What can we Learn Even From the Weakest? Learning Sketches for Programmatic Strategies

Leandro C. Medeiros,[1*] **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

In this paper we show that behavioral cloning can be used to learn effective sketches of programmatic strategies. We show that even the sketches learned by cloning the behavior of weak players can help the synthesis of programmatic strategies. This is because even weak players can provide helpful information, e.g., that a player must choose an action in their turn of the game. If behavioral cloning is not employed, the synthesizer needs to learn even the most basic information by playing the game, which can be computationally expensive. We demonstrate empirically the advantages of our sketch-learning approach with simulated annealing and UCT synthesizers. We evaluate our synthesizers in the games of Can't Stop and MicroRTS. The sketch-based synthesizers are able to learn stronger programmatic strategies than their original counterparts. Our synthesizers generate strategies of Can't Stop that defeat a traditional programmatic strategy for the game. They also synthesize strategies that defeat the best performing method from the latest MicroRTS competition.

Introduction

One needs to search in large program spaces to synthesize effective programmatic strategies. In addition to dealing with large spaces, synthesizers often lack effective functions for guiding the search. This is in contrast with neural methods, where gradient information is available to guide the search. The problem of neural methods is their lack of interpretability. Despite being elusive, we can often understand, verify, and even manually modify programmatic strategies.

In this paper we show that behavioral cloning (Bain and Sammut 1996) can be used to learn program sketches (Solar-Lezama 2009) to speed up the synthesis of strong programmatic strategies. Sketches are incomplete programs that serve as starting points for synthesis. We investigate the use of this sketch learning approach with synthesizers employing Simulated Annealing (SA) (Kirkpatrick, Gelatt, and Vecchi 1983) and UCT (Kocsis and Szepesvári 2006) as search algorithms and evaluate them in the context of computing a best response for a target strategy in two-player zero-sum games. Specifically, we evaluate our methods in the board game of Can't Stop and in the real-time strategy (RTS) game of MicroRTS. We show that our methods can be effective even when cloning the behavior of weak players, such as a player that chooses their actions at random for the game of Can't Stop. Our sketch learning method can be effective when the cloned strategy is weak because even such strategies might convey information that is helpful for the synthesis of programmatic strategies, such as the program structure required to decide when to stop playing in Can't Stop or when to build a specific structure in MicroRTS.

We evaluated our sketch-based SA and UCT methods by synthesizing approximated best responses to a known programmatic strategy for Can't Stop (Glenn and Aloi 2009) and to COAC, the winner of the latest MicroRTS Competition (Ontañón 2020), on four maps. Our sketch-based SA synthesized strong strategies in all settings tested. As a highlight, it synthesized a strategy that defeats COAC on large maps of MicroRTS; none of the strategies synthesized by the baselines were able to defeat COAC on large maps.

Related Works

Our work is related to methods for synthesizing programmatic policies, in particular those that use some form of behavioral cloning such as imitation learning (Schaal 1999). Bastani, Pu, and Solar-Lezama (2018) present an algorithm that uses a variant of the imitation learning algorithm Dagger (Ross, Gordon, and Bagnell 2011) to distill a high-performing neural policy into an interpretable decision tree.

Verma et al. (2018) use Dagger and a neural policy to help with the synthesis of programmatic policies. The actions the neural policy chooses on a set of states are used in a Bayesian optimization procedure for finding suitable constant values for the programmatic policies. Verma et al. (2019) use a similar approach, but the neural model is trained so that it is not "too different" from the synthesized policies, with the goal of easing the optimization task.

We differ from previous work in that we do not assume the oracle is available for queries as in Dagger-like methods. We also do not assume that the oracle is a neural network as Verma et al. (2019) and Bastani, Pu, and Solar-Lezama (2018) do. For example, in our experiments we use a data set from a human oracle. We also do not require the strategy to be cloned to be high performing; we are able to learn effective sketches even from weak strategies.

Mariño et al. (2021) introduced Lasi, a method that uses

[*] Equal contribution.

behavioral cloning to simplify the language used for synthesis. Lasi removes from the language the instructions that are not needed to clone a strategy. Lasi can be used with our sketch-learning methods by simplifying the language to only then learn a sketch. Lasi is not as general as our methods because it cannot be applied to domains in which all symbols in the language are needed, such as Can't Stop.

Others have synthesized programs to serve as evaluation functions (Benbassat and Sipper 2011), but not to serve as complete strategies. Others explored the synthesis of strategies for cooperative games (Canaan et al. 2018) and single-agent problems (Butler, Torlak, and Popović 2017; De Freitas, de Souza, and Bernardino 2018). These methods can potentially benefit from our sketch-learning methods.

While most previous work assume that the user provides the program sketch (Solar-Lezama 2009), Nye et al. (2019) use a neural model to generate sketches. Their approach is designed to solve program synthesis tasks, where one synthesizes a program mapping a set of input values to the desired output values; we synthesize strategies. Also, we are unable to train a neural model for sketch generation because the amount of data we consider is insufficient for training (we use data sets with state-actions of as few as 3 matches).

Problem Definition

Let G be a sequential two-player zero-sum game defined by a set S of states, a pair of players $P = \{i, -i\}$, 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 G 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. The value of the game for state s is denoted by $U(s, \sigma_i, \sigma_{-i})$, which returns the utility of player i if i and $-i$ follow the strategies given by σ_i and σ_{-i}. We also call a match a game played between two strategies.

We consider programmatic strategies written in a domain-specific language (DSL) (Van Deursen, Klint, and Visser 2000). Let D be a DSL and $[\![D]\!]$ be the set of programs written in D. The best response for a strategy σ_{-i} in $[\![D]\!]$ is a strategy that maximizes player i's utility against σ_{-i}, i.e., $\max_{\sigma_i \in [\![D]\!]} U(s_{init}, \sigma_i, \sigma_{-i})$. The computation of a best response for a fixed strategy is a basic operation in game theory approaches such as iterated best response for approximating a Nash equilibrium profile (Lanctot et al. 2017).

In this paper we evaluate different search methods for synthesizing a best response to a strategy. We provide a game G, a DSL D and a strategy σ_{-i} and the synthesizer searches in the space defined by D and returns an approximated best response σ_i in $[\![D]\!]$ to σ_{-i}. We also consider the setting in which a data set with state-action pairs $L = \{(s_j, a_j)\}_{j=1}^{m}$ with the actions a_j a player takes at states s_j is available.

Synthesis of Programmatic Strategies

In this section we review DSLs and explain how SA and UCT can be used to synthesize programmatic strategies. While SA and UCT have been applied to program synthesis tasks, e.g., (Husien and Schewe 2016; Cazenave 2013), this is the first time these approaches are applied to synthesize programmatic strategies, so we describe them in detail.

Domain-Specific Languages

A DSL is defined as a context-free grammar (V, Σ, R, I), where V, Σ, 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 $V = \{I, C, B\}$, $\Sigma = \{c_1, c_2, b_1, b_2 \text{ if, then}\}$, R

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

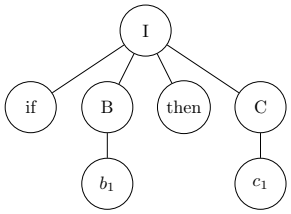

Figure 1: DSL (left) and AST for "if b_1 then c_1" (right).

are the relations (e.g., $C \rightarrow c_1$), and I is the start symbol.

The DSL allows programs with a single command (c_1 or c_2) and programs with branching. We represent programs as abstract syntax trees (AST), where the root of the tree is I, the internal nodes are non-terminals and leaf nodes are terminals. Figure 1 shows an example of an AST, where leaves are terminal symbols and internal nodes are non-terminals.

Simulated Annealing for Synthesis of Strategies

SA is a local search algorithm that uses a temperature parameter to control the greediness of the search. SA behaves like a random walk in the beginning of the search and more like hill climbing with time. We use SA to approximate a programmatic best response to a target strategy σ_{-i}, i.e., SA approximates a solution to $\arg\max_{\sigma_i \in [\![D]\!]} U(s_{init}, \sigma_i, \sigma_{-i})$.

SA starts with a program that is randomly generated as follows. We start with I and we replace it with a randomly chosen production rule for I; we then repeatedly replace a non-terminal symbol in the generated program with a random and valid production rule; we stop when the program contains only terminals. For example, the production rules used to obtain program "if b_1 then c_1" are: $I \rightarrow \text{if}(B)$ then C else C; $B \rightarrow b_1$; $C \rightarrow c_1$.

Once the initial program p is defined, SA generates a neighbor p' of p by changing a subtree in p's AST. We randomly choose a non-terminal symbol n in the AST (all non-terminal symbols can be chosen with equal probability) and we replace the subtree rooted at n with a subtree that is generated with the same procedure used to generate the initial program. For example, if the subtree rooted at C (Figure 1) is chosen and we replace it with c_2, then p' is "if b_1 then c_2." SA decides if it accepts or rejects p'. If it accepts, then p' is assigned to p and the process is repeated. If it rejects, SA repeats the procedure by generating another neighbor of p. The probability in which SA accepts p' is given by

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

Here, T_j is the temperature at iteration j, and Ψ is an evaluation function. In program synthesis tasks, $\Psi(p)$ counts the number of input examples that p correctly maps to the desired output (Alur et al. 2013). In the context of games, $\Psi(p)$ returns the utility of p against the opponent σ_{-i}. If $\Psi(p') \geq \Psi(p)$, then SA accepts p' with probability 1.0. Otherwise, the probability of acceptance depends on T_j and β. β is an input parameter that allows us to adjust how greedy SA is; larger values of β result in a greedier search by more often rejecting programs with small Ψ-values. Larger values of T_j make the search less greedy since large T-values increase the chances of accepting p'. The initial temperature, T_1, is an input parameter and T_j is computed according to the schedule $T_j = \frac{T_1}{(1+\alpha \cdot j)}$. Once the temperature becomes smaller than ϵ, we stop searching and the program with largest Ψ-value encountered in search is returned as the SA's approximated best response to σ_{-i}. In our experiments we run SA multiple times, while we have not exhausted the time allowed for synthesis, and we initialize the search with the program returned in the latest run as it often allows the search to start in a more promising region of the space.

UCT for Synthesis of Strategies

UCT grows a search tree while exploring the space. Each node in the tree represents a program, which can be complete or incomplete. We say that a program is complete if all leaves in its AST are terminals. The root of the UCT tree represents the incomplete program of the DSL's initial symbol I. The children of a node n in the UCT tree are the programs that can be generated by applying a production rule to the leftmost non-terminal symbol of the program n represents. For example, if n represents if(B) then C, then its children represent if(b_1) then C and if(b_2) then C because B is the leftmost non-terminal symbol of the program n represents.

UCT operates in four steps: selection, expansion, simulation, and backpropagation. The selection step starts at the root of the tree and it chooses the j-th child that maximizes $\bar{X}_j + C\sqrt{\frac{\log(N)}{N_j}}$. Here, $\bar{X}_j$ is the average evaluation value of the j-th child of n, N is the number of times node n was visited in previous selection steps, N_j is the number of times n was visited and the j-th child was selected, and C is an exploration constant. The first term of the equation is an exploitation term as it favors the child with highest average evaluation value; the second term is the exploration term.

The selection step stops when it encounters a node n with at least one child n' that is not in the UCT tree. In the expansion step, UCT adds n' to the tree; if more than one child is not in the UCT tree, the algorithm chooses one arbitrarily. The simulation step applies a policy to turn n' into a complete program, which is then evaluated with Ψ. Finally, the backpropagation step updates the X_j-values of all nodes visited in the selection step with the Ψ-value from the simulation step. The four steps are repeated multiple times and UCT returns the program with largest Ψ-value, among all programs evaluated during search, when it reaches a user-specified time limit. UCT caches the Ψ-values of programs that were evaluated in previous iterations of the algorithm. The UCT tree might grow to include complete programs (i.e., nodes with no children). If the selection step ends at a complete program, it performs no expansion and returns the cached Ψ-value of n in the backpropagation step.

We use a single run of SA as UCT's simulation policy. When running SA as simulation policy for an incomplete program p, the neighbors of a program can only be obtained by changing the subtrees of the AST that are rooted at a non-terminal leaf node. For example, if p is if(B) then C, then the neighbors of p can be obtained by changing only B and C, but not the root of the AST, I, because I is not a leaf in the AST. This constraint ensures that the simulation policy does not change the structure of p, which is defined by the production rules along the path in the UCT tree.

Learning Sketches with Behavioral Cloning

We consider the setting in which the synthesizer receives as input a data set of state-action pairs $L = \{(s_j, a_j)\}_{j=1}^{m}$ with actions chosen by strategy σ_o for states of one or more matches of the game. We use this data set to learn a sketch to speed up the synthesis of a programmatic strategy.

SA and UCT can be used to clone the behavior of σ_o by replacing Ψ with an evaluation function $C(L, p)$ that receives the data set L and a program p and returns a score of how well p clones σ_o. Note, however, that cloning the behavior of σ_o can results in weak strategies. This is because σ_o might not be represented in $[\![D]\!]$. Or the data set L is limited and one needs to perform Dagger-like queries (Ross, Gordon, and Bagnell 2011) to augment it and σ_o might not be available for such queries (e.g., σ_o is a human player who is unavailable). Or σ_o is a weak strategy and exactly cloning its behavior would result in a weak strategy. Instead of learning a strategy directly with behavioral cloning, we use it to learn a sketch that helps the synthesis process of a strong strategy.

Sketch-learning methods can be more effective than those that optimize for Ψ directly for two reasons. First, Ψ can be computationally more expensive than C. Using C to learn parts of the programmatic strategy will tend to be more efficient than to learn the entire strategy with Ψ. Second, the function C can offer a denser signal for search (e.g., the neighbor p' of p might not defeat σ_{-i}, but it might have a higher C-score, which can be helpful to guide the search).

Sketch Learning with UCT

We run UCT with the evaluation function $C(L, p)$ for a number of iterations and, whenever we find a complete program p with a C-value larger than the current best solution, we evaluate it with Ψ. We call this search the sketch-search. Once we reach a time limit, we use the program found in the sketch-search with the largest Ψ-value to initialize a second UCT search, which we call best response (BR)-search.

Let p be the program with largest Ψ-value encountered in the sketch-search. The program is defined by a sequence of production rules that replace the leftmost non-terminal symbol in the sequence of partial programs, starting with the initial symbol of the DSL. We start the UCT tree of the BR-search with a branch that represents the production rules of p. For example, let "if(b_1) then c_1" be the program p with largest Ψ-value from the sketch-search. The UCT tree of the

BR-search is initialized with the branch with nodes representing the programs: "I", "if(B) then C," "if(b_1) then C," and "if(b_1) then c_1." We then perform a backpropagation step on the added branch with $\Psi(p)$, which was computed in the sketch-search. By adding the branch leading to p to the tree of the BR-search we are biasing it to explore programs that share the structure of p. This is because the nodes along the added branch will likely have higher $\bar{X}$-values than other branches, specially in the first iterations of search.

The branch added to the UCT tree of the BR-search acts as a sketch as defined in the literature (Solar-Lezama 2009) because it represents a program with "holes" that are filled by the BR-search. In our example, assuming that the Ψ-value of p is somewhat large, the BR-search will be biased to explore the sketches that share the structure of p, such as "if$(?)$ then c_1" and "if$(?)$ then ?", where each question mark represents a hole that needs to be filled. Sketch learning provides a set of sketches with varied levels of detail (deeper nodes in the branch represent sketches with more information) that the BR-search explores while optimizing for Ψ.

Sketch Learning with Simulated Annealing

Like with UCT, we run SA to clone σ_o by using $C(L, p)$ as evaluation function. During search, every time we find a solution with better C-value, we also evaluate it with Ψ. Once we reach a time limit, SA returns the program p with largest Ψ-value. We also call this search the sketch-search. We then use p as the initial program of another SA search that optimizes for Ψ directly, which we also refer to as the BR-search. We reckon that the program p allows the BR-search to start in a more promising part of the program space, because p might have a structure that is similar to the structure of a program that approximates a best response to σ_{-i}.

While the branch added to the UCT tree of the BR-search can be seen as a set of sketches that are explored according to the prioritization defined by UCT, the connection between using program p to initialize the BR-search and sketches is not as clear. We see the program p as a *soft sketch*, because it provides an initial structure to the synthesizer, but it does not explicitly specify a set of holes. Since SA can change any subtree of p's AST, any subtree can be seen as a *soft hole* of p. Some subtrees are more likely to be replaced than others due to SA's acceptance function, i.e., SA prefers to change subtrees that will result in an increase in Ψ-value.

Score Functions for Behavioral Cloning

We use domain dependent functions $C(L, p)$ and described them in the empirical section. We consider score functions that use both the state and actions in the data set $L = \{(s_j, a_j)\}_{j=1}^{m}$ and functions that use only the states in L, as in recent approaches on imitation learning from observations (Torabi, Warnell, and Stone 2018).

Empirical Evaluation

The goal of our evaluation is to verify if synthesizers that learn a sketch with behavioral cloning generate stronger approximated best responses to σ_{-i} than their counterparts, that optimize directly for Ψ. All experiments were run on a single 2.4 GHz CPU with 8 GB of RAM and a time limit of 2 days.[1]

Problem Domains

We use the two-player versions of Can't Stop and MicroRTS. We chose these games because they have different features that add to a diversity of scenarios. While Can't Stop is a stochastic game, MicroRTS is a deterministic game played with real-time constraints. The branching factor of Can't Stop is small (2 or 3 actions per state), while MicroRTS has an action space that grows exponentially with the number of game components (Lelis 2020). Finally, there exist strong human-written programmatic strategies for these games that we can use as σ_{-i}.

Can't Stop The game of Can't Stop is played on a board with 11 columns, numbered from 2 to 12. The column 2 has 3 rows and the number of columns increases in size by 2 for every column until column 7, which has 13 rows. The number of rows decreases by 2, starting at column 8 until column 12, which also has 3 rows. The player who first conquers 3 columns wins the game. In each round of the game the player has 3 neutral tokens and they roll 4 six-sided dice. The player can place a neutral token in any column that is given by the combination of a pair of dice. A neutral token is then placed on the board, initially at the first row of the chosen column and later immediately above a permanent marker. The player can then decide to stop playing or to roll the dice again. If the player chooses the former, the neutral tokens are replaced by permanent tokens, thus securing that position on the board. If the player decides to roll the dice again, they are able to use the remaining neutral tokens, if there are any, or advance in columns in which they already have a neutral token placed. If the player does not have neutral tokens and the combination of dice only result in column numbers for which the player does not have a neutral token on, the player loses the neutral tokens and the other player starts their turn. A column is conquered when a player places a permanent token on the last row of a column.

Glenn and Aloi (2009) used a genetic algorithm to improve an existing programmatic strategy for Can't Stop (Keller 1986). We use Glenn and Aloi's strategy as σ_{-i} and we call it GA. GA decides to stop playing in a turn of the game whenever the sum of scores of the neutral markers exceeds a threshold. GA defines a program for computing such a score (yes-no decision) and another program to decide in which column to advance next (column decision).

We have developed a DSL for synthesizing programs for both the yes-no and column decisions. The DSL includes operators such as `map`, `sum`, `argmax`, and lambda functions. The DSL also includes a set of domain-specific functions such as a function for counting the number of rows a player has advanced in a turn. We describe the DSL in the supplementary materials. The synthesis task is to generate a program that simultaneously solves the yes-no and the column decisions while maximizing the player's utility against GA.

[1]The implementation of all algorithms used in our experiments is available at https://github.com/leandrocouto/sketch-learning.

The Ψ function for Can't Stop is the average number of victories of p against σ_{-i} on 1,000 matches. We run the sketch-search of SA for 1 hour and the BR-search for the remaining time. We run the sketch-search of UCT for 10 hours because UCT is slower than SA in exploring the space. We did not evaluate other time schedules for the synthesis of strategies with the sketch learning approaches.

MicroRTS In MicroRTS each player controls a set of units of different types. Worker units can collect resources, build structures (Barracks and Bases), and attack opponent units. Barracks and Bases can neither attack opponents units nor move, but they can train combat units and Workers, respectively. Combat units can be of type Light, Heavy, or Ranged. These units differ in how long they survive a battle, how much damage they can inflict to opponent units, and how close they need to be from opponent units to attack them. Actions are deterministic and there is no hidden information in MicroRTS. A match is played on a map and each map might require a different strategy for defeating the opponent. We use four maps of different sizes, where the names in parenthesis are the names in the MicroRTS code base:[2] 16×16 (TwoBasesBarracks), 24×24 (BasesWorkers), 32×32 (BasesWorkers), and 64×64 (BloodBath-B). The smallest and largest maps are from the 2020 MicroRTS Competition. We use the winner of the latest MicroRTS Competition, COAC, as σ_{-i} (Ontañón 2020). COAC is a programmatic strategy written by human programmers.

We have implemented a DSL similar to the one presented by Mariño et al. (2021). The DSL includes loops, conditionals, and a set of domain-specific functions that assign actions to units (e.g., build a barracks) and a set of Boolean functions. We describe the DSL in the supplementary materials.

The Ψ function for MicroRTS is the average number of victories of p against σ_{-i} in 2 matches. Each map has two starting locations, so we run 2 matches alternating the players' starting location for fairness. MicroRTS does not require an explicit time schedule for splitting the time between the sketch-search and the BR-search. This is because both Ψ and $C(L, p)$ are computed by having p play 2 matches against σ_{-i}. The transition between sketch-search and BR-search occurs naturally if we define the evaluation function of the search algorithms as the Ψ function with ties being broken according to $C(L, p)$. In the beginning of the synthesis the Ψ-value will be zero for all programs evaluated in search, but $C(L, p)$ quickly provides different values for different programs, which will guide the search toward helpful sketches.

Score Functions

We consider an action-based score function where the score of p is the fraction of actions that p chooses at states in pairs (s_j, a_j) of L that match the action in the pair, i.e., $\sum_{(s_j,a_j)\in L} \mathbb{1}[a_j = p(s_j)]/|L|$, where $\mathbb{1}$ is the indicator function. We denote SA and UCT learning sketches with this score function as Sketch-SA(A) and Sketch-UCT(A).

Can't Stop We use an observation-based score function that measures the percentage of permanent markers on the end-game state of a match that overlaps with the permanent markers obtained by a program p on the match's end-game state if p had played it. This score is computed by iterating through each state s_j of a match in L and applying the effects of actions $p(s_j)$ to an initially empty board of the game; once an end-game state s_f is reached, we compute the percentage of overlapping permanent markers between s_f and the end-game state in L. For example, if the player in the end-game state of a match in L conquered columns 2, 3, and 7, and had one marker on column 12, and program p conquered columns 2, 3, and had one marker on column 8, then the score is $(3+5)/(3+5+13+1+1) = 0.34$. Here, $(3+5)$ is the number of positions in the intersection of the end-game states and $(3+5+13+1+1)$ is the union of the positions. If p and σ_o return the same actions for all states in L, then the score is 1.0. If L has multiple matches, we return the average score across all matches. We denote SA and UCT using this score function as Sketch-SA(O) and Sketch-UCT(O).

MicroRTS We use an observation-based score function that computes a normalized absolute difference between (i) the number of units and resources the strategy p trains and collects in a match of p against σ_{-i} and (ii) the number of units and resources the strategy σ_o trains and collects in a match in the data set L. Let n_u and n'_u be the number of units of type u that p and σ_o have trained in their matches. The score related to units of type u is given by $1 - \frac{|n_u - n'_u|}{\max(n_u, n'_u)}$. For example, if the number of Ranged units the strategy p trained is 4 and if the number of Ranged units the strategy σ_o trained is 10, then the score for Ranged units is $1 - 6/10 = 0.4$. The value returned is the average scores of all types of units and resources. The score is 1.0 if both p and σ_o train the same number of units of each type and collect the same number of resources. We denote SA and UCT using this function as Sketch-SA(O) and Sketch-UCT(O).

Strategies to Clone

We use weak and strong strategies σ_o for generating the data sets L. L is composed of state-action pairs from matches in which σ_o plays the game with either itself or another strategy, which is specified below. For self-play matches, we include in L only the state-action pairs of the winner.

Can't Stop We consider 3 data sets L, each generated with a different σ_o. The first σ_o randomly chooses one of the available actions at each state of the game. The data set is composed of 3 self-play matches of this strategy, which wins approximately only 2.8% of the matches it plays against σ_{-i}. We use a data set composed of 3 self-play matches of the GA strategy and a data set composed of 3 matches a human played with GA; the human won all matches.

MicroRTS We also consider 3 data sets L for MicroRTS. The first L is composed of 2 matches (one in each starting location of the map) of Ranged Rush (RR), which is a simple programmatic strategy (Stanescu et al. 2016), against COAC. We also use a data set composed of 2 matches of A3N, a Monte Carlo tree search algorithm (Moraes et al. 2018), against COAC. A3N considers low-level actions of units (e.g., move one square to the right) while planning their

[2] https://github.com/santiontanon/microrts

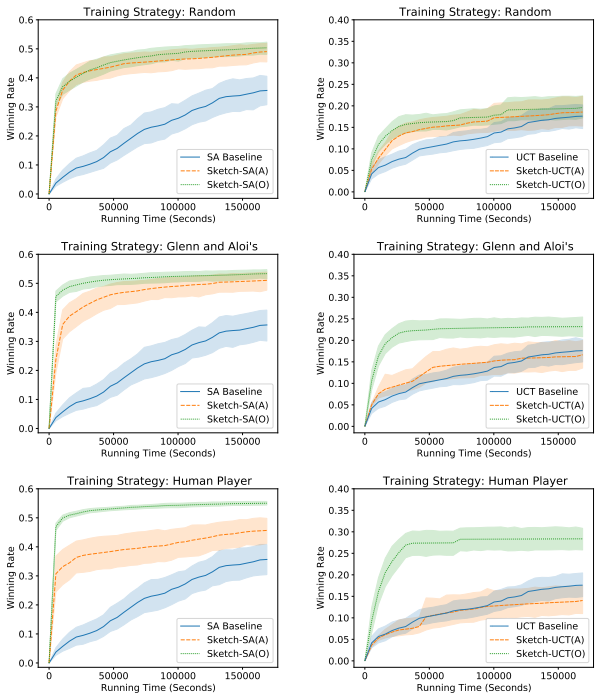

Figure 2: Winning rate of SA (left) and UCT (right) variants.

actions. We chose A3N because we reckon it would be hard for the synthesizer to clone its behavior as the strategies derived with A3N are unlikely to be in the space of strategies defined by the DSL (the DSL does not allow for a fine control of the units, as A3N does). Both RR and A3N are unable to win any matches against COAC, our σ_{-i}, in all maps evaluated. We also consider a data set composed of states from 2 self-play matches of COAC.

Empirical Results: Can't Stop

Figure 2 shows the results on Can't Stop. Each plot shows the winning rate against σ_{-i} (y-axis) of the best synthesized strategy over time (x-axis) of the methods: a search algorithm (either SA or UCT) without behavioral cloning for sketch learning (Baseline), a search algorithm that learns sketches: Sketch-SA(A), Sketch-UCT(A), Sketch-SA(O) or Sketch-UCT(O). In the plots we account for the time used in the sketch-searches. We ran 30 independent runs of each method and the lines represent the average results while the shaded areas the standard deviation of the runs.

The learning-sketch methods are much faster than their baselines. In most of the cases, Sketch-SA and Sketch-UCT achieve winning rates that their baseline counterparts did not achieve within the time limit. The results also show that the SA methods perform better than their UCT counterparts. Only Sketch-SA synthesized strategies that defeat σ_{-i} in more than 50% of the matches. In particular, Sketch-SA(O) is the most effective method for approximating a best response for σ_{-i}. We conjecture SA synthesizes stronger strategies than UCT because it explores the space more quickly than UCT. The time complexity of UCT's selection step is quadratic on the program's length. This is because, in each selection step, the search traverses all production rules of the current incomplete program for each production rule applied to it. By contrast, SA can synthesize a large number of instructions with a single neighborhood operation.

Although Sketch-SA(O) performs better by cloning the behavior of the human player, the method performs surprisingly well when it learns sketches by cloning the behavior of a random strategy. One of the key aspects for playing Can't Stop is to decide when to stop playing so that the neutral markers become permanent markers. The program must have a specific structure for computing the score that leads to a stop action, similar to `sum(map(`$\lambda.f$`, neutrals))`. Here, `neutrals` is a list with the neutral markers and f is a score function for individual markers. The `sum` and `map` operators return the sum of the scores of all markers. While the structure of this program is not trivial, the random strategy has a 50% chance of choosing the stop action, and its effects are reflected on the states in L (i.e., neutral markers become permanent markers). The synthesizers discover sketches like the program above because such programs place permanent markers on the board, as the random strategy does. The BR-search modifies the sketch to maximize the player's utility, but most of program's structure is maintained.

The sketch-based methods perform worse with the action-based function. This is because the observation-based function captures the effects of even rare actions. A good player of Can't Stop chooses to continue playing in most states, but at crucial states they choose to stop. A player that never stops has a high action-based score because the stop action is rare. As a result, the sketch-based methods often fail to learn the program structure needed to correctly decide when to stop.

Empirical Results: MicroRTS

Figure 3 shows the results of the SA variants on MicroRTS; we omit the UCT plots for space. The results of the UCT variants on MicroRTS are similar to those on Can't Stop. The UCT variants perform worse than their SA counterparts. In particular, no UCT method, including the baseline UCT, is able to synthesize a strategy that defeats σ_{-i} on the larger 32×32 and 64×64 maps. Moreover, the approaches that learn sketches perform better than their counterparts on the smaller 16×16 and 24×24 maps. We present the UCT plots in the supplementary materials. The plots for maps of size 16×16 and 24×24 in Figure 3 show a reduced running time (approximately 6 hours for the former and 24 hours for the latter) so we can better visualize the curves. Each line represents the average winning rate and the shaded areas the standard deviation of 10 independent runs of each method.

Like in Can't Stop, the sketch-based methods are superior to the baseline, with Sketch-SA(O) achieving winning rates near 1.0 even when learning sketches from A3N, which is a strategy unable to defeat σ_{-i}. There is also a gap between the action and the observation-based functions and the gap seems to increase with the map size. Sketch-SA(A) did not synthesize strategies that defeated σ_{-i} for L generated with A3N and COAC on the 64×64 map. The explanation for the poor performance of Sketch-SA(A) is similar to that on Can't Stop: some actions are rare but play an important role

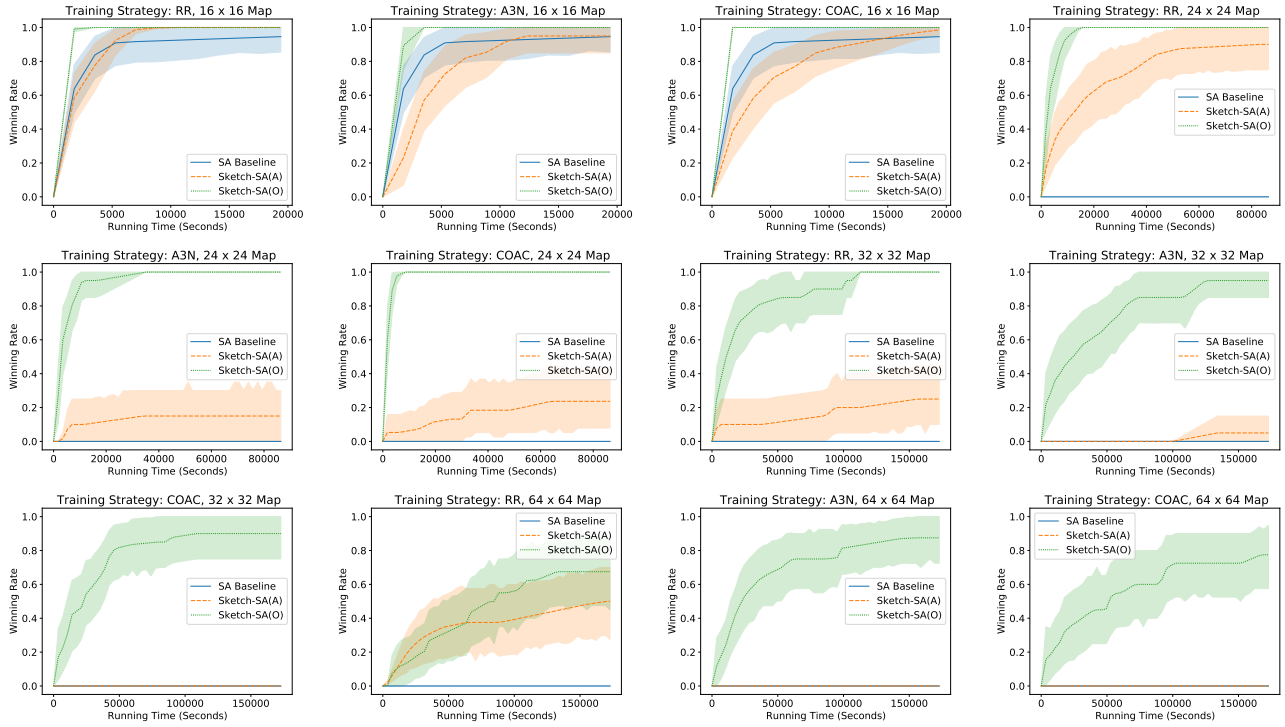

Figure 3: Winning rate of strategies SA variants synthesize; the maps increase in size from left to right and top to bottom.

```
1  def Sketch-SA-O-24x24(state s):
2      for u in s:
3          if not u.isWorker():
4              u.moveToUnit(Ally,
                   LessHealthy)
5          u.train(Ranged)
6          for u in s:
7              u.attackIfInRange()
8          u.build(Barracks)
9      for u in s:
10         u.harvest(4)
11         u.attack(LessHealthy)
```

Figure 4: Sketch-SA(O)'s strategy for the 24×24 map.

in the game (e.g., one can train Ranged units after building a barracks, which might happen only once in a match).

Sample of Programmatic Strategy

Figure 4 shows a strategy Sketch-SA(O) synthesized for the 24×24 map. We lightly edited the strategy for readability. This strategy achieves the winning rate of 1.0 against COAC. The strategy receives a state s and assigns an action to each unit in s; if a unit is not assigned an action then it does not perform an action in the next round of the game. Once the strategy assigns an action to a unit u, the action cannot be replaced by another action. For example, the strategy does not change the action assigned to units in line 4, even if we try to assign them a different action later in the program. This strategy trains Ranged units (line 5) once a barracks is built (line 8); a single barracks is built because all resources are spent training Ranged units once the barracks is available. The Ranged units cluster together (line 4) and attack enemy units within their range of attack (line 7). If there are no enemy units within their range, they attack the enemy's units that are close to being removed from the game (line 11). The strategy assigns 4 Workers to collect resources (line 10). This strategy is representative of the strategies our methods synthesize for both domains. We present other interpretable strategies in the supplementary materials.

Conclusions

In this paper we showed that behavioral cloning can be used to learn effective sketches for speeding up the synthesis of programmatic strategies. We presented Sketch-UCT and Sketch-SA, two synthesizers based on UCT and SA that learn a sketch for a program encoding an approximated best response to a target strategy by cloning the behavior of an existing strategy. The synthesizers use the sketch as a starting point in the search for an approximated best response. Experimental results on Can't Stop and MicroRTS showed that Sketch-SA can synthesize strategies able to defeat programmatic strategies written by human programmers in all settings tested, even when learning sketches from weak strategies. In particular, Sketch-SA synthesized strategies that defeated the winner of the latest MicroRTS competition on all maps used in our experiments, while baseline synthesizers failed to generate good strategies in these settings.

Acknowledgements

This research was partially supported by FAPEMIG, CAPES, and Canada's CIFAR AI Chairs program. The research was carried out using computational resources from Compute Canada. We thank the anonymous reviewers for their feedback. We are currently working on the reviewers' feedback so that they can be incorporated in the camera ready version of this paper.

References

Alur, R.; Bodík, R.; Juniwal, G.; Martin, M. M. K.; Raghothaman, M.; Seshia, S. A.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*, 1–8. IEEE.

Bain, M.; and Sammut, C. 1996. A Framework for Behavioural Cloning. In *Machine Intelligence 15*, 103–129. Oxford University Press.

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

Benbassat, A.; and Sipper, M. 2011. Evolving board-game players with genetic programming. In *Genetic and Evolutionary Computation Conference*, 739–742.

Butler, E.; Torlak, E.; and Popović, Z. 2017. Synthesizing interpretable strategies for solving puzzle games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 1–10.

Canaan, R.; Shen, H.; Torrado, R.; Togelius, J.; Nealen, A.; and Menzel, S. 2018. Evolving agents for the hanabi 2018 cig competition. In *2018 IEEE Conference on Computational Intelligence and Games*, 1–8. IEEE.

Cazenave, T. 2013. Monte-Carlo Expression Discovery. *International Journal on Artificial Intelligence Tools* 22(01): 1250035. doi:10.1142/S0218213012500352.

De Freitas, J. M.; de Souza, F. R.; and Bernardino, H. S. 2018. Evolving Controllers for Mario AI Using Grammar-based Genetic Programming. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, 1–8. IEEE.

Glenn, J. R.; and Aloi, C. J. 2009. A Generalized Heuristic for Can't Stop. In *FLAIRS Conference*.

Husien, I.; and Schewe, S. 2016. Program Generation Using Simulated Annealing and Model Checking. In De Nicola, R.; and Kühn, E., eds., *Software Engineering and Formal Methods*, 155–171. Springer International Publishing. ISBN 978-3-319-41591-8.

Keller, M. 1986. Can't Stop? Try the Rule of 28. World Game Review 6.

Kirkpatrick, S.; Gelatt, C. D.; and Vecchi, M. P. 1983. Optimization by Simulated Annealing. *Science* 220(4598): 671–680. doi:10.1126/science.220.4598.671.

Kocsis, L.; and Szepesvári, C. 2006. Bandit based Monte-Carlo Planning. In *European Conference on Machine Learning*, 282–293. Springer.

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. 2020. Planning Algorithms for Zero-Sum Games with Exponential Action Spaces: A Unifying Perspective. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 4892–4898. International Joint Conferences on Artificial Intelligence Organization. Survey track.

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.

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.

Nye, M.; Hewitt, L.; Tenenbaum, J.; and Solar-Lezama, A. 2019. Learning to Infer Program Sketches. In Chaudhuri, K.; and Salakhutdinov, R., eds., *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, 4861–4870. PMLR.

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

Ross, S.; Gordon, G.; and Bagnell, D. 2011. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, 627–635. PMLR.

Schaal, S. 1999. Is imitation learning the route to humanoid robots? *Trends in Cognitive Sciences* 3: 233–242.

Solar-Lezama, A. 2009. The sketching approach to program synthesis. In *Asian Symposium on Programming Languages and Systems*, 4–13. Springer.

Stanescu, M.; Barriga, N. A.; Hess, A.; and Buro, M. 2016. Evaluating real-time strategy game states using convolutional neural networks. In *Proceedings IEEE Conference on Computational Intelligence and Games*, 1–7. IEEE.

Torabi, F.; Warnell, G.; and Stone, P. 2018. Behavioral Cloning from Observation. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

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.