Show Me the Way! Bilevel Search for Synthesizing Programmatic Strategies

David S. Aleixo[1], **Levi H. S. Lelis**[2]

[1]Departamento de Informática, Universidade Federal de Viçosa, Brazil
[2]Department of Computing Science, Alberta Machine Intelligence Institute (Amii), University of Alberta, Canada

Abstract

The synthesis of programmatic strategies requires one to search in large non-differentiable spaces of computer programs. Current search algorithms use self-play approaches to guide this search. The issue with these approaches is that the guiding function often provides a weak search signal. This is because self-play functions only measure how well a program performs against other programs. Thus, while small changes to a losing program might not transform it into a winning one, such changes might represent steps in the direction of a winning program. In this paper we introduce a bilevel search algorithm that searches concurrently in the space of programs and in a space of state features. Each iteration of the search in the space of features defines a set of target features that the search in the program space attempts to achieve (i.e., features one observes while following the strategy encoded in a program). We hypothesize the combination of a self-play function and a feature-based one provides a stronger search signal for synthesis. While both functions are used to guide the search in the program space, the self-play function is used to guide the search in the feature space, to allow for the selection of target features that are more likely to lead to winning programs. We evaluated our bilevel algorithm in MicroRTS, a real-time strategy game. Our results show that the bilevel search synthesizes stronger strategies than methods that search only in the program space. Also, the strategies our method synthesizes obtained the highest winning rate in a simulated tournament with several baseline agents, including the best agents from the two latest MicroRTS competitions.

Introduction

Programmatic strategies are strategies encoded in human-readable computer programs. Such a programmatic representation often allows one to not only understand, but also modify the encoded strategy. On the downside, synthesizing programmatic strategies requires one to search over large non-differentiable program spaces. Current methods for synthesizing such strategies use the search signal self-play algorithm such as Iterated-Best Response (IBR) provide (Mariño et al. 2021; Medeiros, Aleixo, and Lelis 2022). However, the search signal self-play algorithms provide tends to be weak. This is because they only inform the utility of a strategy while playing the game with another strategy. While changes to a program might not turn it into a winning one, they might represent steps in the direction of achieving a winning program. Self-play functions are unable to capture this progress.

An alternative approach is to use an oracle to provide a richer search signal to guide the search in the programmatic space (Bastani, Pu, and Solar-Lezama 2018; Verma et al. 2018, 2019). In addition to an oracle strategy not always being available, imitating an oracle strategy might lead to weak strategies due to a possible representation gap (Qiu and Zhu 2022). That is, there might not exist a programmatic strategy in the program space that is able to mimic the oracle.

In this paper, we introduce an algorithm for synthesizing programmatic strategies that simultaneously searches in two spaces. Our goal with this bilevel search is to enhance the search signal self-play algorithms such as IBR provide in the context of two-player zero-sum games. Our algorithm, Bilevel Synthesis (Bi-S), searches in the program space defined by a domain-specific language and in the space of domain-dependent game features. Each iteration of the search in the feature space defines a complete assignment of a set of feature-variables. Such an assignment is provided as input to the search in the program space. While searching in the program space, the algorithm uses the self-play algorithm's function to guide the search. In case of a tie between two programs (e.g., both are defeated by a target strategy), then the search prefers the program that reaches game states with features that are more similar to those encoded in the variable assignment provided by the feature-space search.

The evaluations the program-space search performs with respect to the self-play function are then used to inform the search in the feature space. That is, the feature space search attempts to learn the combination of feature values that leads to programs optimizing the self-play function. We hypothesize that this bilevel scheme allows for a more informed search in the program space than the search that uses only the signal from self-play algorithms.

We evaluated Bi-S in MicroRTS, a real-time strategy game. Our results show that Bi-S using either IBR or Fictitious play (Brown 1951), another self-play algorithm, consistently outperforms baselines that search only in the program space. We also simulated a tournament of MicroRTS, where we evaluated Bi-S as well as several other baselines, including the programmatic strategies, written by human programmers, that won the latest two MicroRTS competi-

tions (Ontañón 2017b). Bi-S with IBR and FP came in first and second, respectively, in our simulated tournament.

Related Work

Bi-S is related to programmatically interpretable reinforcement learning (PIRL) (Verma et al. 2018), where one attempts to synthesize a program encoding a policy for solving Markov decision processes (Bastani, Pu, and Solar-Lezama 2018; Verma et al. 2019). Our work is also related to generalized planning where one synthesizes programs for solving sets of classical planning problems (Bonet, Palacios, and Geffner 2010; Srivastava et al. 2011; Hu and De Giacomo 2013; Aguas, Jiménez, and Jonsson 2018). Our work differs from PIRL and generalized planning because we focus on games, while they focus on single-agent problems. Still in the context of single-agent problems, Inductive Logic Programming has been used to learn programs for playing games (Silver et al. 2019) and for learning game rules from game traces (Cropper, Evans, and Law 2020).

Qiu and Zhu (2022) argued that approaches based on imitation learning can lead to weak policies due to a representation gap, i.e., programmatic policies might not be able to imitate neural policies, as a program that imitates the oracle might not exist in the programmatic space. Qiu and Zhu introduced a system for learning programmatic policies without oracles by using a differentiable approximation of the language used to encode the policies. Qiu and Zhu's system and most PIRL algorithms do not support languages with loops. Bi-S supports if-then-else structures and loops. Finally, Bi-S does not use an oracle to guide the search.

Mariño et al. (2021) introduced a system that uses data an oracle generates to simplify the language used in the synthesis of strategies. Medeiros, Aleixo, and Lelis (2022) introduced Sketch-SA, a system that uses imitation learning for synthesizing strategies. Despite using imitation learning, Sketch-SA is unlikely to suffer from representation gaps because it uses an oracle to learn a sketch of a program (i.e., an unfinished program), as opposed to learning complete programs. The complete program is synthesized while searching in the space of programs and optimizing for a self-play function. The bilevel search we introduce is orthogonal to the ideas of simplifying languages and of learning sketches.

Self-play algorithms were used to learn strategies with reinforcement learning (Heinrich, Lanctot, and Silver 2015; Lanctot et al. 2017). In these works the strategy is encoded in a neural network and learned with gradient ascent. We consider the setting where the strategies are encoded in computer programs, so the strategy space is full of discontinuities and gradient-based methods would not be effective.

Problem Definition

We consider sequential two-player zero-sum games defined by a set S of states, a pair of players $\{i, -i\}$, an initial state s_{init} in S, a function $A_i(s)$ that receives a state s and returns the set of actions player i can perform at s, and a function $U_i(s)$ that returns the utility of player i at s. Since the game is zero sum, $U_i(s) = -U_{-i}(s)$. A strategy for player i is a function $\sigma_i : S \rightarrow A_i$ mapping a state s to an action a. A programmatic strategy is a computer program encoding a strategy σ. We denote as $U(s, \sigma_i, \sigma_{-i})$ the value of the game for s given that i and $-i$ follow σ_i and σ_{-i}, respectively.

We use a domain-specific language (DSL) (Van Deursen, Klint, and Visser 2000) to define the space of programs, and thus the space of strategies for playing the game. We denote D as a DSL and $[\![D]\!]$ as the (possibly infinite) set of programs that can be written with D. We use a context-free grammar (N, T, R, I) to define D. Here, N, T, and R are sets of non-terminals, terminals, relations defining the production rules of the grammar, respectively. I is the grammar's start symbol. Figure 1 shows a DSL where $N =$

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

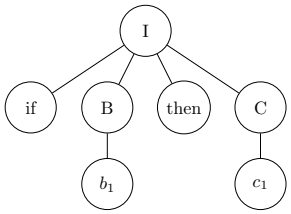

Figure 1: DSL and AST for `if` b_1 `then` c_1.

$\{I, C, B\}$, $T = \{c_1, c_2, b_1, b_2$ if, then$\}$, R are the relations (e.g., $C \rightarrow c_1$), and I is the start symbol.

This DSL allows for programs such as c_1, c_2, and `if` (b_2) `then` c_1. We represent programs in memory as abstract syntax trees (AST), where the root of the tree is the initial symbol, the internal nodes are non-terminals and leaves are terminals. Figure 1 shows an example of an AST.

The best response of σ_{-i} in $[\![D]\!]$ is a strategy that maximizes player i's utility against σ_{-i}, i.e., $\max_{\sigma_i \in [\![D]\!]} U(s_{\text{init}}, \sigma_i, \sigma_{-i})$. In the context of two-player zero-sum games, a Nash equilibrium profile in the space of programmatic strategies is a pair of programs that best respond to one another, i.e., the strategies σ_i and σ_{-i} that solve

$$\max_{\sigma_i \in [\![D]\!]} \min_{\sigma_{-i} \in [\![D]\!]} U(s_{\text{init}}, \sigma_i, \sigma_{-i})\,. \tag{1}$$

Our goal is to approximately solve Equation 1. Since we consider DSLs whose programs encode pure strategies, we assume the existence of a pure-strategy Nash equilibrium.

Self-Play Algorithms

We consider Iterated-Best Response (IBR) (Lanctot et al. 2017) and Fictitious Play (FP) (Brown 1951) as the learning algorithms to approximate a solution to Equation 1.

Iterated-Best Response In IBR one starts with an arbitrary strategy σ_i for player i and computes a best response σ_{-i} to σ_i. Then, in the next iteration of IBR, it computes a best response to σ_{-i}, and so. This alternating process of computing best responses is repeated until finding a Nash equilibrium profile or reaching a time limit.

Fictitious Play Similarly to IBR, FP starts with an arbitrary strategy σ_i for which a best response is computed. However, in contrast with IBR, FP maintains two sets, Σ_i and Σ_{-i}, with all best responses computed for each player. In each iteration of FP, one computes a best response to a

Algorithm 1: Bi-S

Require: Game G, DSL D, evaluation U, strategy σ_{-i}.
Ensure: Approximated best response σ_i of σ_{-i}.

1: Initialize empty library L of programs. *#see text*
2: Initialize dictionary T mapping a feature to a value.
3: $p_b \leftarrow$ **None**
4: **while not** reached time limit **do**
5: $V \leftarrow$`Feature-Search`(T) *#NaïveSampling*
6: $p \leftarrow \text{argmin}_{p \in L} \sum_i^{|\mathcal{F}|} |F(p)[i] - V[i]|$
7: $p, P \leftarrow$`Local-Search`$(G, D, p, V, U, \sigma_{-i})$ *#SA*
8: Update $T[F(p')]$ with $U(p')$ for all $p' \in P$
9: $L \leftarrow L \cup P$
10: **if** $U(p) > U(p_b)$ **then**
11: $p_b \leftarrow p$
12: **return** p_b

mixed strategy whose support is formed by all strategies in Σ_i (or Σ_{-i}). The distribution of strategies in Σ_i and Σ_{-i} converges to a mixed-strategy Nash equilibrium profile for the game. In the context pure programmatic strategies, we run FP until a time limit is reached and return the last best response obtained to each set as the algorithm's approximation of a pure-strategy Nash equilibrium profile.

IBR is computationally cheaper than FP because the latter evaluates all strategies in Σ_i and Σ_{-i} while learning. However, IBR can fail to progress toward a Nash equilibrium profile because it might repeatedly generate the same sequence of strategies in scenarios where A is a best response of B, B is a best response of C, and C is a best response of A. Learning with FP does not generate such sequences because FP best responds to all strategies seen thus far in the process.

IBR and FP require the computation of best responses. We design Bi-S to derive approximated best responses in the context of self-play algorithms such as IBR and FP.

Bilevel Synthesis (Bi-S)

Learning algorithms such as IBR and FP offer a function to guide the synthesis process. That is, given a strategy σ_{-i}, which is either defined by a single strategy in the context of IBR or by a mixture of strategies in the context of FP, a guiding function is defined as the utility of a program p encoding a strategy for player i against σ_{-i}, i.e., $U(s_{\text{init}}, p, \sigma_{-i})$. Such a guiding function provides sparse signals as small modifications to a losing program might not turn it into a winning one. As a result, the optimization landscape induced by these functions tends to have large plateaus. Bi-S attempts to induce an optimization landscape that is more amenable to local search algorithms by searching in two spaces: the space of programs and the space of features of the game.

Let $\mathcal{F}$ denote a set of features for a set of states of the game. Each feature $f \in \mathcal{F}$ defines a function mapping a set of states to an integer. As an example, a feature could count the maximum number of pieces of a given type a player controlled throughout a match of a board game. Let F be a function that receives an initial state s_{init}, a program p, and an opponent σ_{-i} and returns a vector with one entry for each feature $f \in \mathcal{F}$ for the set of states encountered in a game played between p and σ_{-i}. We denote as $F(s_{\text{init}}, p, \sigma_{-i})[j]$ the value of the j-th entry of this vector and we write $F(p)[j]$ whenever s_{init} and σ_{-i} are clear from the context.

Each iteration of Bi-S's search in the feature space defines a vector V of feature values, which is passed to the search in the program space. The search in the program space attempts to primarily find a program p that maximizes $U(s_{\text{init}}, p, \sigma_{-i})$. In case of ties between programs p and p' in terms of U-values, Bi-S minimizes the difference between V and the vectors $F(p)$ and $F(p')$. For example, in case the target strategy σ_{-i} defeats both p and p', i.e., $U(s_{\text{init}}, p, \sigma_{-i}) = U(s_{\text{init}}, p', \sigma_{-i})$, Bi-S's search will prefer the program whose feature values are more similar to V.

Once the search in the program space reaches a time limit and returns a program p approximating a best response to σ_{-i}, we use the value of $U(s_{\text{init}}, p, \sigma_{-i})$ to update statistics about the feature values of $F(p)$, so that the search in the feature space can learn the combination of feature values that are more likely to maximize $U(s_{\text{init}}, p, \sigma_{-i})$. In our implementation of Bi-S we treat the search problem in the feature space as a combinatorial multi-armed bandit (CMAB) problem (Gai, Krishnamachari, and Jain 2010) and use the two-phase Naïve Sampling algorithm (NS) (Ontañón 2017a) for searching over the space of possible feature vectors. We use Simulated Annealing (SA) (Kirkpatrick, Gelatt, and Vecchi 1983), a local search algorithm, to search over the space of programs. We describe both NS and SA below.

Bi-S's Pseudocode

Algorithm 1 shows Bi-S's pseudocode. Bi-S is invoked to approximate a best response to a target strategy σ_{-i} in each iteration of a learning algorithm such as IBR and FP. Bi-S receives the game G, DSL D, evaluation function U, which is defined based on the learning algorithm, and a strategy σ_{-i}. Bi-S returns an approximated best response to σ_{-i} encoded in a program p. Bi-S uses two data structures: a set L of programs encountered in search and a dictionary T that stores statistics about the features evaluated in search.

The set L is a library of programs that are used to initialize the local search in the space of programs. Given that the search in the feature space returns a vector V (line 5), Bi-S searches for the program $p \in L$ whose feature vector $F(p)$ is most similar to V in terms of absolute difference of the feature values (line 6). The program p is used to initialize the local search in the program space (line 7). By initializing the search with p, we allow the search to start in a location of the program space where it is more likely to encounter a program that will match the feature values in V.

The local search returns a program p that approximates a best response to σ_{-i}; it also returns the set of all programs P considered in search (line 7). We update the statistics the feature-space search uses for each program in P (line 8) and we add all programs in P to the library of programs (line 9). Once Bi-S reaches a time limit, it returns the program with best U-value it has encountered in search (lines 10–12).

Using Library of Programs to Transfer Learning

If the library L of programs Bi-S accumulates is reused in other calls of the algorithm, one is able to transfer some of

the knowledge generated in search across calls of Bi-S. We initialize the library L with an empty set only in the first call of Bi-S. Later calls of Bi-S within an single run of either IBR or FP will reuse the library L accumulated in previous calls to Bi-S. The library of programs allows the local search to start in a location of the program space that the search in the feature space deems as promising. For example, if the search in the feature space discovers that feature f_j leads to high U-values, then the search in the program space is able to start in a region where f_j is more likely to occur. The library of programs allows Bi-S to effectively "jump" around the program space according to the feature-space search guidance.

We experiment with a version Bi-S that reuses L across similar, but different games. We call such a version Bi-S+.

Two-Phase Naïve Sampling (NS)

We model the problem of searching in the feature space as a combinatorial multi-armed bandit (CMAB) problem. A tuple (X, μ) defines a CMAB problem, where

- $X = \{X_1, \cdots, X_n\}$. Each X_j is a variable that can assume K_j distinct values $\mathcal{X}_j = \{v_j^1, \cdots, v_j^{K_j}\}$. The set $\mathcal{X} = \{(v_1, \cdots, v_n) \in \mathcal{X}_1 \times \cdots \times \mathcal{X}_n\}$ is the possible combinations of value assignments for variables in X. A value assignment $V \in \mathcal{X}$ is called a macro-arm.
- $\mu : \mathcal{X} \rightarrow \mathbb{R}$ is a utility function, that receives a macro-arm and returns a utility value for that macro-arm.

In a CMAB problem one attempts to find the macro-arm with largest expected utility value. In Bi-S, each variable X_j represents a feature in $\mathcal{F}$ and $\mathcal{X}_j = \{v_j^1, \cdots, v_j^{K_i}\}$ is the set of K_j integer values the X_j can receive. A macro-arm V is a feature vector and $\mu(V)$ is the average U-value of the programs p the search in the program space returns when it receives V as input.

Naïve Sampling (NS) was designed to deal with CMAB problems with a number of macro-arms that is so large that one cannot exhaustively evaluate all of them in search (Ontañón 2017a). NS divides a CMAB problem with n variables in $n + 1$ multi-armed bandit (MAB) problems.

- n local MABs, one for each variable $X_j \in X$. For variable X_j representing a feature, the arms of the MAB are the K_j values in $\mathcal{X}_j$.
- 1 global MAB, denoted MAB_g, that treats each macro-arm A evaluated in search as an arm in MAB_g; MAB_g has no arms in the beginning of search.

At each iteration, NS uses a policy π_0 to determine whether it adds an arm to MAB_g through the local MABs (explore) or evaluates an existing arm in MAB_g (exploit).

1. If NS chooses to explore, a macro-arm A is added to MAB_g by using a policy π_l to independently choose a value for each variable in X (local MAB). NS assumes that the utility of a macro-arm V can be approximated with the sum of the utilities of the individual values $v_i \in V$, i.e., $\mu(V) \approx \sum_{v_i \in V} \mu'(v_i)$, where $\mu'(v_i)$ is the average of the $\mu(V)$-values for which the i-th value in V is v_i.
2. If NS chooses to exploit, then a policy π_g is used to select an existing macro-arm in MAB_g.

We use ϵ-greedy for policies π_0, π_l, and π_g. We also use the two-phase version of NS where the first k iterations we use a set of ϵ-values (ϵ_0^1, ϵ_l^1, and ϵ_g^1) that allows for a more aggressive exploration; the algorithm uses another set of ϵ-values in the remaining steps (ϵ_0^2, ϵ_l^2, and ϵ_g^2) that allows NS to exploit the macro-arms collected in the first phase. We multiply all μ'-values by γ, with $0 < \gamma < 1$, after each call to NS. This decaying factor allows the search in the feature space to be more biased by the U-values of programs encountered in more recent iterations.

Simulated Annealing

SA uses a temperature parameter to control the greediness of the search, so it behaves similarly to a random walk in the beginning of the search, when the temperature is high, and similarly to hill climbing later in search, when the temperature is low. The SA search starts with an initial program, which can be provided as input (e.g., a program from the library L) or, alternatively, be randomly generated according to the rules of the grammar. When the library is empty, SA starts with a randomly generated program. A program can be randomly generated by starting with the grammar's initial symbol I and replacing it with a random production rule of I; then, we repeatedly replace a non-terminal symbol of the program with a valid and randomly selected production rule, until the program contains only terminal symbols.

Once the initial program p is defined, in each iteration of search, SA generates a neighbor p' of p by changing a subtree in p's AST. SA randomly chooses a non-terminal symbol n in the AST (all non-terminal nodes of the AST are chosen with equal probability). Then, SA replaces the subtree rooted at n with a subtree that is generated with the same procedure used to generate a random program. SA decides if it accepts or rejects the neighbor p'. If it accepts, then p' is assigned to p and the process is repeated—SA performs a walk in the program space. If it rejects, SA repeats the procedure by generating another neighbor of p. The probability in which SA accepts p' is given by the following equation.

$$\min\left(1, \exp\left(\frac{\beta \cdot (\Psi(p') - \Psi(p))}{T_j}\right)\right),$$

where, T_j is the temperature at iteration j, and Ψ is the evaluation function. In the context of Bi-S, Ψ is initially U and, in case of a tie between p and p', we use a function based on the vector V of features Bi-S provides as input to SA. Namely, we maximize the similarity between the program's p feature values and V according to the following function.

$$\sum_{j=1}^{|\mathcal{F}|} 1 - \frac{|V[j] - F(p)[j]|}{\max(V[j], F(p)[j])}$$

If $\Psi(p') \geq \Psi(p)$, then SA accepts p'. Otherwise, the probability of acceptance depends on T_j and β. β is an input parameter that adjusts SA's greediness. Larger values of β result in a greedier search as SA is more likely to reject programs with small Ψ-values. Larger values of T_j make the

search less greedy because SA becomes more likely to accept worse neighbors. The initial temperature, T_1, is an input parameter and T_j is computed according to the schedule $T_j = \frac{T_1}{(1+\alpha \cdot j)}$, where α is also an input parameter. Once the temperature value is smaller than a threshold value ϵ, SA returns the program with largest U value encountered in search. In case of a tie in terms of U-values, SA returns the program whose feature vector is the most similar to the target vector V.

Empirical Evaluation

In this section we evaluate the hypothesis that Bi-S's bilevel search allows for a faster synthesis of effective programmatic strategies. All experiments were run on a single 2.4 GHz CPU with 8 GB of RAM and a time limit of 2 days.[1] We denote our methods as IBR(Bi-S), FP(Bi-S), IBR(Bi-S+), FP(Bi-S+), depending on the learning algorithm and the version of the bilevel synthesis.

We use $\alpha = 0.9$, $\beta = 200$, $T_1 = 100$, $\epsilon = 1$ with our SA implementation as these are the values Medeiros, Aleixo, and Lelis (2022) used in their experiments. We use the following exploration rates for the first phase of NS: $\epsilon_0^1 = 0.7$, $\epsilon_l^1 = 0.7$, and $\epsilon_g^1 = 0.4$; and the following for the second phase: $\epsilon_0^1 = 0.1$, $\epsilon_l^1 = 0.3$, and $\epsilon_g^1 = 0.1$. We use $\gamma = 0.95$. These parameters allow NS to explore more in the first phase and exploit the learned macro-arms in the second. We run each phase for 1 day. All baselines we use in our experiments that also synthesize programmatic strategies are also given the time limit of 2 days.

Problem Domain: MicroRTS

MicroRTS is a real-time strategy game widely used to evaluate intelligent systems (Ontañón 2020). It is a two-player zero-sum game where each player controls a set of units in real time (each player has 100 milliseconds to decide on their next action) The units gather resources and build structures, which are used to train more units that will eventually battle the opponent. MicroRTS has the following types of units: Worker, Ranged, Heavy, and Light, in addition to the Barracks and Base structures. All units are able to battle the opponent, but only Workers can build structures and collect resources; the units differ in how much damage they can suffer and cause in battle. A Base can train Workers and store resources, while a Barracks can train the other units.

We use MicroRTS because it is a challenging domain and the winners of the recent competitions are programmatic strategies written by programmers. Thus we can evaluateBi-S against strategies programmers wrote with the intent of winning the competition (and its monetary prize). Moreover, MicroRTS can be played on different maps and each map might require a different strategy. Thus, with the same implementation we can evaluate Bi-S on similar, but different games. We use the maps basesWorkers24x24A, basesWorkers32x32A, and (4)BloodBath.scmB, whose sizes are 24×24, 32×32, and 64×64, respectively, from MicroRTS's official code base. The bigger map is from the commercial game StarCraft. The maps are shown in the supplementary materials. We consider two starting locations for each player in each map; we run two games alternating the players' starting location to ensure a fair evaluation. We synthesize one program for each location of the map for all synthesizers used in our experiments (Bi-S and baselines).

For Bi-S+ we use the library L of one of the maps to synthesize a strategy for another map. Namely, we use the library generated while synthesizing a strategy with Bi-S for map 24×24 to initialize the Bi-S+ synthesis process for map 32×32 and 64×64, and the Bi-S library for 32×32 to initialize the Bi-S+ synthesis process for map 24×24.

The set of features over set of states we use are: number of Worker, Ranged, Heavy, and Light units trained in all states in the set; the number Bases and Barracks built, and the number of resources collected across all states.

We use Medeiros, Aleixo, and Lelis (2022)'s DSL for MicroRTS. Their DSL supports if-then-else structures and loops.

Baselines

The main baselines for our experiments are the synthesizers that search only in the program space. We use IBR and FP with SA to approximate a best response in each iteration of the algorithms, we denote them as FP(SA) and IBR(SA).

We also evaluate the programmatic strategies Bi-S synthesizes with several existing methods, which we will refer to as the "competition set." This set of agents includes the winners of the two latest MicroRTS competitions, COAC and Mayari. We also use the Sketch-SA (Medeiros, Aleixo, and Lelis 2022) method where it uses either COAC or Mayari as oracles for learning a program sketch; we denote them as IBR(Sketch(COAC)), FP(Sketch(COAC)), IBR(Sketch(Mayari)), FP(Sketch(Mayari)) according to the learning algorithm and oracle used. We also consider the agents MentalSeal, UmSbot, Droplet, Rojo, GuidedRojoA3N, and Alet from past competitions (Ontañón 2017b). We also use in our evaluation the search algorithms Portfolio Greedy Search (PGS) (Churchill and Buro 2013), Stratified Strategy Search (SSS) (Lelis 2017), A3N (Moraes et al. 2018), and Strategy Tactics (STT) (Barriga, Stanescu, and Buro 2018). We also use programmatic strategies that are often used as baselines in the MicroRTS competition: WR, HR, RR, and LR (Barriga, Stanescu, and Buro 2017).

Comparison with IBR(SA) and FP(SA)

The top row of Figure 2 shows the learning curves for Bi-S and SA when using IBR and FP. The bottom row shows the learning curves for Bi-S+. These plots are generated as follows: we select the strategy returned by each approach every two hours of computation for each method and perform a round robin evaluation with these four strategies. The winning rate is computed as the total number of victories plus half for each draw, divided by three. Each curve shows the average winning rate over 10 independent runs of each system and the shaded area shows the standard deviation.

FP(Bi-S) and FP(Bi-S+) are superior to FP(SA) as the solid blue line is above the dashed orange one. Similar pattern is observed for the dotted green line, which represents

[1] https://github.com/dsaleixo/BilevelSearchforSynthesizing.

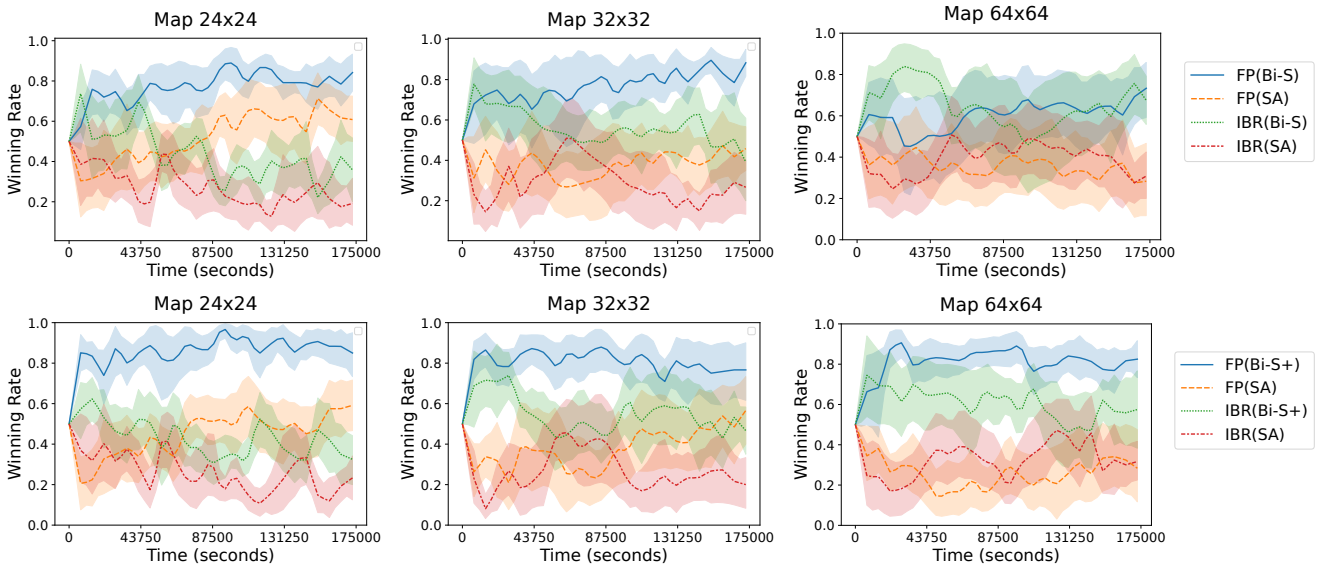

Figure 2: Learning curves for Bi-S (top row), Bi-S+ (bottom row) and SA with IBR and FP as learning algorithms. The strategy of an algorithm is evaluated in terms of winning rate against the strategies of the other three algorithms in a given time.

IBR(Bi-S) and IBR(Bi-S+), with respect to the pointed and dashed red line, which represents IBR(SA). Bi-S+ seems to perform better when used with FP; the solid blue lines reach higher winning rate earlier in the plots of the bottom row than in the plots of the top row. By contrast, there seems to be no noticeable difference between IBR(Bi-S) and IBR(Bi-S+). We conjecture that the difference between IBR and FP with Bi-S+ is due to FP providing a better guiding function. In the case of the IBR(Bi-S) and IBR(Bi-S+) results, we conjecture that a better search algorithm (Bi-S+) might not necessarily lead to stronger strategies due to limitations of the guiding function IBR provides.

Figure 3 shows the learning curve of the same algorithms shown in Figure 2, but the winning rate is computed by having the strategies the systems synthesize play against all strategies in our competition set. The lines show the average winning rate across 10 independent runs of the systems. In general, all lines follow an upward trend with computation, which demonstrates that the strategies these systems synthesize become stronger even against strategies that are not in the set of strategies that can be encoded in a program written using D (e.g., strategies derived with tree search algorithms).

Some of the trends observed in Figure 2 are also observed in Figure 3. Namely, the Bi-S and Bi-S+ lines are above the SA lines for a fixed learning algorithm. Also, transfer learning allows for a faster synthesis of stronger strategies with FP. It is also noticeable how the standard deviation of FP(Bi-S+) is smaller than that of FP(Bi-S). Finally, transfer learning does not seem to improve the results if IBR is used.

Agents	**Maps**			**Total**
	24×24	32×32	64×64	
IBR(Bi-S+)	0.85	0.92	0.89	0.89
FP(Bi-S+)	0.95	0.85	0.85	0.88
IBR(Sketch(COAC))	0.70	0.83	0.87	0.80
FP(Sketch(COAC))	0.78	0.81	0.76	0.78
COAC	0.78	0.80	0.73	0.77
IBR(Sketch(Mayari))	0.69	0.76	0.84	0.76
FP(Sketch(Mayari))	0.66	0.89	0.71	0.75
Mayari	0.87	0.70	0.65	0.74
FP(SA)	0.69	0.64	0.63	0.65
IBR(SA)	0.67	0.50	0.75	0.64
MentalSeal	0.67	0.51	0.70	0.63
UMSBot	0.65	0.63	0.54	0.61
Droplet	0.47	0.50	0.61	0.53
PGS	0.41	0.42	0.39	0.41
HR	0.38	0.51	0.30	0.39
SSS	0.39	0.38	0.32	0.36
LR	0.40	0.40	0.26	0.35
STT	0.24	0.30	0.42	0.32
RR	0.21	0.23	0.17	0.21
Rojo	0.15	0.09	0.25	0.16
WR	0.13	0.17	0.17	0.16
GuidedRojoA3N	0.13	0.04	0.17	0.11
A3N	0.11	0.10	0.02	0.08
RandomBiasedAI	0.01	0.03	0.02	0.02

Table 1: Tournament results in terms of winning rate.

Tournament Evaluation

We also test FP(Bi-S+) and IBR(Bi-S+) in a tournament setting where we perform a round robin evaluation of all strategies in the tournament set. For the synthesizer-based systems (Bi-S+, Sketch, and SA) we select the programs used in the tournament as follows. We collect the last 20 best re-

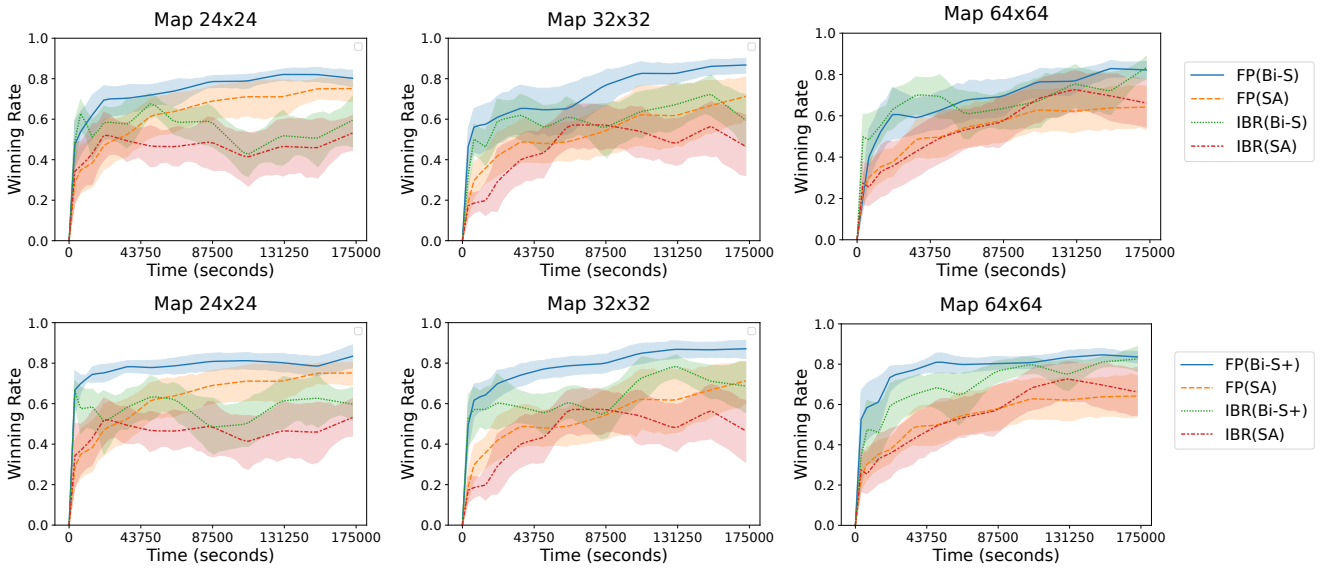

Figure 3: Learning curves for Bi-S (top row), Bi-S+ (bottom row) and SA with IBR and FP as learning algorithms. The strategy of an algorithm is evaluated against all strategies in our competition set of strategies.

sponses (either in the context of IBR or FP) and perform a round robin tournament among them. The strategy with highest winning rate for each starting location of the map in this "internal" tournament was used in the tournament with the competition set.

Table 1 presents the winning rate of the tournament. IBR(Bi-S+) and FP(Bi-S+) had the highest total winning rate, the Sketch approaches imitating COAC had the third and fourth highest total winning rate of the tournament. FP(Bi-S+) had the highest winning rate on the 24×24 map while IBR(Bi-S+) had the highest one on the 32×32 and 64×64 maps. Interestingly, while FP performed better than IBR when using Bi-S+ in the results shown in Figures 2 and 3, they performed equally well in the tournament. This is likely due to the way that we selected the programs to take part in the tournament. By running a round robin tournament with the last 20 best responses of IBR and selecting the best performing program, we are performing with IBR an operation that is similar to the one used in FP, where each strategy plays against a set of other strategies.

FP(SA) and IBR(SA) performed reasonably well in the tournament. They were outperformed only by the Bi-S+ and the Sketch approaches, and by COAC and Mayari, the winners of the last two competitions. Nevertheless, the difference in terms of winning rate between Bi-S+ and SA is substantial (0.89 to 0.65). These results also confirm a trend observed in the last two MicroRTS competitions, where programmatic strategies have outperformed tree search agents.

The results shown in Figures 2 and 3 and in Table 1 support our hypothesis that the guiding function our bilevel method uses provides a better search signal, in the context of synthesis with SA, than the signal provided by a function that relies only on self-play algorithms.

A limitation of Bi-S is that it requires a set of feature functions as input. An interesting direction of future research is to use a language such as Game Description Language (GDL) (Love, Genesereth, and Hinrichs 2006) to encode the game, such that the system can reason about the game and automatically discover the set of functions for computing the features. The search for such functions can be seen as the addition of another layer in Bi-S's search.

Conclusions

In this paper we introduced Bi-S, a bilevel search algorithm for synthesizing programmatic strategies. Current methods for synthesizing such strategies often suffer from a weak signal for guiding the search because self-play algorithms can only inform the search algorithm of the utility of the game. While small changes to a winning program might not turn it into a winning one, they might represent progress in the right direction. Our method searches over a space of game features while attempting to learn the relation between feature values and the strength of a strategy. The search in the feature space is then able to bias the search in the programmatic space by defining a set of features that should be observed in games played by the programs. This set of "desirable features" provides an extra search signal for the synthesis. We performed an evaluation of Bi-S in MicroRTS and our results showed that Bi-S is able to synthesize stronger strategies than the baselines that search only in the programmatic space. Moreover, two versions of Bi-S outperformed 22 other agents, including programmatic strategies written by human programmers and search algorithms in a simulated tournament.

Acknowledgements

This research was partially supported by FAPEMIG, CAPES, and Canada's NSERC and CIFAR AI Chairs pro-

gram. The research was carried out using computational resources from Compute Canada. We thank the anonymous reviewers for their feedback. L. Lelis was on leave from Universidade Federal de Viçosa while this research was carried out.

References

Aguas, J. S.; Jiménez, S.; and Jonsson, A. 2018. Computing Hierarchical Finite State Controllers With Classical Planning. *Journal of Artificial Intelligence Research*, 62: 755–797.

Barriga, N. A.; Stanescu, M.; and Buro, M. 2017. Combining Strategic Learning and Tactical Search in Real-Time Strategy Games. *Thirteenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Barriga, N. A.; Stanescu, M.; and Buro, M. 2018. Game Tree Search Based on Nondeterministic Action Scripts in Real-Time Strategy Games. *IEEE Transactions on Games*, 10(1): 69–77.

Bastani, O.; Pu, Y.; and Solar-Lezama, A. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *Advances in Neural Information Processing Systems*, 2499–2509.

Bonet, B.; Palacios, H.; and Geffner, H. 2010. Automatic Derivation of Finite-State Machines for Behavior Control. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1656–1659. AAAI Press.

Brown, G. 1951. Iterative solution of games by Fictitious Play, 1951. *Activity Analysis of Production and Allocation (TC Koopmans, Ed.)*, 374–376.

Churchill, D.; and Buro, M. 2013. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *Proceedings of the Conference on Computational Intelligence in Games*, 1–8. IEEE.

Cropper, A.; Evans, R.; and Law, M. 2020. Inductive general game playing. *Machine Learning*, 109(7): 1393–1434.

Gai, Y.; Krishnamachari, B.; and Jain, R. 2010. Learning multiuser channel allocations in cognitive radio networks: A combinatorial multi-armed bandit formulation. In *New Frontiers in Dynamic Spectrum, 2010 IEEE Symposium on*, 1–9. IEEE.

Heinrich, J.; Lanctot, M.; and Silver, D. 2015. Fictitious self-play in extensive-form games. In *International conference on machine learning*, 805–813. PMLR.

Hu, Y.; and De Giacomo, G. 2013. A Generic Technique for Synthesizing Bounded Finite-State Controllers. *Proceedings of the International Conference on Automated Planning and Scheduling*, 23(1): 109–116.

Kirkpatrick, S.; Gelatt, C. D.; and Vecchi, M. P. 1983. Optimization by Simulated Annealing. *Science*, 220(4598): 671–680.

Lanctot, M.; Zambaldi, V.; Gruslys, A.; Lazaridou, A.; Tuyls, K.; Pérolat, J.; Silver, D.; and Graepel, T. 2017. A Unified Game-Theoretic Approach to Multiagent Reinforcement Learning. In *Proceedings of the International Conference on Neural Information Processing Systems*, 4193–4206.

Lelis, L. H. S. 2017. Stratified Strategy Selection for Unit Control in Real-Time Strategy Games. In *International Joint Conference on Artificial Intelligence*, 3735–3741.

Love, N.; Genesereth, M.; and Hinrichs, T. 2006. Mobile Systems IV. Technical report, Stanford University.

Mariño, J. R. H.; Moraes, R. O.; Oliveira, T. C.; Toledo, C.; and Lelis, L. H. S. 2021. Programmatic Strategies for Real-Time Strategy Games. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(1): 381–389.

Medeiros, L. C.; Aleixo, D. S.; and Lelis, L. H. S. 2022. What can we Learn Even From the Weakest? Learning Sketches for Programmatic Strategies. In *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press.

Moraes, R. O.; Mariño, J. R. H.; Lelis, L. H. S.; and Nascimento, M. A. 2018. Action Abstractions for Combinatorial Multi-Armed Bandit Tree Search. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 74–80. AAAI.

Ontañón, S. 2017a. Combinatorial Multi-armed Bandits for Real-Time Strategy Games. *Journal of Artificial Intelligence Research*, 58: 665–702.

Ontañón, S. 2017b. MicroRTS Competition. https://sites.google.com/site/micrortsaicompetition/. [Online; accessed 15-August-2022].

Ontañón, S. 2020. Results of the 2020 MicroRTS Competition. https://sites.google.com/site/micrortsaicompetition/competition-results/2020-cog-results. Accessed: 2021-09-30.

Qiu, W.; and Zhu, H. 2022. Programmatic Reinforcement Learning without Oracles. In *International Conference on Learning Representations*.

Silver, T.; Allen, K. R.; Lew, A. K.; Kaelbling, L. P.; and Tenenbaum, J. B. 2019. Few-Shot Bayesian Imitation Learning with Logical Program Policies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 10251–10258. AAAI Press.

Srivastava, S.; Immerman, N.; Zilberstein, S.; and Zhang, T. 2011. Directed Search for Generalized Plans Using Classical Planners. In *Proceedings of the International Conference on Automated Planning and Scheduling*. AAAI.

Van Deursen, A.; Klint, P.; and Visser, J. 2000. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6): 26–36.

Verma, A.; Le, H.; Yue, Y.; and Chaudhuri, S. 2019. Imitation-Projected Programmatic Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 32, 1–12. Curran Associates, Inc.

Verma, A.; Murali, V.; Singh, R.; Kohli, P.; and Chaudhuri, S. 2018. Programmatically Interpretable Reinforcement Learning. In *Proceedings of the International Conference on Machine Learning*, 5052–5061.