Searching for Programmatic Policies in Semantic Spaces

Rubens O. Moraes[1] and **Levi H. S. Lelis**[2,3]

[1] Departamento de Informática, Universidade Federal de Viçosa, Brazil
[2]Department of Computing Science, University of Alberta, Canada
[3]Alberta Machine Intelligence Institute (Amii)
rubens.moraes@ufv.br, levi.lelis@ualberta.ca

Abstract

Syntax-guided synthesis is commonly used to generate programs encoding policies. In this approach, the set of programs, that can be written in a domain-specific language defines the search space, and an algorithm searches within this space for programs that encode strong policies. In this paper, we propose an alternative method for synthesizing programmatic policies, where we search within an approximation of the language's semantic space. We hypothesized that searching in semantic spaces is more sample-efficient compared to syntax-based spaces. Our rationale is that the search is more efficient if the algorithm evaluates different agent behaviors as it searches through the space, a feature often missing in syntax-based spaces. This is because small changes in the syntax of a program often do not result in different agent behaviors. We define semantic spaces by learning a library of programs that present different agent behaviors. Then, we approximate the semantic space by defining a neighborhood function for local search algorithms, where we replace parts of the current candidate program with programs from the library. We evaluated our hypothesis in a real-time strategy game called MicroRTS. Empirical results support our hypothesis that searching in semantic spaces can be more sample-efficient than searching in syntax-based spaces.

1 Introduction

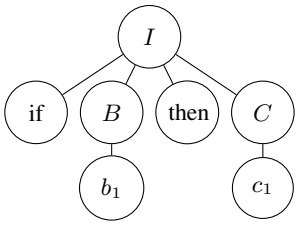
Programmatic representations of policies for solving Markov Decision-Processes (MDPs) offer advantages over neural representations, such as the ability to better generalize to similar but different scenarios than those used in training [Inala *et al.*, 2020]. Previous work has also argued that programmatic representations can allow for policies that are more amenable to interpretability [Verma *et al.*, 2018a]. Nevertheless, programmatic representations pose a difficult hurdle, since programmatic policies are generated by searching in often very large and discontinuous spaces of programs.

A commonly used approach to searching in the space of programs is local search algorithms [Koza, 1992; Husien and Schewe, 2016; Aleixo and Lelis, 2023; Moraes *et al.*, 2023]. All programs that can be written in a domain-specific language form the space of candidate solutions for local search algorithms. The search starts in one of these candidate programs, and through a neighborhood function, the search decides which program to evaluate next. The search continues until reaching a local optimum or exhausting a search budget (e.g., the number of programs evaluated). Searching in the programmatic space is difficult not only because the space of programs is often vast, but also because the search lacks guidance. The neighborhood function is defined by making small modifications to the current candidate program, which often do not result in a change of behavior of the policy encoded in the resulting program. The search algorithm thus spends a considerable portion of its computational budget evaluating programs that are different, but that represent the same policy.

In this paper, we present an alternative approach to defining the programmatic search space of local search algorithms. Instead of defining neighborhood functions where neighbors differ in syntax, we present a method to approximate the underlying semantic space of the language. That is, neighbors in the semantic space will differ in agent behavior instead of simply syntax. We hypothesized that local search algorithms searching in semantic spaces are more sample-efficient than the same algorithms searching in traditional syntax spaces.

We consider the setting in which the agent learns programmatic policies for an MDP and transfers the knowledge learned to speed up learning in other MDPs. Specifically, our method learns a library of semantically different programs while generating a policy for the first MDP, which is then used to define approximations of semantic spaces for downstream MDPs. This approximation is achieved by defining a neighborhood function in which, instead of simply changing the current candidate program in terms of syntax, we replace parts of it with programs from our library of programs.

We evaluated our hypothesis that local search algorithms are more sample-efficient in semantic spaces than in traditional spaces in the game of MicroRTS, a challenging real-time strategy game [Ontañón, 2017]. Our results show that neighbor programs in our library-induced space tend to be semantically different, while often neighbor programs in traditional spaces are semantically identical. The results also support our sample efficiency hypothesis, since Stochastic Hill Climbing (SHC) synthesized much stronger policies while



$I \rightarrow \text{if}(B) \text{ then } C$
$C \rightarrow C\,C \mid c_1 \mid c_2$
$B \rightarrow b_1 \mid b_2$

Figure 1: Abstract syntax tree for program "if b_1 then c_1" and the grammar defining the DSL.

searching in our semantic space than when searching in traditional spaces. Finally, the policies SHC synthesized while searching in our semantic spaces compared favorably with the winners of the last three MicroRTS competitions.[1]

2 Problem Definition

The problems we solve can be represented as Markov decision processes (MDPs) $(S, A, p, r, \mu, \gamma)$, where S represents the set of states and A is the set of actions. The function $p(s_{t+1}|s_t, a_t)$ encodes the transition model since it gives the probability of reaching state s_{t+1} given that the agent is in s_t and takes action a_t at time step t. The agent observes the reward value of R_{t+1} when moving from s_t to s_{t+1}; the function r returns the reward after a state transition. μ represents the distribution of initial states of the MDP. Finally, γ in $[0, 1]$ is the discount factor. A policy π is a function that receives a state s and an action a and returns the probability in which a should be taken in s. The goal is to learn a policy π that maximizes the expected sum of discounted rewards for π starting in s_0, an initial state sampled from μ: $\mathbb{E}_{\pi,p,\mu}[\sum_{k=0}^{\infty} \gamma^k R_{k+1}]$.

Let P_{train} be an MDP, which we refer to as the training problem, for which the agent learns to maximize the expected sum of discounted rewards. After learning a policy for P_{train}, we evaluate the agent while learning policies for another problem, P_{test}. In this paper, we learn a semantic space while learning a policy for P_{train} and use it to learn a policy for P_{test}.

2.1 Programmatic Policies

We consider programmatic policies, i.e., policies encoded in computer programs written in a domain-specific language (DSL). The programs a language accepts can be defined as a context-free grammar (M, Ω, R, I), where M, Ω, R, and S are the sets of non-terminals, terminals, the production rules, and the grammar's initial symbol, respectively. Figure 1 shows an example of a DSL (right), where $M = \{I, C, B\}$, with I being the initial symbol, $\Omega = \{c_1, c_2, b_1, b_2, \text{if}, \text{then}\}$, R are the production rules (e.g., $C \rightarrow CC$ and $B \rightarrow b_1$).

Programs are represented as abstract syntax trees (AST). In such trees, each node n and its children represent a production rule of the DSL if n represents a non-terminal symbol. For example, the node B and its child b_1, in Figure 1, represent the production rule $B \rightarrow b_1$. In the AST, every leaf node represents terminal symbols of the grammar. Figure 1 shows an example of an AST for the program "if b_1 then c_1".

[1]The implementation of our system is available online at https://github.com/rubensolv/Library-Induced-Semantic-Spaces

A language D defines the space of programs $[\![D]\!]$, which can be infinite. Often, in practice, the set of programs $[\![D]\!]$ can be defined to be finite. For example, one can define a maximum AST size (in terms of nodes in the tree) for the programs in $[\![D]\!]$. We consider DSLs such that the programs in $[\![D]\!]$ represent programmatic policies. The problem we tackle in this paper is to find a program encoding a policy that maximizes the expected sum of discounted rewards for an MDP.

3 Programmatic Search Spaces

In syntax-guided synthesis [Alur *et al.*, 2013; Verma *et al.*, 2018b], one searches in the space the DSL's grammar defines. A popular approach to syntax-guided synthesis, especially for solving reinforcement learning problems, is stochastic local search [Verma *et al.*, 2018b; Verma *et al.*, 2019; Mariño *et al.*, 2021]. In stochastic local search, every program in $[\![D]\!]$ is a candidate solution (a program that attempts to maximize the sum of the rewards), and these candidates are related through a neighborhood function $\mathcal{N}_k(p)$ that receives a candidate p and returns a set of k neighbor candidates. The neighborhood function is often stochastic, which means that different calls to $\mathcal{N}_k(p)$ can return a different set of neighbors. Search algorithms, such as SHC, search in the space D and $\mathcal{N}_k(p)$ induce for a program encoding a policy that maximizes the expected sum of rewards of an MDP. We define a search space for local search algorithms as follows.

Definition 1 (Search Space)**.** *A search space is defined with a tuple $(D, \mathcal{N}_k, \mathcal{I}, \mathcal{E})$, where D is a DSL with $[\![D]\!]$ defining the set of candidate solutions. $\mathcal{N}_k$ is a neighborhood function that receives a candidate in $[\![D]\!]$ and returns k candidates from $[\![D]\!]$. $\mathcal{I}$ is a function that returns an initial candidate in $[\![D]\!]$. Finally, $\mathcal{E}$ is the evaluation function, that receives a candidate in $[\![D]\!]$ and returns a real value $\mathbb{R}$.*

A search space commonly used in the literature is defined as follows [Koza, 1992; Mariño *et al.*, 2021]. The set of candidates is all programs in $[\![D]\!]$ whose AST has at least z nodes, where z is a hyperparameter. Controlling the size of the candidate programs offers an additional inductive bias. In our case, we prevent the generation of programs that are "too simple". The function $\mathcal{I}$ generates an initial candidate by starting with the initial symbol I of the grammar and applying one of the production rules, chosen uniformly at random, to replace I. If the resulting string p only contains terminal symbols, $\mathcal{I}$ returns p. Otherwise, $\mathcal{I}$ arbitrarily selects a non-terminal symbol C in p and replaces it by applying a production rule to C that is also chosen uniformly at random from the available rules. This process is repeated while there is a non-terminal symbol in p. We enforce programs with at least z nodes with a sampling rejection scheme.

The neighborhood function $\mathcal{N}_k(p)$ is defined as follows. It selects, uniformly at random, a node n in the AST of p that represents a non-terminal symbol. The subtree rooted at n is replaced in the AST by a subtree that is generated in a process similar to the one described for the function $\mathcal{I}$. We select uniformly at random a production rule that can be applied to the non-terminal symbol n represents; this process is repeated to all non-terminals in the resulting string and it stops once the string only has terminal symbols. The process of replacing

a subtree of p is repeated k times, thus generating k neighbors. The evaluation function $\mathcal{E}$ is problem-dependent. For example, if the programmatic space encodes policies for solving MDPs, then $\mathcal{E}(p)$ approximates the expected sum of discounted rewards for program p, which can be approximated by rolling out p from initial states sampled from μ. We refer to this search space as the **syntax space** since it uses the syntax of the language to define the functions $\mathcal{N}_k$ and $\mathcal{I}$.

3.1 Searching in Programmatic Search Spaces

A popular approach to searching in programmatic spaces is with local search algorithms, such as SHC. For a given computational budget, SHC starts its search with the candidate c that the function $\mathcal{I}$ returns; we refer to c as the current candidate. In every iteration, it queries the neighborhood function $\mathcal{N}_k(c)$ and evaluates all neighbors of c with respect to $\mathcal{E}$. Before starting a new iteration, SHC sets c to be the neighbor with the best $\mathcal{E}$-value. The search stops and returns the current candidate c if it reaches a local optimum, that is, none of the neighbors has a better $\mathcal{E}$-value than c. SHC is implemented with a restarting scheme: the search is restarted with a new candidate $\mathcal{I}$ returns whenever it reaches a local optimum and the algorithm has not exhausted its computational budget.

The set of neighbors in the syntax space often allows neighbors to be syntactically similar, since they differ from the current candidate only by one subtree of their ASTs. This notion of syntactical locality is important because it can result in a sequence of locally improving programs that guide local search algorithms from the initial candidate to a candidate that maximizes the sum of discounted rewards for an MDP.

4 Semantic-Based Search Spaces

We hypothesize that local search algorithms will be more sample-efficient if the neighborhood function induces search spaces in which neighbors *behave* differently, i.e., neighbor programs are semantically different. We base our hypothesis on the observation that algorithms searching in the syntax space spend a large portion of their computational budget evaluating programs that are syntactically different but are semantically identical. The evaluation of semantically identical programs can slow down the search for two reasons. First, the search uses its computational budget to evaluate the same behavior multiple times. Second, local search algorithms, such as SHC, are more effective when the evaluation function provides search guidance. For example, if all neighbors of a program p are semantically identical to p, then the search is forced to restart, since all semantically identical programs evaluate to the same $\mathcal{E}$-value, thus representing a local optimum. The lack of search guidance can force the search to restart even if the search is near a promising program. We introduce the notion of β-proper spaces to measure the extent to which neighbors in a space are semantically different.

Definition 2 (β-proper)**.** *Let p_β be the probability that a program p, which is sampled uniformly at random from $[\![D]\!]$, has a neighbor that behaves identically to p. A search space is β-proper if its p_β-value is less than β, for a small β.*

Algorithm 1 Library Construction

Require: Training problem P_{train}, local search algorithm LS
Ensure: A library $\mathcal{L}$ of semantically different programs.
1: $\mathcal{P}, \mathcal{S} \leftarrow$LS$(P_{\text{train}})$ *# using syntax space*
2: $\mathcal{L} \leftarrow \emptyset$
3: SIGNATURE $\leftarrow \emptyset$
4: **for** each p in $\mathcal{P}$ **do**
5: **for** each subtree t rooted at a non-terminal in p **do**
6: $\mathcal{A}[s] \leftarrow t(s)$ for each s in $\mathcal{S}$
7: **if** $\mathcal{A}$ is not in SIGNATURE **then**
8: $\mathcal{L} \leftarrow \mathcal{L} \cup \{t\}$
9: SIGNATURE $\leftarrow$ SIGNATURE $\cup\{\mathcal{A}\}$
10: **return** $\mathcal{L}$

The behavior of programs used in Definition 2 depends on the application. For example, in sequential decision-making problems, behavior can be determined by the sequences of actions of two programs performed from a set of initial states.

5 Semantic Spaces

We introduce a method that uses a library of learned programs to induce β-proper spaces. Instead of generating neighbors by replacing a subtree of the current candidate program with another subtree randomly generated from the grammar of the DSL, we replace the subtree with a program from our library. The library is composed of semantically different programs, which are generated while searching for a policy that maximizes the reward in P_{train}. We hypothesized that this library-induced space can be β-proper because the subtrees used to generate neighbors are themselves programs with well-defined behavior—subtrees for which we cannot evaluate their semantics (e.g., the code crashes) are not added to the library. When generating subtrees by randomly applying the rules of the grammar, as we do in the syntax space, we can generate subtrees that do not affect the program behavior. For example, the subtree "if False then c_1" does not change any program's behavior, independently of c_1. By using programs from the library, we guarantee that the subtree added to the program while generating one of its neighbors has a well-defined behavior. Moreover, the behavior of the program added is unique within the library of programs.

5.1 Library Construction

Algorithm 1 shows the process of constructing our library $\mathcal{L}$ of semantically different programs. The procedure receives the training problem P_{train} and a local search algorithm LS, and it returns $\mathcal{L}$. In line 1, the procedure invokes LS to search, using the syntax space, for a programmatic policy that maximizes the reward in P_{train}. The search LS performs returns all programs encountered in the process, $\mathcal{P}$, and a set of states $\mathcal{S}$ encountered while evaluating programs in the LS search. In lines 2 and 3, we initialize the library $\mathcal{L}$ and a set of action-signatures SIGNATURES. An action-signature is a vector $\mathcal{A}$ with one entry for each s in $\mathcal{S}$ containing the action a a program returns for s. We use the action-signatures to approximate program semantics: if two programs have the same action-signature, we deem them as semantically identical.

The procedure then iterates through all programs in $\mathcal{P}$ (line 4) and, for each p, it iterates through each subtree t of the AST of p that represents a non-terminal symbol (line 5). For example, the AST shown in Figure 1 would have three subtrees: I, B, and C. We then evaluate the action t returns for each s in S (line 6), thus computing the action-signature of t. Note that if the program t cannot be executed (e.g., the subtree does not represent a complete program), then we do not consider adding it to the library (this is not shown in the pseudocode). If the action-signature $\mathcal{A}$ of a program t is not in SIGNATURES, then we add t to $\mathcal{L}$ and $\mathcal{A}$ to SIGNATURES—t is the program in $\mathcal{L}$ representing the behavior $\mathcal{A}$ defines.

5.2 Library-Induced Semantic Space

We define a library-induced semantic space (LISS) as a search space that is identical to the syntax space, except for its neighborhood function. Instead of generating neighbors by using the rules of the grammar, we use the programs in $\mathcal{L}$. Similarly to the neighborhood function of the syntax space, in LISS we randomly select a node n representing a non-terminal symbol N in the AST of the current candidate. Then, we randomly select a program t in $\mathcal{L}$ whose AST root also represents the non-terminal N; t replaces the subtree rooted n in the candidate program, thus generating a neighbor. All k neighbors in the LISS are generated with the same process.

In practice, we do not rely entirely on LISS, but on a mixture of LISS and the syntax space. This is because, depending on P_{train} and the policy generated for it, LISS might not be able to reach a good portion of the original program space through its neighborhood function because some of the symbols might be missing from the library. During the search with LISS, we generate neighbors using the function $\mathcal{N}_k$ of the syntax space with probably ϵ and, with probability $1 - \epsilon$, we generate them using the function $\mathcal{N}_k$ from LISS. This guarantees that the search has access to all programs of the language, not only in the initialization of the search but also during the search, through the neighborhood function.

The semantic space of LISS continues to improve as we search for a solution to P_{test}. We use the programs encountered while searching for a programmatic policy for P_{test} to grow the library of programs. If the search encounters a program p while searching for a solution to P_{test} that is semantically different from all the programs in the library, then p is added to the library, thus updating the semantic space used in the search. The idea is that the semantic space is continually learned, which could be helpful in settings where there is no split of training and testing problems, but just a stream of problems that the agent needs to learn how to solve.

6 Empirical Methodology

In this section, we describe our methodology to evaluate the hypothesis that search algorithms operating in LISS are more sample-efficient than when operating in the syntax space.

6.1 Problem Domain: MicroRTS

We evaluate LISS using the MicroRTS domain, a real-time strategy game designed for research. There is an active research community that uses MicroRTS as a benchmark to evaluate intelligent systems.[2] MicroRTS is a game played with real-time constraints and very large action and state spaces [Lelis, 2021]. Each player controls two types of stationary units (Bases and Barracks) and four types of mobile units (Workers, Ranged, Light, and Heavy). Bases are used to store resources and train Workers. Barracks can train Ranged, Light, and Heavy units. Workers can build stationary units, harvest resources, and attack opponent units. Ranged, Light, and Heavy units have different amounts of hit points and inflict different amounts of damage to opponent units. Ranged units differ from each other by causing damage from far away. In MicroRTS, a match is played on a grid, which represents the map. Due to the different structures of the maps, different maps might require different policies to play the game.

Since MicroRTS is a two-player zero-sum game, one learns a policy through a self-play scheme. One of the simplest methods we can use is Iterated Best Response (IBR) [Lanctot *et al.*, 2017]. IBR starts with an arbitrary policy for one of the players and it approximates, by searching in the programmatic space, a best response to this initial policy. In the next iteration, IBR attempts to approximate a best response to the best response computed in the previous iteration. This process is repeated for a number of iterations, which is normally determined by a computational budget, and the last policy for each player is returned as the output of IBR. Self-play algorithms such as IBR require one to solve many MDPs. Every computation of a best response represents an MDP since the other player is fixed and can be seen as part of the environment. Instead of using IBR, we use Local Learner (2L), another self-play algorithm that was shown to synthesize strong programmatic policies for MicroRTS [Moraes *et al.*, 2023].

We use six maps of different sizes in our experiments (the names of the maps reflect their names in the MicroRTS public repository): NoWhereToRun (NWR 9×8), itsNotSafe (INS 15×14), letMeOut (LMO 16×8), Barricades (BRR 24×24), Chambers (CHB 32×32), and BloodBath.scmB (BBB 64×64). The last is an adaption for MicroRTS of a map from the commercial game StarCraft. All these maps are used as P_{test} in our experiments; we used the map basesWorkersA (24×24) as P_{train}. We consider two starting locations on each map. When evaluating two policies, to ensure fairness, each policy plays one match in both locations on the map.

The results are presented in terms of winning rate. The winning rate of a policy against a set of other policies is given by the number of victories added to half of the number of draws, divided by the total number of matches played, and multiplied by 100. For example, if a policy plays 10 matches, wins 2, draws 1, and loses 7, its winning rate is 25. We are interested in measuring the sample efficiency of the approaches in terms of the number of games played, so we will present plots showing the winning rate by the number of games.

We use a domain-specific language developed for MicroRTS called Microlanguage [Medeiros *et al.*, 2022]. This language includes high-level functions such as "haverst" and also loops that iterate through the players' units. The loops in the Microlanguage allow for implementing prioritization schema. This is because once an action is assigned to a unit,

[2] https://github.com/Farama-Foundation/MicroRTS/wiki

it cannot be replaced, so instructions appearing early in the loops will have higher priority than later instructions.

6.2 Experiments Performed

We perform three experiments. The first experiment evaluates the values of β for which the syntax and the semantic spaces are β-proper. We evaluate the "pure" version of LISS, where we do not mix the neighborhood function of the semantic space with that of the syntax space. We generate 50 programs by following the function $\mathcal{I}$ of the syntax space, and for each of these programs p, generate 1000 neighbors p' of p according to the space's neighborhood function. Then, we roll-out p and each of its p's once in the following maps: NWR, LMO, and BRR. Since MicroRTS is deterministic, we can use the action-signature of p and p' in these roll-outs to approximate their behavior. If the action-signatures are identical, then we assume them to be semantically identical; they are not semantically identical otherwise. When evaluating p and p', we use another neighbor of p as the other player in the roll-outs. This data approximates p_β for both spaces.

The second experiment evaluates our hypothesis that a local search algorithm searching in LISS is more sample-efficient than the same algorithm searching in the syntax space. We use SHC with restarts as the algorithm in our experiments. We chose to use SHC because it was shown to perform well in the synthesis of programmatic policies for MicroRTS [Moraes *et al.*, 2023]. We use $k = 1000$ in $\mathcal{N}_k$ and a limit of 400 seconds for SHC to return a best response; once it reaches this time limit, it returns the best policy it encountered across all restarts of the search. We induce LISS with the library learned from all the programs generated in a single run of 2L with SHC on the basesWorkersA 24×24 map. The action-signatures of the programs considered for the library were generated by evaluating them in 400 states. These 400 states are collected by randomly selecting pairs of policies representing best responses in the self-play process 2L executes and playing them on P_{train}; every state encountered in these matches is added to the set, up to a total of 400 states. We use $\epsilon = 0.20$ for mixing the syntax and semantic spaces. We evaluated ϵ in $\{0.10, 0.20\}$ in preliminary experiments and found that 0.20 performed better. We perform 30 independent runs of this experiment and present average winning rates and the 95% confidence intervals. Each of the 30 independent runs uses a library generated with an independent run of 2L with SHC on the basesWorkerA 24×24 map.

As baselines for the second experiment, we use 2L with SHC in the syntax space (denoted 2L) and a version of 2L with SHC in the syntax space that initializes the search for best responses in each P_{test} with the solution returned in P_{train}, denoted 2L-I. The programmatic representation we use is known to generalize well across maps [Aleixo and Lelis, 2023], so 2L-I is expected to be a strong baseline.

The third experiment compares the policies SHC searching in LISS synthesizes with those of the winners of the last three MicroRTS competitions:[3] COAC,[4] Mayari,[5] and RAISocketAI.[6] COAC and Mayari are programmatic policies human programmers wrote in Java for the MicroRTS competition. RAISocketAI is a Deep Reinforcement Learning (DRL) system trained with the Proximal Policy Optimization algorithm [Schulman *et al.*, 2017]. We evaluate all these algorithms in all six P_{test} maps. COAC, Mayari, and RAISocketAI were evaluated and trained only in the maps NWR and BBB. For the RAISocketAI agent, we used the models trained for maps of the same size for the remaining four maps.

In the third experiment, we ran our system six times for each of these maps, with each run returning a programmatic policy. We then run a round-robin tournament among these six policies and select the one with the highest winning rate. This selected policy is the one we evaluate against the winners of the last three competitions by playing each of them five times at each of the two starting locations we consider, for a total of ten matches on each map. We repeat this entire process five times, and we present average results for the five runs. This experiment evaluates how well the policies generalize to unseen opponents by simulating a tournament.

We used a dedicated number of computers with the following settings: 16 GB of RAM, i7-1165G7 CPUs at 2.80 GHz with 8 threads. We also use $z = 4$ in all our experiments.

7 Empirical Results

Next, we present the results of our three experiments.

7.1 Experiment 1: p_β

The p_β-value for the syntax space was 0.19 with a standard deviation of 0.15, while the p_β-value for LISS was 0.01 with a standard deviation of 0.01. These results support our hypothesis that our LISS is β-proper, for β as low as 0.02. The value of 0.19 for the syntax space is quite high, which means that almost 20% of the neighbors are semantically identical to the current candidate program, forcing the search to evaluate equivalent policies multiple times, possibly slowing down the search. While the p_β-values provide indication that LISS is more "friendly" to local search algorithms than the syntax space, one can imagine degenerate cases where the space is β-proper but the behaviors of neighboring candidates do not result in a space with good search guidance. Experiments 2 and 3 complement the results of Experiment 1 to evaluate how friendly LISS is to local search.

7.2 Experiment 2: Sample Efficiency

Figure 2 presents the results. In these plots, we compute the winning rate of each method by having them play the last policy each of the other methods synthesizes. For example, on the Barricades map, the policy 2L synthesizes after playing 10000 games is evaluated against the policies 2L-I and LISS synthesize after playing 30000 games, which is the maximum computational budget used in the experiments on this map.

Before these experiments, 2L was the state-of-the-art synthesizer for policies for playing MicroRTS. The ability to start the search from a strong policy synthesized for P_{train} already outperforms 2L on most maps by a large margin. The use

[3]https://sites.google.com/site/micrortsaicompetition

[4]https://github.com/Coac/coac-ai-microrts

[5]https://github.com/barvazkrav/mayariBot

[6]https://github.com/sgoodfriend/rl-algo-impls

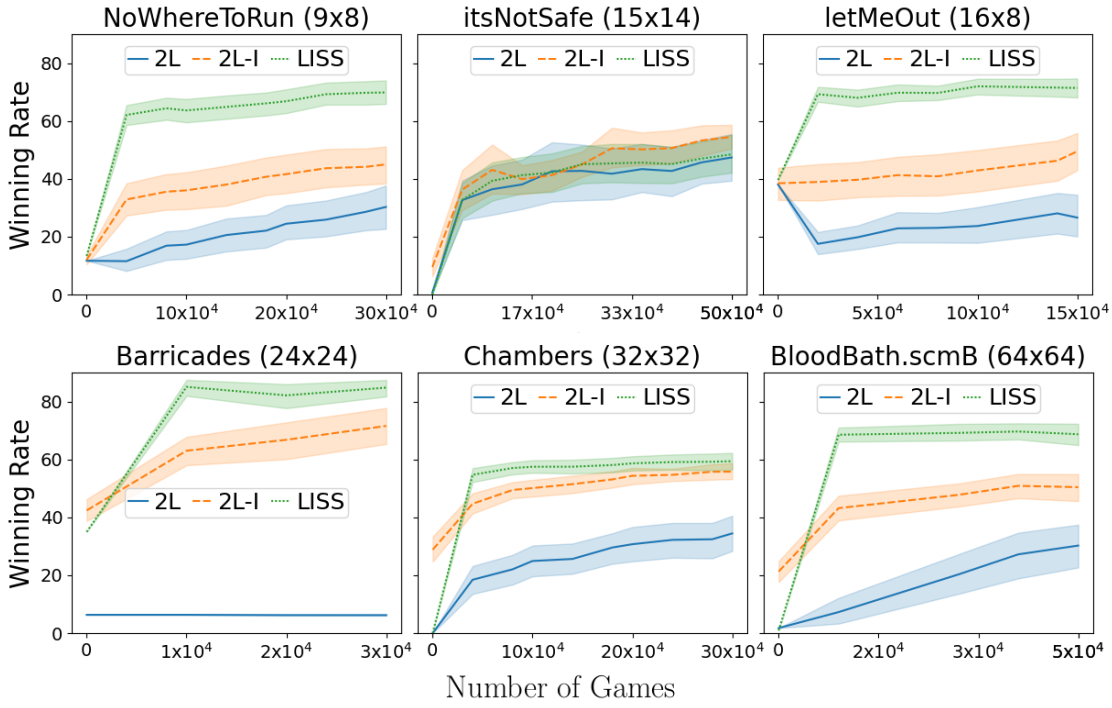


Figure 2: Results of each learning algorithm on six MicroRTS maps using 2L as the learning algorithm and Stochastic Hill Climbing as the search algorithm. These curves represent the winning rate of each algorithm compared to their opponents using the same amount of games.

of the initialization of search with a strong policy for the baseWorkersA map makes a large difference in the Chambers map, where a policy similar to the one learned to play in P_{train} is already quite strong. However, the semantic space of LISS allows for a much more sample-efficient learning process, so LISS quickly catches up and eventually synthesizes a policy that is stronger than that which 2L-I manages to synthesize. A similar pattern is observed on the largest map, but with a larger gap between 2L-I and LISS. Overall, LISS outperforms both baselines by a large margin on all but the itsNotSafe map, where there is no statistical difference between the methods. In this map, a simple policy, which trains a Ranged unit and attacks the opponent, is already very strong. This map essentially represents an easy search problem, that SHC is able to tackle searching in either the syntax space or LISS.

These results support our hypothesis that SHC is more sample-efficient while searching in LISS than in the syntax space. Moreover, the results also suggest that searching in the semantic space of LISS can be more effective than using a powerful initialization of the search in the syntax space.

7.3 Experiment 3: Competition Agents

Table 1 shows the results of our third experiment. If we consider only the maps on which COAC, Mayari, and RAISocketAI were evaluated and trained (NWR and BBB), LISS has a winning rate higher than 50 in all comparisons, but against RAISocketAI on the NWR map. NWR is a small map where DRL is able to learn a strong micromanagement of units (e.g., clever optimizations during combat). The ability to synthesize strong micromanagement of units is currently beyond the reach of LISS due to a limitation of the Microlanguage that focuses on the macro aspects of the game (e.g., which combinations of units to train). LISS obtains a winning rate of 100 against RAISocketAI on the larger BBB map. This highlights the difficulties of training DRL agents on larger problems.

If we consider all six maps, then the comparisons in which LISS has a winning rate smaller than 50 against a particular opponent are clearly outnumbered by the comparisons in which its rate is greater than 50. In addition to BBB, LISS performs notably well on the INS map because the strong policies for this map do not generalize to other maps, despite being easy to find them, as discussed in the second experiment. However, COAC, Mayari, and RAISocketAI were not designed or trained for this map, which explains their performance. The policies SHC synthesizes by searching in the semantic space of LISS outperform, on average, the winners of the last three competitions, thus demonstrating that these policies can generalize to unseen but similar MDPs, as the MDP effectively changes as we change the opponent.

8 Related Works

Our research is related to previous work on Programmatically Interpretable Reinforcement Learning (PIRL) [Verma *et al.*, 2018a], where the objective is to generate human-readable programs encoding policies for addressing Reinforcement Learning tasks [Bastani *et al.*, 2018; Verma *et al.*, 2019]. Previous work in this area does not distinguish between syntactic and semantic spaces and performs the search for programs in spaces that are probably not β-proper for the small values of β we observed in our experiments. Thus, the idea of learning

Agents	Maps						
	NWR 9×8	**INS 15×14**	**LMO 16×8**	**BRR 24×24**	**CHB 32×32**	**BBB 64×64**	**Avg. Agent**
COAC	55.00	100.00	72.50	45.00	37.50	67.50	62.92
Mayari	57.50	95.00	88.33	65.00	47.50	90.00	73.89
RAISocketAI	40.83	100.00	60.00	97.50	100.00	100.00	83.06
Avg. Map	51.11	98.33	73.61	69.17	61.67	85.83	73.29

Table 1: Average winning rate of LISS against the winner of the last three MicroRTS competitions (rows) in six maps (columns).

a semantic space can potentially be applied to some of these works. In generalized planning one is interested in synthesizing programs encoding policies for solving classical planning problems [Bonet *et al.*, 2010; Srivastava *et al.*, 2011; Hu and De Giacomo, 2013; Aguas *et al.*, 2018]. Similarly to the work in PIRL, it would be interesting to investigate the idea of learning β-proper semantic spaces for solving classical planning problems with generalized planning.

Previous work has also investigated the use of learned latent spaces for synthesizing programmatic policies [Trivedi *et al.*, 2021; Liu *et al.*, 2023]. In this line of work, a latent space is trained before the search starts by sampling programs from the grammar describing the language. The latent space is trained so that vectors near each other in the space represent programs that behave similarly. Like our work, this line of research is also concerned with the semantics of the programs. In contrast to our work, the latent space is not trained with the goal of attaining β-proper spaces. In fact, since the space is continuous, it is challenging to design neighborhood functions as one needs to choose the magnitude in which the vector representing the current candidate will be modified to generate a neighbor. Furthermore, recent work showed that the SHC we use in our experiments outperformed algorithms searching in learned latent spaces [Carvalho *et al.*, 2024].

Programmatic policies have also been used to guide tree search algorithms by reducing the action space in games such as MicroRTS. Puppet Search (PS) [Barriga *et al.*, 2017b] defines a search space, similar to the one defined by semantic space, by changing the parameter variables of handmade programmatic policies. Strategy Tactics (STT) [Barriga *et al.*, 2017a] combines PS's search with a NaïveMCTS search [Ontañón, 2017] in a small fraction of the state space for combat micromanagement. Strategy Creation via Voting (SCV) generates policies via voting [Silva *et al.*, 2018], which plays the game by combining also manually crafted policies. In contrast to PS, STT, and SCV which use manually crafted programmatic policies, LISS introduces a method to synthesize programmatic policies. Another line of research uses programmatic policies for inducing action abstractions [Moraes and Lelis, 2018], where tree search algorithms consider only a subset of the actions available for search. This subset is determined by the actions a set of programmatic policies return at a given state. Future research might investigate the use of policies LISS synthesizes to induce action abstractions for tree search algorithms.

Dynamic Scripting (DS) is an algorithm for synthesizing programmatic policies for zero-sum role-playing games. DS generates programmatic policies by extracting rules from an expert-designed rule base according to a learned policy [Spronck *et al.*, 2004]. DS has also been applied for zero-sum RTS games [Ponsen and Spronck, 2004; Dahlbom and Niklasson, 2006]. Our method is more expressive because it considers spaces defined by DSLs, as opposed to being dependent on a particular DSL that allows one to define rules.

DreamCoder and its extensions, Stitch and Babble, are systems that learn a library of programs in the context of supervised learning [Ellis *et al.*, 2023; Bowers *et al.*, 2023; Cao *et al.*, 2023]. Our approach differs from these systems in important ways. We use a library of programs to define the search space of local search algorithms, while DreamCoder uses an enumerative approach guided by a learned function. Moreover, it is not clear how to learn a function to guide the search for programmatic policies. This is because we do not know the MDP we will solve ahead of time, before the agent interacts with the environment. By contrast, in DreamCoder's supervised learning setting, it is reasonable to assume that the training and testing problems come from similar distributions and an effective guiding function can be learned a priori.

9 Conclusions

We hypothesized that algorithms searching in programmatic spaces where neighbors encode similar but semantically different programs are more sample-efficient than algorithms searching in the syntax space. In this paper, we showed empirically that, often in syntax spaces, neighbors will be semantically identical, which could slow down the search. We then introduced Library-Induced Semantic Spaces (LISS), where the neighbors of a candidate program in the space are generated by replacing parts of the candidate with programs from a library of semantically different programs. We showed empirically that LISS is β-proper, for a small value of β, in the domain of MicroRTS, while the syntax space is not. Our results also supported our sample efficiency hypothesis, since the programs a local search algorithm synthesized by searching in LISS encoded much stronger policies than the programs the same search algorithm synthesized while searching in the syntax space. We also showed empirically that the policies LISS synthesizes can outperform the winners of the last three MicroRTS competitions, which include two programmatic policies written by human programmers and a DRL agent. Overall, our results suggest that the design of semantic spaces is a promising direction for methods that rely on search algorithms for synthesizing programmatic policies.

Acknowledgments

This research was supported by Canada's NSERC, and the CIFAR AI Chairs program, and Brazil's CAPES. The research was carried out using computational resources from the Digital Research Alliance of Canada and the UFV Cluster. We thank the anonymous reviewers for their feedback.

References

[Aguas *et al.*, 2018] Javier Segovia Aguas, Sergio Jiménez, and Anders Jonsson. Computing hierarchical finite state controllers with classical planning. *Journal of Artificial Intelligence Research*, 62:755–797, 2018.

[Aleixo and Lelis, 2023] 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.

[Alur *et al.*, 2013] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo Martin, Mukund Raghothaman, Sanjit Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. pages 1–17, 10 2013.

[Barriga *et al.*, 2017a] Nicolas Barriga, Marius Stanescu, and Michael Buro. Combining strategic learning and tactical search in real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 9–15. AAAI, 2017.

[Barriga *et al.*, 2017b] Nicolas Barriga, Marius Stanescu, and Michael Buro. Game tree search based on nondeterministic action scripts in real-time strategy games. *IEEE Transactions on Computational Intelligence and AI in Games*, pages 69–77, 2017.

[Bastani *et al.*, 2018] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Advances in Neural Information Processing Systems*, pages 2499–2509, 2018.

[Bonet *et al.*, 2010] Blai Bonet, Héctor Palacios, and Héctor Geffner. Automatic derivation of finite-state machines for behavior control. In *Proceedings of the AAAI Conference on Artificial Intelligence*, page 1656–1659. AAAI Press, 2010.

[Bowers *et al.*, 2023] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Procedings of the ACM on Programming Languages*, 7(POPL), 2023.

[Cao *et al.*, 2023] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. Babble: Learning better abstractions with e-graphs and anti-unification. *Procedings of the ACM on Programming Languages*, 7(POPL), 2023.

[Carvalho *et al.*, 2024] Tales Henrique Carvalho, Kenneth Tjhia, and Levi H. S. Lelis. Reclaiming the source of programmatic policies: Programmatic versus latent spaces. In *The Twelfth International Conference on Learning Representations*, 2024.

[Dahlbom and Niklasson, 2006] Anders Dahlbom and Lars Niklasson. Goal-directed hierarchical dynamic scripting for rts games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 21–28, 2006.

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

[Hu and De Giacomo, 2013] Yuxiao Hu and Giuseppe De Giacomo. A generic technique for synthesizing bounded finite-state controllers. *Proceedings of the International Conference on Automated Planning and Scheduling*, 23(1):109–116, 2013.

[Husien and Schewe, 2016] Idress Husien and Sven Schewe. Program generation using simulated annealing and model checking. In Rocco De Nicola and Eva Kühn, editors, *Software Engineering and Formal Methods*, pages 155–171. Springer International Publishing, 2016.

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

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

[Lanctot *et al.*, 2017] Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Pérolat, David Silver, and Thore Graepel. A unified game-theoretic approach to multiagent reinforcement learning. *Advances in neural information processing systems*, 30, 2017.

[Lelis, 2021] Levi H. S. Lelis. Planning algorithms for zero-sum games with exponential action spaces: A unifying perspective. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 4892–4898, 2021.

[Liu *et al.*, 2023] 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.

[Mariño *et al.*, 2021] Julian R. H. Mariño, Rubens O. Moraes, Tassiana C. Oliveira, Claudio Toledo, and Levi H. S. Lelis. Programmatic strategies for real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 381–389, 2021.

[Medeiros *et al.*, 2022] 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.

[Moraes and Lelis, 2018] Rubens O. Moraes and Levi H. S. Lelis. Asymmetric action abstractions for multi-unit control in adversarial real-time scenarios. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*. AAAI, 2018.

[Moraes *et al.*, 2023] Rubens O. Moraes, David S. Aleixo, Lucas N. Ferreira, and Levi H. S. Lelis. Choosing well your opponents: How to guide the synthesis of programmatic strategies. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 4847–4854, 2023.

[Ontañón, 2017] Santiago Ontañón. Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research*, 58:665–702, 2017.

[Ponsen and Spronck, 2004] Marc Ponsen and Pieter Spronck. *Improving adaptive game AI with evolutionary learning*. PhD thesis, Citeseer, 2004.

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

[Silva *et al.*, 2018] Cleyton Silva, Rubens O Moraes, Levi HS Lelis, and Kobi Gal. Strategy generation for multi-unit real-time games via voting. *IEEE Transactions on Games*, 2018.

[Spronck *et al.*, 2004] Pieter Spronck, Ida Sprinkhuizen-Kuyper, and Eric Postma. Online adaptation of game opponent ai with dynamic scripting. *International Journal of Intelligent Games and Simulation*, 3(1):45–53, 2004.

[Srivastava *et al.*, 2011] Siddharth Srivastava, Neil Immerman, Shlomo Zilberstein, and Tianjiao Zhang. Directed search for generalized plans using classical planners. In *Proceedings of the International Conference on Automated Planning and Scheduling*. AAAI, 2011.

[Trivedi *et al.*, 2021] 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.

[Verma *et al.*, 2018a] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, pages 5052–5061, 2018.

[Verma *et al.*, 2018b] 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.

[Verma *et al.*, 2019] Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 32, pages 1–12. Curran Associates, Inc., 2019.