `m)`, whose child is generated by the rule $s :=$ `IF c(` b `c) i(` s `i)`. Its conditional expression is generated by the rules $b := h$ and $h :=$ `markersPresent`, while its statement is generated by $s := s; s$, branching into $s :=$ `pickMarker` and $s :=$ `move`.

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

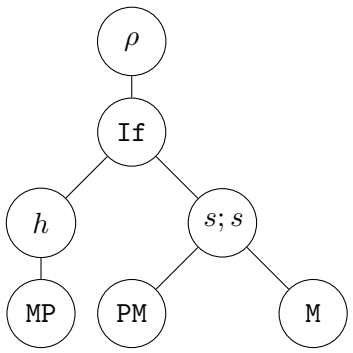


Figure 2.2: An example of a program defined in KAREL DSL (left) and its AST representation (right). In the AST, `MP` stands for `markersPresent`, `PM` for `pickMarker`, and `M` for `move`.

2.2 Latent Representation of Policies

LEAPS [Trivedi et al., 2021] and HPRL [Liu et al., 2023] introduce a method for defining a continuous representation of programmatic policies. This representation is constructed by training a variational autoencoder (VAE) to reconstruct the text representation of a program $\rho \in \Pi_{\text{DSL}}$.

2.2.1 Variational Autoencoders

The VAE framework consists of employing an encoder $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ and a decoder $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$, parameterized as neural networks with parameters $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$, respectively, to reconstruct i.i.d. data from a dataset $\boldsymbol{X} = \left\{\boldsymbol{x}^{(i)}\right\}_{i=1}^{X}$ using a d-dimensional latent variable $\boldsymbol{z} \in \mathbb{R}^d$ [Kingma and Welling, 2013]. The reconstruction is achieved by maximizing the evidence lower bound (ELBO) objective for each data point $\boldsymbol{x}^{(i)} \in \boldsymbol{X}$, which is equivalent [Kingma et al., 2019] to minimizing

$$\mathcal{L}_{\text{rec}}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}^{(i)}, \beta) = -\mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}^{(i)})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}|\boldsymbol{z})] + \beta D_{\text{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}^{(i)}) \| p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}^{(i)})), \quad (2.1)$$

where D_{KL} represents the Kullback-Leibler (KL) divergence, that approximates the latent representation to a posterior (in the case of VAE, the posterior is a normal distribution). The β hyperparameter, introduced by Higgins et al. [2016], constrains the capacity of the latent information channel, acting as a regularization term.

As the training objective involves sampling $\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}^{(i)})$, the encoder network is constructed to output $(\boldsymbol{\mu}, \boldsymbol{\sigma})$ from a data point, used to obtain $\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I}_d)$, $\odot$ represents element-wise multiplication and $\boldsymbol{I}_d$ is the $d \times d$ identity matrix. This, known as the reparameterization trick, allows $\mathcal{L}_{\text{rec}}$ to be optimized with gradient descent algorithms. In specific, the trick allows the first term of Equation 2.1 to be computed as a classification loss between the output of the decoder and the data point $\boldsymbol{x}^{(i)}$, and the second term to be approximated as

$$\beta D_{\text{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}^{(i)}) \| p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}^{(i)})) \approx -\frac{\beta}{2} \sum_{j=1}^{d} \left(1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2\right), \quad (2.2)$$

where μ_j and σ_j represent the j-th element of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$, respectively.

2.2.2 Training Objective

The VAE model introduced in LEAPS [Trivedi et al., 2021] parameterizes both encoder and decoder networks as gated recurrent units (GRUs). In addition to minimizing $\mathcal{L}_{\text{rec}}$ on a dataset composed of P programmatic policies $\left\{\rho^{(i)}\right\}_{i=1}^{P}$, the authors minimize two losses related to program behavior, which ensures that programs that yield similar trajectories over a set of initial states are embedded close together in the latent space.

We first define a method to measure the similarity in behavior between any two programs ρ and ρ'. By computing trajectories from an initial state s_0 as $\tau(\rho, s_0) = (a_0, \ldots, a_T)$ and $\tau(\rho', s_0) = (a'_0, \ldots, a'_{T'})$, we define the similarity as

$$\rho\text{-similarity}(\rho, \rho', s_0) = \frac{\max\{0 \leq t \leq l \mid a_{0:t} = a'_{0:t}\}}{L}, \tag{2.3}$$

where $l = \min\{T, T'\}$ and $L = \max\{T, T'\}$, $x_{0:t} = (x_0, \ldots, x_t)$, and $\max\{0 \leq t \leq l \mid a_{0:t} = a'_{0:t}\}$ is the length of the longest common prefix of the action sequences produced by ρ and ρ' when starting at state s_0. Thus, the ρ-similarity returns the normalized length of this longest common prefix of the action sequences.

The LEAPS framework uses this definition to optimize a program behavior reconstruction loss, maximizing the similarity between an input program and its decoding. This is achieved by minimizing

$$\mathcal{L}_{\text{R}}(\boldsymbol{\theta}, \boldsymbol{\phi}; \rho^{(i)}) = -\mathbb{E}_{\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\rho^{(i)}), \rho' \sim p_{\boldsymbol{\theta}}(\rho|\boldsymbol{z}), s_0 \sim S_0}[\rho\text{-similarity}(\rho^{(i)}, \rho', s_0)], \tag{2.4}$$

estimated by sampling multiple states from a given initial state distribution S_0, and by sampling multiple programs ρ' from $p_{\boldsymbol{\theta}}(\rho|\boldsymbol{z})$. Minimizing $\mathcal{L}_{\text{R}}$ is not differentiable, as running the programs in the environment does not allow gradients to flow back to the model. Therefore, this is done via the policy gradient algorithm REINFORCE [Williams, 1992], where ρ-similarity composes its reward signal.

Additionally, the authors introduce a behavior-related loss that can be trained with supervised learning. This relies on a *neural program executor* model $\pi_{\boldsymbol{\psi}}(a|\boldsymbol{z}, s_t)$, parameterized as a GRU with parameters $\boldsymbol{\psi}$, that sequentially predicts the probability of an agent action a_t given its latent encoding $\boldsymbol{z}$ and current state in environment s_t. The behavior similarity can then be encouraged by minimizing the cross-entropy between the actions obtained by executing the program ρ and the action probabilities from the model, formalized by

$$\mathcal{L}_{\mathrm{L}}(\boldsymbol{\psi}, \boldsymbol{\phi}; \rho^{(i)}) = -\mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\rho^{(i)}), s_0\sim S_0}[\sum_{t=1}^{T}\sum_{j=1}^{|\mathcal{A}|}\mathbb{1}\{a_t = a_j\}\log \pi_{\boldsymbol{\psi}}(a_j|\boldsymbol{z}, s_t)], \tag{2.5}$$

also estimated by sampling multiple states from a given initial state distribution S_0. In this equation, each a_t is obtained by computing the trajectory $\tau(\rho, s_0) = (a_0, \ldots, a_T)$

The training objective of the autoencoder framework can be summarized by the combined objective

$$\min_{\boldsymbol{\theta},\boldsymbol{\phi},\boldsymbol{\psi}} \lambda_1\mathcal{L}_{\mathrm{rec}}(\boldsymbol{\theta}, \boldsymbol{\phi}; \rho^{(i)}, \beta) + \lambda_2\mathcal{L}_{\mathrm{R}}(\boldsymbol{\theta}, \boldsymbol{\phi}; \rho^{(i)}) + \lambda_3\mathcal{L}_{\mathrm{L}}(\boldsymbol{\psi}, \boldsymbol{\phi}; \rho^{(i)}), \tag{2.6}$$

where λ_1, λ_2, and λ_3 are hyperparameters corresponding to the relative importance of each loss, and $\mathcal{L}_{\mathrm{rec}}$, $\mathcal{L}_{\mathrm{R}}$ and $\mathcal{L}_{\mathrm{L}}$ are given by Equations 2.1, 2.4 and 2.5, respectively.

2.3 Local Search

Local search represents a set of approaches to approximate a solution of a combinatorial optimization problem, the problem of maximizing or minimizing a function $f : \mathcal{R} \to \mathbb{R}$ defined on a search space [Pirlot, 1996]. In our formulation, a search space specifies a feasible set $\mathcal{R}$, domain of the optimization function, and a *neighborhood* function $N_K : \mathcal{R} \to \mathcal{R}^K$ which, given a feasible solution $x \in \mathcal{R}$, defines its K-neighborhood set $N_K(x)$.

Given a search space, a local search algorithm defines:

- a method for selecting the first candidate solution $x_1 \in \mathcal{R}$;

- in each search step $n = 1, 2, \ldots$, a method for selecting a new candidate solution x_{n+1} from the K-neighborhood of the current candidate x_n;
- a stopping condition.

The description implies the existence of a trivial local search algorithm. Such an algorithm starts by selecting an initial candidate arbitrarily or according to a distribution. Considering a maximization problem, it then selects a new candidate solution by looking at the solution x that yields the highest return in the function among the neighborhood of the current candidate $N_K(x_n)$. In other words, x_{n+1} is selected such as $f(x_{n+1}) \geq f(x) \forall x \in N_K(x_n)$. If there is no $x \in N_K(x_n)$ that yields a higher return in f than x_n, then the algorithm stops. This algorithm is often referred to as *hill climbing* in the literature.

Chapter 3

Programmatic Policies as a Local Search Problem

The goal of applying local search for programmatic policies is to find an approximate solution to Equation 1.1. Therefore, we are interested in finding a programmatic policy ρ that maximizes the expected episodic return $\mathbb{E}_{s_0 \sim S_0}[g(s_0, \rho)]$. In practice, we estimate the expectation by considering a set $\boldsymbol{s}_0$, where each element $s \in \boldsymbol{s}_0$ is sampled from S_0, and calculating

$$f_{\text{task}}(\rho) = \frac{1}{|\boldsymbol{s}_0|} \sum_{s \in \boldsymbol{s}_0} g(s, \rho), \tag{3.1}$$

which represents the optimization function we want to maximize, given a task's initial state distribution S_0 and return function g. We emphasize that this objective depends on the task that defines the POMDP with the subscript in f_{task}.

In order to formalize the local search procedure, we specify search spaces, including a feasible set $\mathcal{R} \subseteq \Pi_{\text{DSL}}$ and a corresponding neighborhood function $N_K : \mathcal{R} \to \mathcal{R}^K$, and local search algorithms.

3.1 Search Spaces for Programmatic Policies

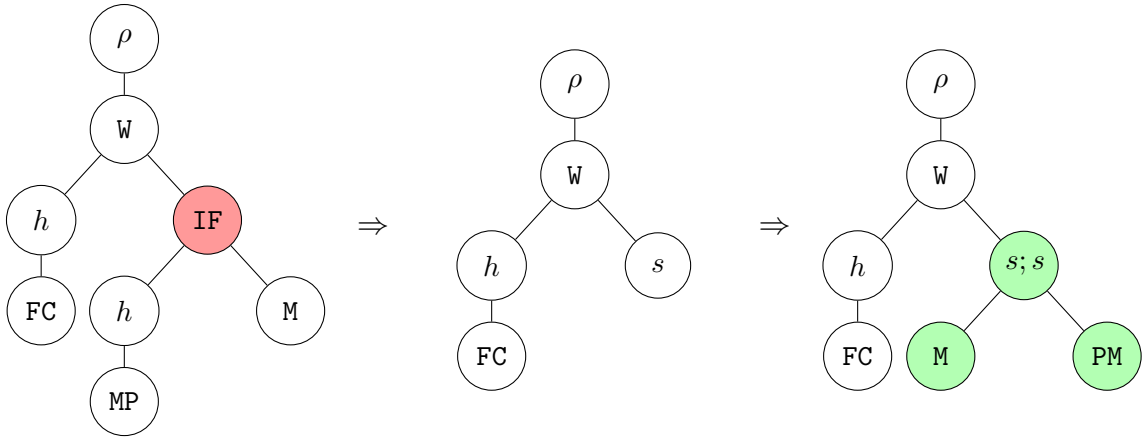
In this work, we evaluate two search spaces: Programmatic Space, which uses the DSL directly, and Latent Space, which uses a learned embedding of the DSL.

3.1.1 Programmatic Space

In this formulation, $\mathcal{R}^{\text{prog}}$ is a subset of the programs the DSL accepts. We define the subset $\mathcal{R}^{\text{prog}}$ 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 AST of every program. These constraints are identical to the constraints used on the distribution of programs used to train the autoencoder framework of LEAPS, as described in Appendix A.

The K-neighborhood of a program $\rho \in \mathcal{R}^{\text{prog}}$, $N_K^{\text{prog}}(\rho)$, consists of K samples of a *mutation* applied in ρ. A mutation is defined by uniformly sampling a node in the AST of ρ and deleting one of its children, also chosen from a uniform distribution. To replace the newly created non-terminal node, 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 LEAPS autoencoder. 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 ρ is a valid program. We ignore programs that are not in $\mathcal{R}^{\text{prog}}$ through a sample rejection scheme. That is, if the neighbor of ρ is not in $\mathcal{R}^{\text{prog}}$, we sample a new neighbor until we obtain one that is in $\mathcal{R}^{\text{prog}}$. We show a sample of the neighbor generation procedure in Figure 3.1, where the red node in the left-most AST represents the node selected for removal, and the green branch in the right-most AST represents the sampled sub-tree that replaces the original node.

For the probability distribution used to choose the DSL production rules in a program mutation, e.g. when choosing a rule to replace s in Figure 3.1(b), we adopt a fixed probability for each rule, described in Figure 3.2 as a probabilistic context-free grammar (PCFG). The adopted probabilities are exactly the same as the ones on the LEAPS project specifications [Trivedi et al., 2021].




```
DEF run m(
  WHILE c( FC c) w(
    IF c( MP c) i(
      move
    i)
  w)
m)
```

(a)

```
DEF run m(
  WHILE c( FC c) w(
    s
  w)
m)
```

(b)

```
DEF run m(
  WHILE c( FC c) w(
    move
    putMarker
  w)
m)
```

(c)

Figure 3.1: Example of a mutation in Programmatic Space, indicating the AST and text representation of the original program (a), the intermediary incomplete program (b), and the final mutated program (c). We use the abbreviations `FC` for `frontIsClear`, `MP` for `markersPresent`, `PM` for `putMarker`, and `M` for `move`.

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

Figure 3.2: Adopted probabilities for the Karel the Robot DSL as a probabilistic context-free grammar.

3.1.2 Latent Space

Given a trained autoencoder framework as described in Section 2.2, we define the Latent Space with the feasible set $\mathcal{R}^{\text{lat}} \subseteq \Pi_{\text{DSL}}$ as the set of all programs that $p_{\boldsymbol{\theta}}$, the underlying VAE decoder, can generate. Meanwhile, given a program $\rho \sim p_{\boldsymbol{\theta}}(\boldsymbol{z}) \in \mathcal{R}^{\text{lat}}$, $\boldsymbol{z} \in \mathbb{R}^d$, each program in its K-neighborhood $N_K^{\text{lat},\sigma}(\rho)$ is given by decoding $\boldsymbol{z}+\epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma \boldsymbol{I}_d)$ and $\boldsymbol{I}_d$ represents the $d \times d$ identity matrix. In this space, we do not set a sample rejection method to restrict the programs one can decode from the model.

The standard deviation of the noise σ is a hyperparameter of the Latent Space, along with the hyperparameters that describe the model itself, as outlined in Section 2.2.

3.2 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 approximately maximizes the objective of Equation 3.1. In addition to CEM, we also consider cross-entropy beam search (CEBS), a method inspired in CEM that retains information about the best candidate solutions from a population. We also consider 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.

Hill-Climbing (HC) This algorithm starts by sampling a candidate solution, using the PCFG from Figure 3.2 in the case of a search in the Programmatic Space, or a vector from the distribution $\mathcal{N}(0, \boldsymbol{I}_d)$ 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

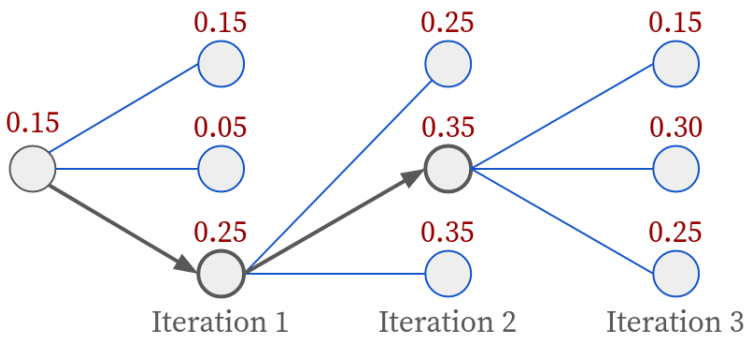


Figure 3.3: Diagram representing an example execution of HC. Nodes represent policies, edges show neighborhood relations, and numbers are the estimated episodic return of each policy. Nodes with a bold contour are selected as the candidate of each iteration. In iteration 3, as no policy yields a higher return than the current candidate, the algorithm stops and outputs the policy that yields a 0.35 episodic return.

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. An implementation of this procedure is provided in Algorithm 1 and an example of its execution is represented in the diagram of Figure 3.3.

Algorithm 1 Hill-climbing for Programmatic Policies

Require: N, the neighborhood function; K, the neighborhood size; P_0, the initial program distribution; f_{task}: the function that estimates the expected episodic return in the given task.

Ensure: $\bar{\rho}$, best-seen program with respect to highest episodic return estimate; $\bar{g}$, estimated episodic return of best-seen program.

$\bar{\rho} \sim P_0$
$\bar{g} \leftarrow f_{\text{task}}(\bar{\rho})$
repeat
    in_local_maximum $\leftarrow$ true
    for each ρ in $N_K(\bar{\rho})$ **do**
        if $f_{\text{task}}(\rho) > \bar{g}$ **then**
            $\bar{\rho} \leftarrow \rho$
            $\bar{g} \leftarrow f_{\text{task}}(\rho)$
            in_local_maximum $\leftarrow$ false
until in_local_maximum = true

Cross-Entropy Method (CEM) CEM generates a set of K candidate solutions from the K-neighborhood of an initial candidate, whose latent vector is sampled 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 the mean of the top E episodic returns is unchanged or lower, as described in Algorithm 2. An example of a CEM execution is shown in Figure 3.4.

Algorithm 2 Cross-Entropy Method for Programmatic Policies in LATENT SPACES

Require: N^{lat}, the neighborhood function; K, the neighborhood size; E, the size of the beam; σ, the noise parameter for the LATENT SPACE; P_0, the initial program distribution; f_{task}: the function that estimates the expected episodic return in the given task.

Ensure: $\bar{\rho}$, best-seen program with respect to highest episodic return estimate; $\bar{g}$, estimated episodic return of best-seen program.

$\bar{\rho} \sim P_0$
$\bar{g} \leftarrow f_{\text{task}}(\bar{\rho})$
best_mean_elite_return $\leftarrow -\infty$
(latent_vectors, candidates) $\leftarrow N_K^{\text{lat},\sigma}(\bar{\rho})$ ▷ The neighborhood function, in the case of LATENT SPACE, returns the latent vectors used to decode the candidates in a tuple.
repeat
    in_local_maximum $\leftarrow$ true
    $\mathbf{g} \leftarrow []$
    for each ρ in candidates **do**
        $\mathbf{g}$.append($f_{\text{task}}(\rho)$)
        if $f_{\text{task}}(\rho) > \bar{g}$ **then**
            $\bar{\rho} \leftarrow \rho$
            $\bar{g} \leftarrow f_{\text{task}}(\rho)$
    elite_indices $\leftarrow$ argtop-$E(\mathbf{g})$
    elite_estimated_return $\leftarrow \frac{1}{E} \sum_{i \in \text{elite\_indices}} \mathbf{g}[i]$
    if elite_estimated_return $>$ best_mean_elite_return **then**
        best_mean_elite_return $\leftarrow$ elite_estimated_return
        in_local_maximum $\leftarrow$ false
    mean_elite_vector $\leftarrow \frac{1}{E} \sum_{i \in \text{elite\_indices}}$ latent_vectors$[i]$
    (latent_vectors, candidates) $\leftarrow N_K^{\text{lat},\sigma}$(mean_elite_vector) ▷ For simplicity, we use the same notation of neighborhood function of a program to indicate the neighborhood around a latent vector. This is done to skip decoding and re-encoding the vector in the VAE.
until in_local_maximum = true

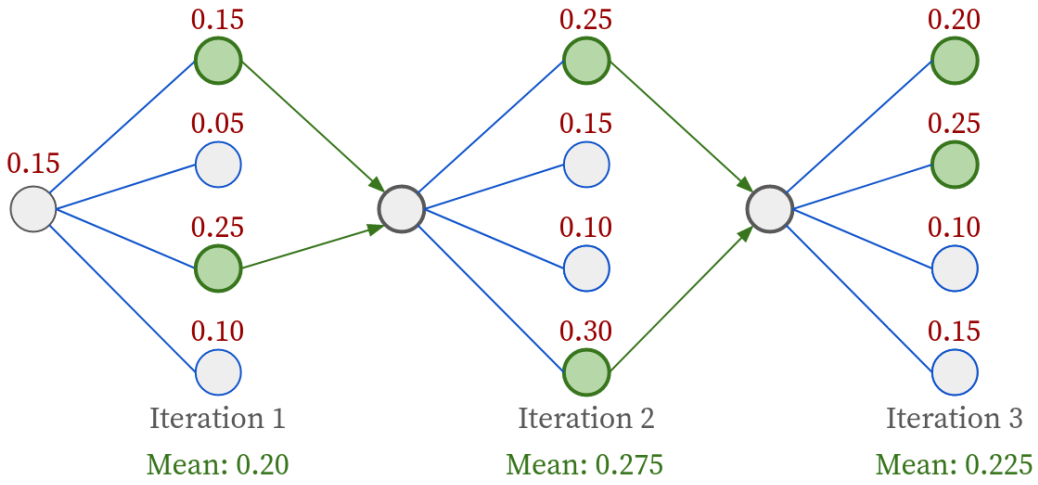


Figure 3.4: Diagram representing an example execution of CEM. Nodes represent policies, edges show neighborhood relations, and numbers are the estimated episodic return of each policy. Nodes in green represent the policies that yield the top-E episodic return in that iteration, whose latent vectors are averaged as the candidate policy, depicted as a node with a bold contour in gray. In iteration 3, as the mean return of the selected policies is lower than the previous iteration, the algorithm stops.

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. Note that, in practice, the only difference to CEM is how CEBS generates its next candidate from the list of top E vectors from the candidates, as shown in line 18 of the implementation provided in Algorithm 3. The diagram of Figure 3.5 represents an example execution of CEBS.

3.3 Topology Metrics

Aside from finding policies that maximize our objective, we are interested in measuring how conducive the search spaces are to local search. We do so by analyzing the properties of the average search path induced by each neighborhood function, and

Algorithm 3 Cross-Entropy Beam Search for Programmatic Policies in Latent Spaces

Require: N^{lat}, the neighborhood function; K, the neighborhood size; E, the size of the beam; σ, the noise parameter for the Latent Space; P_0, the initial program distribution; f_{task}: the function that estimates the expected episodic return in the given task.

Ensure: $\bar{\rho}$, best-seen program with respect to highest episodic return estimate; $\bar{g}$, estimated episodic return of best-seen program.

$\bar{\rho} \sim P_0$
$\bar{g} \leftarrow f_{\text{task}}(\rho)(\bar{\rho})$
best_mean_elite_return $\leftarrow -\infty$
candidates $\leftarrow N_K^{\text{lat},\sigma}(\bar{\rho})$
repeat
    in_local_maximum $\leftarrow$ true
    $\mathbf{g} \leftarrow []$
    for each ρ in candidates **do**
        $\mathbf{g}$.append($f_{\text{task}}(\rho)$)
        if $f_{\text{task}}(\rho) > \bar{g}$ **then**
            $\bar{\rho} \leftarrow \rho$
            $\bar{g} \leftarrow f_{\text{task}}(\rho)$
    elite_indices $\leftarrow$ argtop-$E(\mathbf{g})$
    elite_estimated_return $\leftarrow \frac{1}{E} \sum_{i \in \text{elite\_indices}} \mathbf{g}[i]$
    if elite_estimated_return $>$ best_mean_elite_return **then**
        best_mean_elite_return $\leftarrow$ elite_estimated_return
        in_local_maximum $\leftarrow$ false
    candidates $\leftarrow \bigcup_{i \in \text{elite\_indices}} N_{K/E}^{\text{lat},\sigma}(\rho[i])$ ▷ Aggregates as a K-neighborhood of the elite.
until in_local_maximum = true

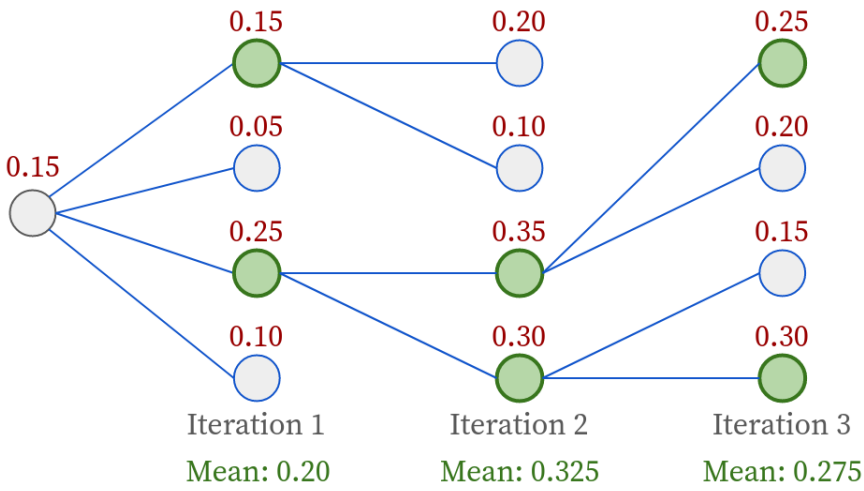


Figure 3.5: Diagram representing an example execution of CEBS. Nodes represent policies, edges show neighborhood relations, and numbers are the estimated episodic return of each policy. Nodes in green represent the policies that yield the top-E episodic return in that iteration (in this example, $E = 2$). In iteration 3, as the mean return of the selected policies is lower than the previous iteration, the algorithm stops.

by measuring how likely an arbitrary search algorithm finds a solution in each search space. We formalize these metrics in this section.

3.3.1 Local Behavior Similarity

We use the concept of behavior similarity presented by LEAPS in Equation 2.3 as ρ-similarity to define similarity along an average search path of a search space. 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

$$\text{behavior-similarity}(N_1, n_{\text{mutations}}) = \mathbb{E}_{\rho_0 \sim P_0, s_0 \sim S_0}[\rho\text{-similarity}(\rho_0, \rho_{n_{\text{mutations}}}, s_0)], \tag{3.2}$$

where $\rho_{n_{\text{mutations}}}$ is the single individual of the neighborhood function N_1 (N_K with $K = 1$) recursively applied $n_{\text{mutations}}$ times in ρ_0, thus producing a path in the underlying search graph.

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

$$\text{identity-rate}(N_1, n_{\text{mutations}}) = \mathbb{E}_{\rho_0 \sim P_0, s_0 \sim S_0}[\mathbb{1}\{\rho_0 = \rho_{n_{\text{mutations}}}\}]\,. \tag{3.3}$$

3.3.2 Convergence Rate

To isolate the search space analysis in the local search procedure, we fix an arbitrary local search algorithm and measure how likely the search is to find a solution of a given quality in the search space of interest. This measurement gives us an estimate of how conducive the search space is to local search.

We define the convergence rate of a search space given by the neighborhood function N_K, with initial program distribution P_0 and a task's return estimation function f_{task} (Equation 3.1). The convergence rate is measured in terms of $g_{\text{target}} \in [0, 1]$ by computing

$$\text{convergence-rate}(N_K, g_{\text{target}}) = \mathbb{E}_{\rho_0 \sim P_0}[\mathbb{1}\{f_{\text{task}}(\rho^*) \geq g_{\text{target}}\}], \tag{3.4}$$

where ρ^* is the best-performing program returned by applying a specified search algorithm in the space defined by N_K starting on ρ_0, and K is the neighborhood size.

Note that the convergence rate measure is related to the performance of local search in finding a high return policy. If the convergence rate, specified by a search algorithm, is higher in a search space, we can expect this space to yield high return policies using the same search algorithm in fewer search iterations.

Chapter 4

Empirical Results

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, as described in Section 3.3. We start by introducing Karel the Robot, the domain in which we carried out our experiments. In all instances of the Latent Space, we use the model trained by the LEAPS authors with training procedure and hyperparameters described in Appendix A.

4.1 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 presented

in Section 2.1. This DSL, introduced by the work of Bunel et al. [2018], 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 {`front`|`left`|`right`}`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`{`Left`|`Right`}, and interact with the markers with {`put`|`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 B.

4.2 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 in the PROGRAMMATIC SPACE and CEBS in the LATENT 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 original papers.

In this experiment, we consider a set of 16 initial states for the estimation of Equation 3.1. 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 4.1 summarizes our results in the KAREL and KAREL-HARD problem sets, comparing them to the results reported by the authors of LEAPS and HPRL. In order to better outline the performance of each algorithm, we plot the reached episodic return as a function of the number of episodes in Figure 4.1[1] and analyze differences in running time in Appendix C. We also present representative examples of programs that HC and CEBS synthesize in Appendix D.

	LATENT			PROGRAMMATIC
Task	LEAPS	HPRL	CEBS	HC
STAIRCLIMBER	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00
MAZE	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00
TOPOFF	0.81 $\pm$ 0.07	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00
FOURCORNERS	0.45 $\pm$ 0.25	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00
HARVESTER	0.70 $\pm$ 0.02	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00
CLEANHOUSE	0.32 $\pm$ 0.01	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00
DOORKEY	0.50 $\pm$ 0.00	0.50 $\pm$ 0.00	0.50 $\pm$ 0.00	**0.84** $\pm$ 0.02
ONESTROKE	0.65 $\pm$ 0.14	0.80 $\pm$ 0.02	0.90 $\pm$ 0.00	**0.95** $\pm$ 0.00
SEEDER	0.51 $\pm$ 0.03	0.58 $\pm$ 0.06	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00
SNAKE	0.23 $\pm$ 0.06	0.33 $\pm$ 0.07	0.26 $\pm$ 0.01	**0.65** $\pm$ 0.03

Table 4.1: 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.

We see that HC, based on PROGRAMMATIC SPACE, achieves the highest episodic

[1]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.

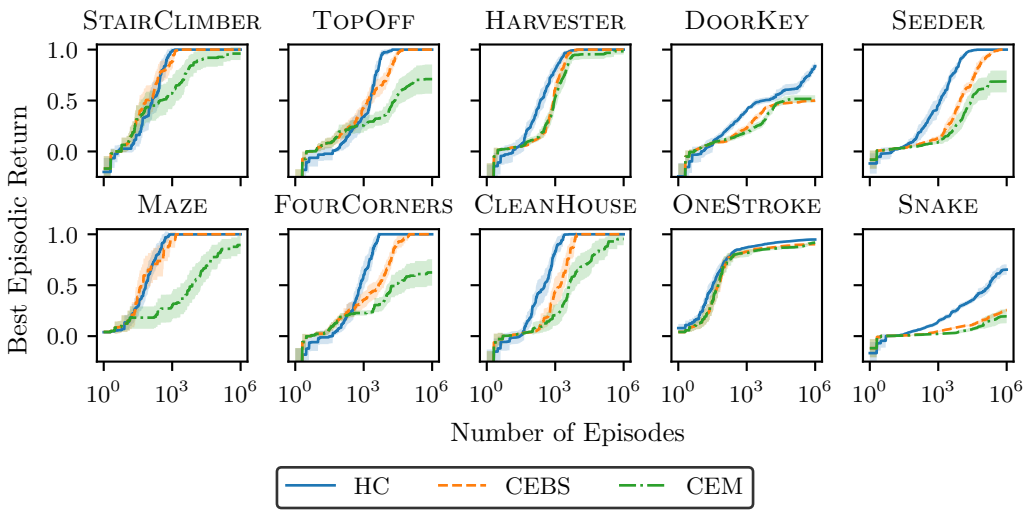


Figure 4.1: Episodic return performance of all methods in Karel and Karel-Hard problem sets. Reported mean and 95% confidence interval over 32 executions. In this Figure, CEM results are based on our implementation of the search algorithm. The x-axis is represented in a log scale for better visualization.

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 in Table 4.1 in the DoorKey task. 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.

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 trajectories are stricter. Figure 4.2 outlines the episodic return performance of CEM, CEBS, and HC on the CRASHABLE version of every task of the problem sets.

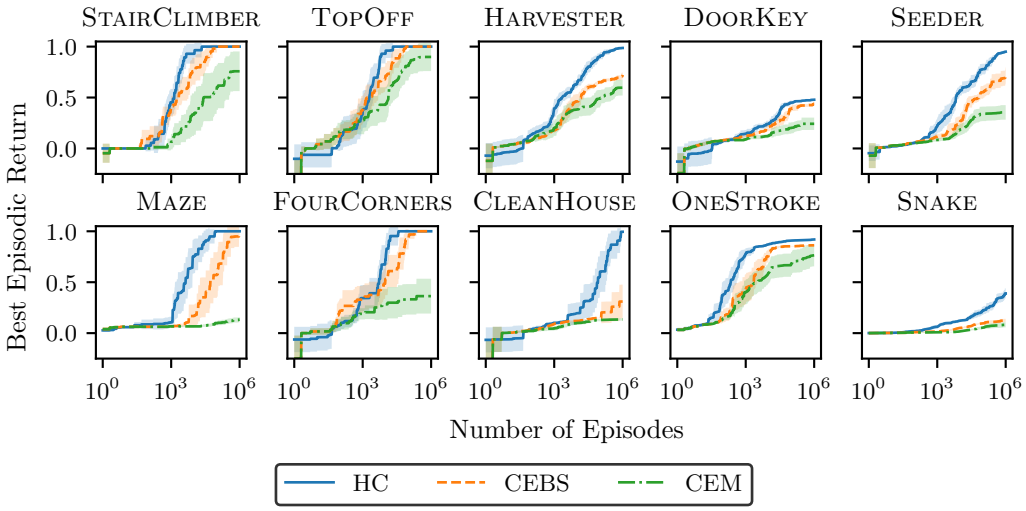


Figure 4.2: Episodic return performance of all methods in KAREL and KAREL-HARD problem sets, using the CRASHABLE version of the environment, that does not allow invalid actions. Reported mean and 95% confidence interval over 16 seeds. The x-axis is represented in a log scale for better visualization.

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.

To observe the impact of the initialization methods of the search algorithms, we evaluate a second version of all search algorithms using the initialization rule of the opposed search space. For CEM and CEBS, this version of the algorithms initializes the search with policies sampled from the defined PCFG. And for HC in PROGRAMMATIC SPACE, search is initialized by decoding a latent sampled from $\mathcal{N}(0, \boldsymbol{I}_d)$ in

the Latent Space. As presented and detailed in Appendix E, the episodic return values obtained by searching with these proposed versions of the algorithms were very similar to the values obtained by the original algorithms. This suggests that both initialization methods are similar and one does not provide a significant advantage over the other.

4.3 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, as described in Section 3.3.

4.3.1 Local Behavior Similarity Analysis

To estimate the metrics given by Equations 3.2 and 3.3, we sample a set of 32 initial states from a distribution S_0 and a set of 1000 initial programs from P_0. S_0 is composed of random Karel maps unrelated to any task in the problem sets, generated by the same procedure considered while training LEAPS (described in Appendix A), and we set P_0 differently for each search space. For Programmatic Space, it is given by the PCFG from Figure 3.2, and for Latent Space, it is given by $\mathcal{N}(0, \boldsymbol{I}_d)$. 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 4.3.

Although this was the objective of constructing the Latent Space, we see that the Programmatic Space achieves comparable behavior-similarity metrics. We also see that although the setting $\sigma = 0.1$ on Latent Space achieves high behavior-similarity, it is not more conducive to search, which is likely due to its higher identity-rate.

The observed result still does not provide evidence of the performance discrepancy

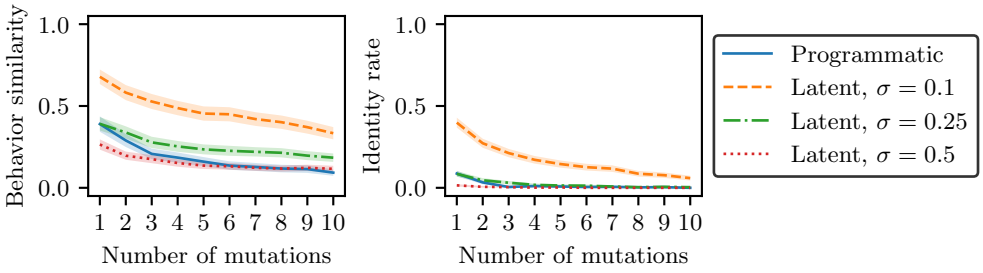


Figure 4.3: Behavior-similarity and identity-rate metrics on Programmatic Space and Latent Space ($\sigma = \{0.1, 0.25, 0.5\}$). Reported mean and 95% confidence interval of the estimation of each metric over a set of 32 initial states of the environment and 1000 initial programs.

that we observe while searching for policies in the programmatic and in the latent spaces.

4.3.2 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 use the Equation 3.4 to estimate convergence rate on a given task and search space, guided by HC. Similarly, the initial program distribution for the Programmatic Space is given by generating a program using the PCFG from Figure 3.2, and for the Latent Space, it is given by the programs one decodes after sampling a latent vector from $\mathcal{N}(0, \boldsymbol{I}_d)$.

As the HC return depends on the task we are solving, we estimate different values from Equation 3.4 for each task. The estimation involves sampling a set of 32 states given by the task's initial state distribution, and sampling $10,000$ initial programs to start each execution of HC. The estimation of convergence-rate of Programmatic Space and Latent Space[2], both set to a neighborhood size $K = 250$, for every

[2]Here, σ follows the values that the LEAPS authors chose for each task: $\sigma = 0.5$ for Four-

task in our problem sets is shown in Figure 4.4 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 rates. Table 4.1 and Figure 4.1 show that the search in the Programmatic Space achieves episodic return values larger than 0.5 for DoorKey and Snake; the zoomed-in regions in Figure 4.4 show that these are rare events, but possible to be observed with a reasonable search budget.

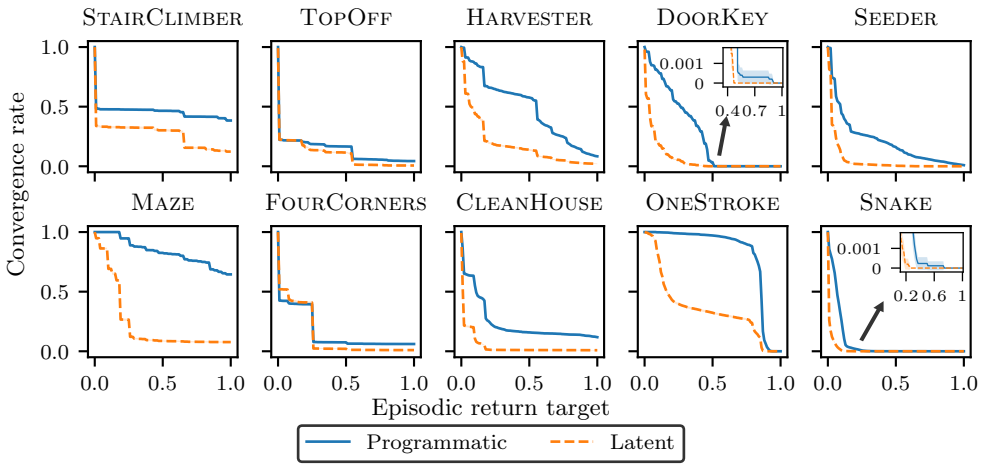


Figure 4.4: Convergence rate of Programmatic Space and Latent Space with neighborhood size $K = 250$, guided by hill-climbing. Reported mean and 95% confidence interval of estimation over a set of $10,000$ search initializations.

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, when considering algorithms not equipped with mechanisms for escaping local optima.

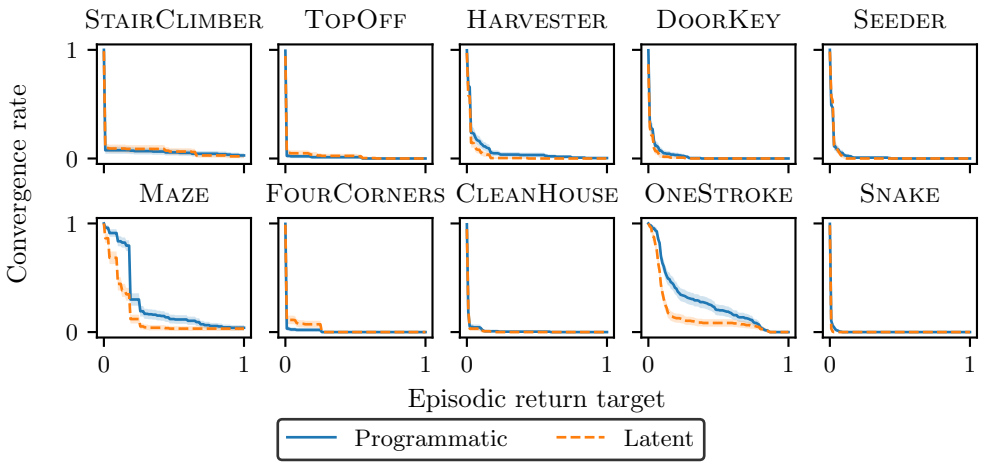
It is important to note that the convergence rate values of zero for episodic return target larger than 0.5 in DoorKey for the Latent Space do not imply that achieving policies with that return is impossible in the space. The analysis shows that HC did not find those policies after testing $10,000$ different search initializations, but we show that they exist with the following experiment: we encode a policy that yields an

Corners and Harvester, $\sigma = 0.1$ for Maze, and $\sigma = 0.25$ for all other tasks.

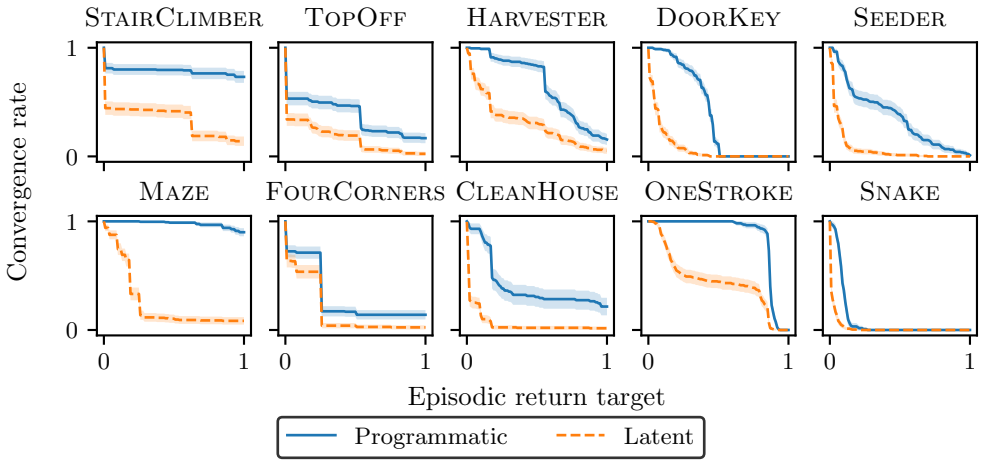
episodic return larger than 0.5 as the initial candidate of a HC search in the Latent Space, and report the best-performing policy after the algorithm stopping condition. This process, when done with a particular policy, resulted in a policy that yields a 0.72 episodic return, decoded from the Latent Space. More details about this experiment are present in Appendix F.

We expand the convergence rate analysis by adopting neighborhood functions N_K with different neighborhood sizes K in Equation 3.4. The convergence rate analysis on a space given by a lower K indicates how robust would a search process be in the search space, given that it relies on a small number of samples. On the other hand, the result of the 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 4.5 compares the convergence-rate estimation of both Programmatic and Latent Space adopting $K = \{10, 1000\}$, 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 $K = 1,000$.

Finally, we further analyze the convergence rate of the Latent Space by using CEM and CEBS as the search algorithm that produces ρ^* in Equation 3.4. Figure 4.6 compares the convergence rate obtained with HC (original setting), CEM, and CEBS, with neighborhood size $K = 64$. Although CEM and HC have a similar convergence rate across all tasks, we see that CEBS outperforms both CEM and HC 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 4.1).



(a) $K = 10$



(b) $K = 1,000$

Figure 4.5: Convergence rate of Programmatic Space and Latent Space with neighborhood sizes $K = 10$ (a) and $K = 1,000$ (b), guided by hill-climbing. Reported mean and 95% confidence interval of estimation over a set of 250 search initializations.

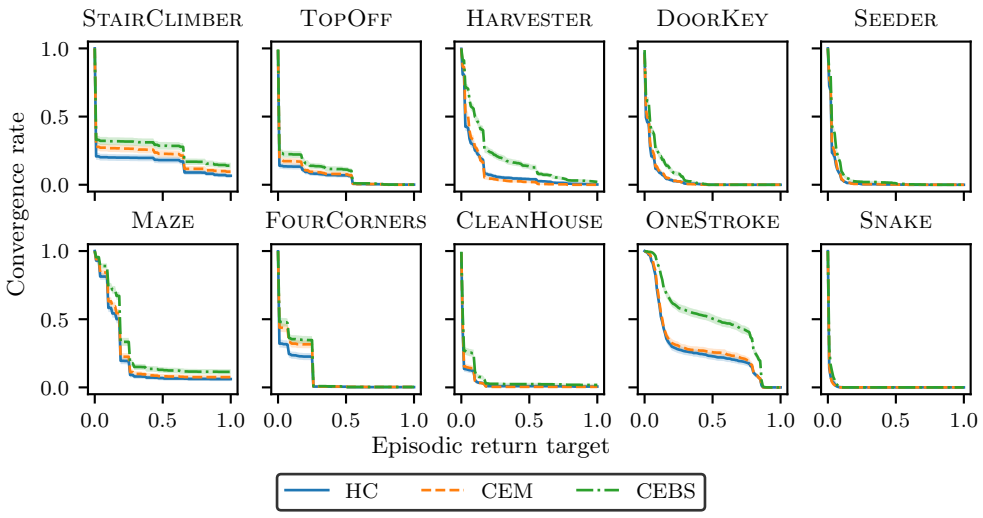


Figure 4.6: Convergence rate of Latent Space with neighborhood size $K = 64$, guided by HC, CEM, and CEBS. Reported mean and 95% confidence interval over $1,000$ seeds.

Chapter 5

Related Works

Most of the early work on programmatic policies considered stateless programs, such as decision trees. For example, Verma et al. [2018] and Verma et al. [2019] learn tree-like programs with no internal states. Bastani et al. [2018] use imitation learning to induce decision tree encoding policies. Qiu and 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. 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, Trivedi et al. [2021] and 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 [Medeiros et al., 2022, Aleixo and 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 work [Waldinger and Lee, 1969, Solar-Lezama et al., 2006], where one must search in the programmatic spaces for a program that satisfies the user's intent. In this case, a user's intent is a specification of the solution, usually given as input-output examples or as a natural language description. These problems can be solved with brute-force search [Udupa et al., 2013, Albarghouthi et al., 2013] or with algorithms that are

guided by a function, which is often learned [Odena et al., 2021, Barke et al., 2020, Shi et al., 2022, Ellis et al., 2020, Wong et al., 2021, Ameen and Lelis, 2023]. 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.

Constructing a latent space as a search space for combinatorial search has been attempted in other problems throughout the literature. Hottung et al. [2020] construct a latent space that encodes solutions for routing problems, and apply differential evolution to search for solutions in it. The authors show that the method finds solutions with low optimality gap in smaller execution time, when compared to a baseline that applies REINFORCE on a model with attention layers [Kool et al., 2018]. Lynch et al. [2020] train a VAE that reconstructs programs in program synthesis problems, as opposed to reinforcement learning, and compare an evolutionary algorithm in latent space with genetic algorithms in the original programmatic space. The authors show that searching in the latent space is competitive with searching in the programmatic space, but the latter presents higher success rates in problems with discontinuous optimization landscapes. These results are consistent with our findings in this work. Liskowski et al. [2020] also construct a latent space for program synthesis problems with VAEs, but show a higher success rate for an evolutionary continuous search in latent space when compared to a genetic programming approach. Although this is not consistent with our findings, we conjecture that this is due to the difficulty of the problem. In program synthesis, we are given specifications about solutions and can use that information to guide the process of finding a solution, analogue to a supervised learning problem. We conjecture that it is easier to learn latent spaces that are conducive to search to supervised learning tasks than to reinforcement learning tasks.

Chapter 6

Conclusions

In our work, 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 the original programmatic space was an important missing baseline in previous work and that learning latent spaces for programming languages is still an open and challenging research question.

6.1 Future Work

We expect that a latent representation is still necessary to effectively find programmatic policies in programmatic spaces composed of languages more expressive than Karel DSL. We hypothesize that constructing a latent space that achieves a high convergence rate requires compression of the programmatic space. Combining com-

pression with high behavior similarity could aid search procedures to effectively find policies with a desired behavior, as neighborhood sets of a candidate policy would include a set of solutions with more diverse behavior.

This work compares programmatic representations within a single POMDP setting. We highlight that there is no current research on more general settings, including the ones commonly used in deep reinforcement learning, such as discounted reward functions in episodic and continuing tasks, and stochastic dynamics and policies. We see that, as a direction for future work, it is important to consider these settings to compare representations for programmatic policies more extensively.

References

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

David S. Aleixo and Levi H. S. Lelis. Show me the way! Bilevel search for synthesizing programmatic strategies. In *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press, 2023.

Saqib Ameen and Levi H. S. Lelis. Program synthesis with best-first bottom-up search. *Journal of Artificial Intelligence Research*, 77:1275–1310, 2023. doi: 10.1613/jair.1.14394.

Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4 (OOPSLA):1–29, 2020.

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. Curran Associates Inc., 2018.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Luc Cary, Lucas Morales, Luke B. Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *CoRR*, abs/2006.08381, 2020. URL https://arxiv.org/abs/2006.08381.

Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. In *International conference on learning representations*, 2016.

André Hottung, Bhanu Bhandari, and Kevin Tierney. Learning a latent search space for routing problems using variational autoencoders. In *International Conference on Learning Representations*, 2020.

Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. Synthesizing programmatic policies that inductively generalize. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=S1l8oANFDH.

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

Diederik P Kingma, Max Welling, et al. An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12(4):307–392, 2019.

Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

Paweł Liskowski, Krzysztof Krawiec, Nihat Engin Toklu, and Jerry Swan. Program synthesis as latent continuous optimization: Evolutionary search in neural embeddings. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pages 359–367, 2020.

Guan-Ting Liu, En-Pei Hu, Pu-Jen Cheng, Hung-Yi Lee, and Shao-Hua Sun. Hierarchical programmatic reinforcement learning via learning to compose programs. *arXiv preprint arXiv:2301.12950*, 2023.

David Lynch, James McDermott, and Michael O'Neill. Program synthesis in a continuous space using grammars and variational autoencoders. In *Parallel Problem Solving from Nature–PPSN XVI: 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part II 16*, pages 33–47. Springer, 2020.

Leandro C. Medeiros, David S. Aleixo, and Levi H. S. Lelis. What can we learn even from the weakest? Learning sketches for programmatic strategies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 7761–7769. AAAI Press, 2022.

Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. BUSTLE: Bottom-up program synthesis through learning-guided exploration. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=yHeg4PbFHh.

Spyros Orfanos and Levi H. S. Lelis. Synthesizing programmatic policies with actor-critic algorithms and relu networks, 2023.

Richard E Pattis. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, 1994.

Marc Pirlot. General local search methods. *European journal of operational research*, 92(3):493–511, 1996.

Wenjie Qiu and He Zhu. Programmatic reinforcement learning without oracles. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=6Tk2noBdvxt.

Reuven Y. Rubinstein. *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning*. Springer Science & Business Media, 1999.

Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. Crossbeam: Learning to search in bottom-up program synthesis. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=qhC8mr2LEKq.

Eui Chul Shin, Illia Polosukhin, and Dawn Song. Improving neural program synthesis with inferred execution traces. *Advances in Neural Information Processing Systems*, 31, 2018.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 404–415, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934510. doi: 10.1145/1168857.1168907. URL https://doi.org/10.1145/1168857.1168907.

Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim. Learning to synthesize programs as interpretable and generalizable policies. *Advances in neural information processing systems*, 34:25146–25163, 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 *International Conference on Machine Learning*, 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.

Richard J. Waldinger and Richard C. T. Lee. Prow: A step toward automatic program writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, page 241–252, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

Catherine Wong, Kevin M Ellis, Joshua Tenenbaum, and Jacob Andreas. Leveraging language to learn program abstractions and search heuristics. In *International Conference on Machine Learning*, pages 11193–11204. PMLR, 2021.

Appendix A: Training details of LEAPS

The Latent Space of this work is based on the trained model provided by the LEAPS authors Trivedi et al. [2021]. The model training involves generating a dataset of programs in Karel DSL, and, for each program, a set of trajectories following the rules of the environment.

To compose the program dataset, the authors start by sampling a program from the PCFG from Figure 3.2. This program sample will only compose the dataset if it suits the following restrictions:

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

Generating trajectories for a program involves generating an initial state where the program is executed. This is generated as a random Karel map unrelated to any task by following the procedure: Starting from an empty 8×8 map, walls are placed with a chance of 10% on each cell. Afterward, the agent is placed in a uniformly random coordinate on the map, which is re-sampled in case this lands on a wall. Finally, a single marker is placed on each empty cell with a probability of 10%.

To train the model used in the final work, the authors generated $50,000$ programs and 10 trajectories for each program, following the procedure described above. The

dataset was split into $35,000$ programs for training, $7,500$ programs for validation, and $7,500$ programs for testing. The authors then set $d = 256$ as the dimension of the latent vectors to specify the model architecture. They used the hyperparameters $\beta = 0.1$, $\lambda_1 = 1$, $\lambda_2 = 1$ and $\lambda_3 = 1$ to train the model using Equation 2.6.

Appendix B: Karel problem sets

In this Section, we specify the initial state distribution and episodic 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.

B.1 Karel

StairClimber This environment is given by a 12×12 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 8×8 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.

TopOff Markers are placed randomly on the bottom row of an empty 12×12 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.

FourCorners Starting on a random cell on the bottom row of an empty 12×12 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 8×8 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.

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.

B.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.

Appendix C: 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 C.1. We see that sampling from the programmatic space is more than 10 times faster than sampling from the latent space.

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 C.1. We see that sampling from the programmatic space is more than 10 times faster than sampling from the latent space.

	Programmatic Space	Latent Space
Elapsed time (seconds)	0.0021 ± 0.0002	0.0293 ± 0.0004

Table C.1: Average time for generating one neighbor in Programmatic Space and Latent Space from a given candidate, measured over $1,000$ initial random candidates. Reported mean and 95% confidence interval.

Appendix D: 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 D.1 and D.2 for HC and CEBS, respectively.

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 D.1: Representative high-return solutions from HC search in the Programmatic Space.

Task	Solution	Return
Harvester	`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)`	1.0
CleanHouse	`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)`	1.0
DoorKey	`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)`	0.6875
OneStroke	`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)`	0.9288
Seeder	`DEF run m( WHILE c( noMarkersPresent c) w( putMarker move move WHILE c( markersPresent c) w( turnRight move w) w) m)`	1.0
Snake	`DEF run m( WHILE c( noMarkersPresent c) w( REPEAT R=13 r( IFELSE 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)`	0.4375

Table D.2: Representative high-return solutions from CEBS in Latent Space.

Appendix E: Evaluating the impact of initialization methods

To measure the impact of the initialization methods of the search algorithms, we evaluate an alternative version of all search algorithms using the initialization rule of the opposed search space. For CEM and CEBS, this version of the algorithms initializes the search with policies sampled from the defined PCFG. And for HC in Programmatic Space, search is initialized by decoding a latent sampled from $\mathcal{N}(0, \boldsymbol{I}_d)$ in the Latent Space. Figure E.1 compares the performance of HC in Programmatic Space with its alternative version, while Figures E.2 and E.3 compare the performance of CEM and CEBS, respectively, with their alternative versions.

Results show that the performance of the alternative version of the algorithms was very similar to the original algorithm in most cases, and marginally inferior in others. This suggests that both initialization methods are similar and one does not provide a significant advantage over the other.

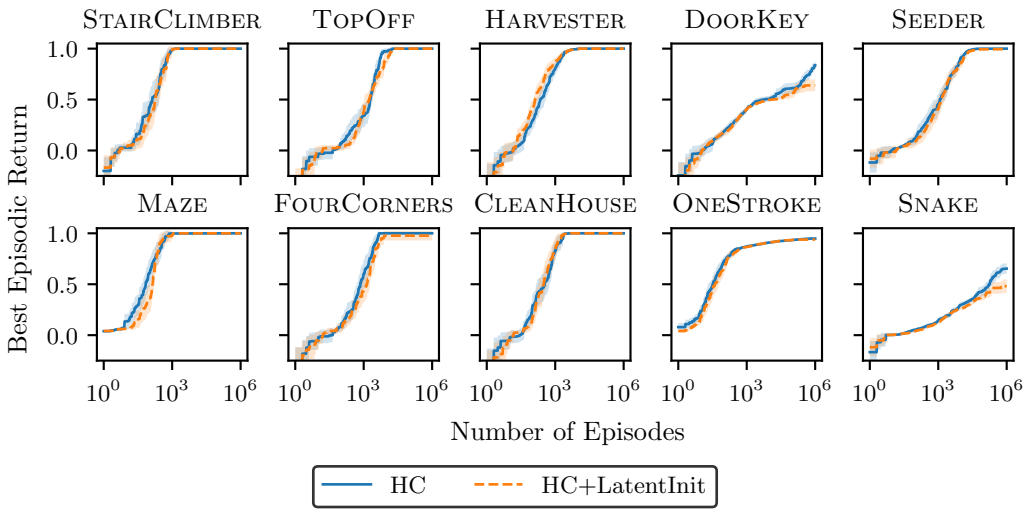


Figure E.1: Episodic return performance of HC in Programmatic Space with original and latent initialization in Karel and Karel-Hard problem sets. Reported mean and 95% confidence interval over 32 seeds. The x-axis is represented in a log scale for better visualization.

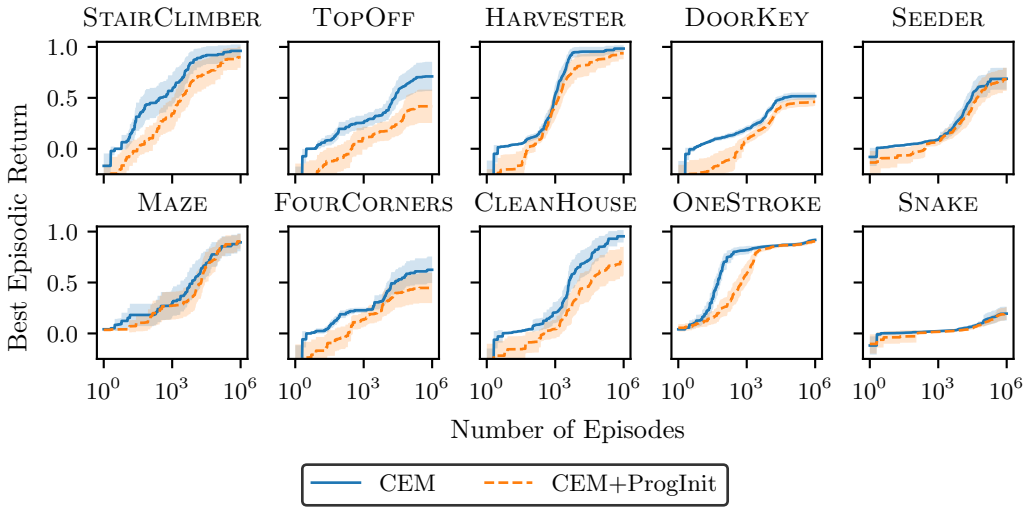


Figure E.2: Episodic return performance of CEM in Latent Space with original and programmatic initialization in Karel and Karel-Hard problem sets. Reported mean and 95% confidence interval over 32 seeds. The x-axis is represented in a log scale for better visualization.

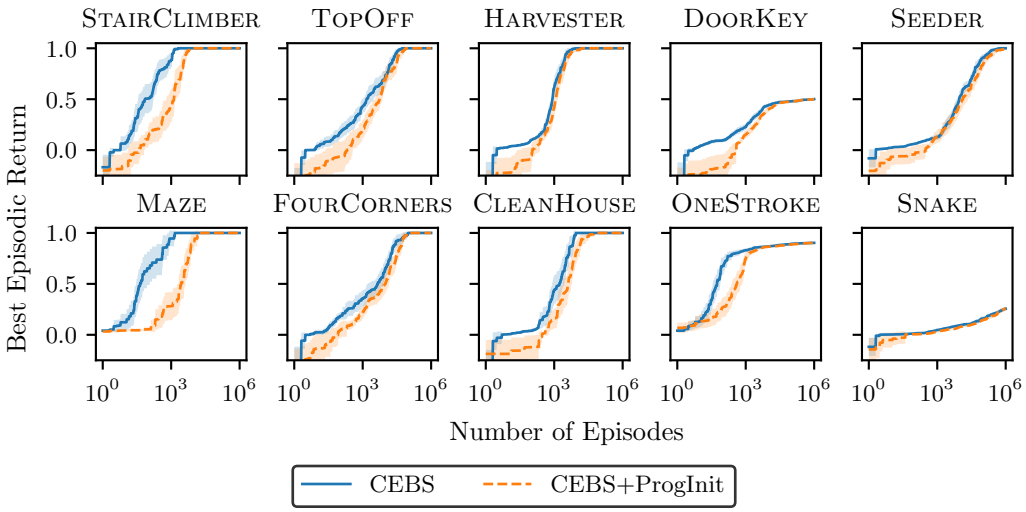


Figure E.3: Episodic return performance of CEBS in Latent Space with original and programmatic initialization in Karel and Karel-Hard problem sets. Reported mean and 95% confidence interval over 32 seeds. The x-axis is represented in a log scale for better visualization.

Appendix F: Proving the existence of high-performing policies in Latent Space

To complement the performance and topology analysis of the Latent Space, we designed an additional experiment to validate the existence of high-performing policies in the space. Specifically, we are interested in confirming if policies with an episodic return larger than 0.5 in the DoorKey task exist in the Latent Space.

We start by selecting a high-performing policy found by searching in the Programmatic Space. This policy is then encoded as a latent vector to be represented in the Latent Space. Note that the encoded policy does not necessarily decode to the same original policy. To account for that, we use the encoded policy as the initial candidate for a HC search on the Latent Space with neighborhood size $K = 1,000$. Table F.1 shows a particular choice of initial candidate that led to the discovery of a policy in the Latent Space that yields an episodic return of 0.71875 in DoorKey.

This experiment confirms that high-performing policies exist in the Latent Space. However, the convergence rate analysis shows that searching with HC in the space did not result in such policies, after considering $10,000$ different search initializations.

	Policy	DoorKey return
Original initial candidate	`DEF run m( WHILE c( noMarkersPresent c) w( turnLeft pickMarker move move w) pickMarker move WHILE c( noMarkersPresent c) w( REPEAT R=17 r( move r) move turnLeft move w) putMarker m)`	0.9375
Decoded initial candidate	`DEF run m( WHILE c( noMarkersPresent c) w( turnLeft pickMarker move w) move move REPEAT R=3 r( move r) putMarker turnLeft m)`	-0.5
Search result	`DEF run m( WHILE c( noMarkersPresent c) w( turnLeft move move w) pickMarker move WHILE c( noMarkersPresent c) w( turnLeft move w) WHILE c( rightIsClear c) w( putMarker w) m)`	0.71875

Table F.1: Initial candidate and resulting program from searching for a high-performing policy in Latent Space for the DoorKey task. The original initial candidate is defined in the Programmatic Space, while the decoded initial candidate and the search result are defined in the Latent Space.