LEANDRO COUTO MEDEIROS

LEARNING SKETCHES FOR PROGRAMMATIC STRATEGIES

Dissertation submitted to the Computer Science Graduate Program of Universidade Federal de Viçosa in partial fulfillment of the requirements for the degree of *Magister Scientiae*.

Advisor: Levi Henrique Santana de Lelis

VIÇOSA - MINAS GERAIS
2021

Ficha catalográfica elaborada pela Biblioteca Central da Universidade Federal de Viçosa - Campus Viçosa

T

M488L
2021

Medeiros, Leandro Couto, 1993-
Learning sketches for programmatic strategies / Leandro Couto Medeiros. – Viçosa, MG, 2021.
1 dissertação eletrônica (51 f.): il. (algumas color.).

Inclui apêndice.
Orientador: Levi Henrique Santana de Lelis.
Dissertação (mestrado) - Universidade Federal de Viçosa, Departamento de Informática, 2021.
Referências bibliográficas: f.43-45.
DOI: https://doi.org/10.47328/ufvbbt.2022.076
Modo de acesso: World Wide Web.

1. Inteligência artificial. 2. Aprendizado do computador. 3. Jogos. I. Lelis, Levi Henrique Santana de, 1984-. II. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDD 22. ed. 006.3

Bibliotecário(a) responsável: Renata de Fátima Alves CRB6/2578

LEANDRO COUTO MEDEIROS

LEARNING SKETCHES FOR PROGRAMMATIC STRATEGIES

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 26 de novembro de 2021.

Assentimento:

Leandro Couto Medeiros

Leandro Couto Medeiros
Autor

[illegible]

Levi Henrique Santana de Lelis
Orientador

Acknowledgements

I would like to thank my advisor, Dr. Levi Lelis, for agreeing to mentor me and for the guidance throughout this work. Your advices, teachings and dedication were crucial to me to become the researcher and the person I am today. Thank you!

To all the professors I have met throughout my academic journey, all of you contributed for me to be where I am now.

To my long time friends Igor and Fábio, I appreciate all the support. You two have consistently inspired me to always move forward.

To the friends I made at DPI. I'll always remember the banter in the labs, the late night group studies before a test and the occasional jogging groups "na reta". In special, I would like to thank David, Rubens and Julian for all the important insights and help in this study.

To my parents Vanilda and Sérgio and my brothers Túlio and Artur, your continuous support was fundamental to me. Thank you for always believing in me.

This study was financed in part by the Fundação de Amparo à Pesquisa de Minas Gerais (FAPEMIG) and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES).

Abstract

MEDEIROS, Leandro Couto, M.Sc., Universidade Federal de Viçosa, November, 2021. **Learning Sketches for Programmatic Strategies**. Adviser: Levi Henrique Santana de Lelis.

Program synthesis has been a major focus of research in recent years due to its innate capability of generating interpretable programs. By contrast, neural network models are implemented as opaque models and are thus hard to interpret. Neural networks are easier to train because gradient information is available while program synthesis tasks are not differentiable, making the optimization task challenging. In this work we show that behavioral cloning can be used to learn effective sketches of programmatic strategies, facilitating the optimization task. 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 synthesizers based on simulated annealing and with synthesizers based on UCT. 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.

Keywords: Artificial Intelligence. Program Synthesis. Search. Games.

Resumo

MEDEIROS, Leandro Couto, M.Sc., Universidade Federal de Viçosa, novembro de 2021. **Aprendendo rascunhos para estratégias programáticas**. Orientador: Levi Henrique Santana de Lelis.

Síntese de programas tem sido um grande foco de pesquisa nos últimos anos devido à sua inata capacidade de gerar programas interpretáveis. Em contraste com modelos de redes neurais, que são implementados como modelos opacos e portanto são difíceis de interpretar. Redes neurais são mais fáceis de treinar devido à informação do gradiente estar disponível, enquanto que tarefas de síntese de programas não são diferenciáveis, tornando a tarefa de otimização desafiante. Nesta dissertação é mostrado que a clonagem comportamental pode ser usada para aprender rascunhos de estratégias programáticas, facilitando a tarefa de otimização. Foi observado que até rascunhos aprendidos ao clonar o comportamento de jogadores fracos podem ajudar na síntese de estratégias programáticas. Isto ocorre porque até mesmo jogadores fracos conseguem prover informações úteis, e.g., que um jogador deve escolher uma ação em sua rodada do jogo. Caso clonagem comportamental não seja usada, o sintetizador precisa aprender até mesmo as informações mais básicas jogando o jogo, o que pode ser computacionalmente custoso. É empiricamente demonstrado as vantagens da abordagem de aprendizado por rascunhos com sintetizadores baseados na têmpera simulada e com sintetizadores baseados no algoritmo UCT. Os sintetizadores foram avaliados nos jogos *Can't Stop* e *MicroRTS*. Os sintetizadores baseados em rascunhos são capazes de aprender estratégias programáticas mais fortes do que as abordagens originais. Os sintetizadores geraram estratégias de *Can't Stop* que derrotaram uma estratégia programática tradicional do jogo. Também foram sintetizadas estratégias que derrotaram o método com a melhor performance da última competição de *MicroRTS*.

Palavras-chave: Inteligência Artificial. Síntese de Programas. Busca. Jogos.

List of figures

List of tables

Table of Contents

1 Introduction

1.1 The Problem and its Importance

GULWANI et al. (2017) define program synthesis as the task to find programs that satisfy users intent expressed in the form of a specification. Program synthesis tasks can be used in a myriad of distinct problems. SINGH (2014) developed a system where the synthesizer adopted a programming teacher role, assisting students to develop programs by providing feedback to its user. AHMED; GULWANI; KARKARE (2013)'s application is able to solve programming problems and generate similar problems. CHEUNG; SOLAR-LEZAMA; MADDEN (2012) combined program synthesis with machine learning algorithms for a recommendation system.

With the advent of deep neural networks (DNN), one could argue that DNNs are the master key for computational problems given the large number of hard problems it was able to solve. However, DNNs as they are currently designed, have one major disadvantage which is their lack of interpretability. Because they are implemented as an opaque model, it is often not possible for a human to fully understand how the solution actually works. The problem of lacking interpretability can be solved with program synthesis. A solution a synthesizer writes can be easier for a human to understand how it works.

Search algorithms are used to navigate through the program space allowed by a domain-specific language (DSL) (DEURSEN; KLINT; VISSER, 2000). In cases where the problem is hard to solve or the size of the program space induced by the grammar is huge, search algorithms presumably will struggle to find a solution that has a good performance in a timely manner.

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 learning process. In order to overcome this hurdle, we study the use of behavioral cloning (BAIN; SAMMUT, 1996), a field under the umbrella of imitation learning, as such a function to guide the search algorithm to learn a sketch. A sketch is a "partial program that encodes the structure of a solution while leaving its low-level details unspecified" (SOLAR-LEZAMA, 2009). By learning a sketch, we are simplifying the optimization task by allowing the search algorithm to look for programs with a predetermined program structure, hence reducing the original substantial search space.

In this work, we show that using behavioral cloning to learn sketches can be used to speed up the synthesis of programmatic strategies. We investigate the use of this sketch learning approach with synthesizers employing Simulated Annealing (SA) (KIRKPATRICK; GELATT; VECCHI, 1983) and UCT (KOCSIS; 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 evaluate our sketch-based SA and UCT methods by synthesizing approximated best responses to a known programmatic strategy for Can't Stop (GLENN; 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.

1.2 Hypothesis

The use of behavioral cloning for learning program sketches can speed up the synthesis process of programs encoding strong strategies for playing zero-sum games.

2 Background

2.1 Behavioral Cloning

Given an oracle to generate a list of state-action pairs, a Behavioral Cloning (BC) agent will learn a policy based only on the data generated by an oracle. If $L = \{(s_i, a_i)\}_{i=1}^{m}$ is the data set of state-actions pairs produced by the oracle, a BC agent tries to learn a policy π that maps each s_i in L to the action a_i the oracle takes at s_i.

Behavioral Cloning has been used successfully in distinct applications. SAMMUT et al. (1992) experiment logged human data using a flight simulation program as the oracle and used a decision tree to learn a policy based on the oracle data. ROSS; GORDON; BAGNELL (2011) observed a weakness on the classical BC setting that the learned policy struggles when the agent faces a state that it has never seen before during the learning stage. They proposed an algorithm called DAgger, that iteratively adds unseen states to the original dataset. The oracle labels all newly collected states s_j and the imitation learning step is repeated.

TORABI; WARNELL; STONE (2018a) proposes a variation of the BC setting called Behavioral Cloning from Observation (BCO), in which the agent learns a policy only from observations, i.e., the data set used is $L = \{(s_i)\}_{i=1}^{m}$. They then proposed a variation called BCO(α) which is a two-step algorithm: first the proposed BCO implementation is executed to learn a policy π. In the second step, the policy π is executed in the environment to collect a new dataset $L_\pi = \{(s_j, a_j)\}_{j=1}^{n}$ and the learned policy π is updated using the classical BC implementation using the new dataset L_π, where newly collected pairs action-observation are imitated.

2.2 Simulated Annealing

Simulated Annealing (SA) is a local search algorithm initially proposed by KIRKPATRICK; GELATT; VECCHI (1983) that searches for a global minimum/maximum given a evaluation function Ψ. This algorithm overcomes a big disadvantage of the Hill Climbing algorithm: it might escape local minima/maxima in case Ψ is non-convex.

SA uses a temperature scheduling approach to control the greediness of the search. Instead of always choosing a neighbor state that minimizes/maximizes Ψ as it is done in the Hill Climbing algorithm, SA uses an acceptance function that uses the current temperature to decide if the neighbor sample will be accepted.

The SA pseudocode is shown below:

```
1 def simulated_annealing():
2     $s$ = random_sample()
3     for i in range(MAX_ITE):
4         current_temperature = update_temperature(i)
5         $s'$ = choose_neighbor($s$)
6         if acceptance_function($s$, $s'$, current_temperature, $\Psi$):
7             $s$ = $s'$
8     return $s$
```

In line 2, SA starts with a random sample s and it then loops for MAX_ITE iterations (line 3). SA updates the temperature according to a temperature scheduling function (line 4) and it generates a neighbor s' of s (line 5). At line 6, SA accepts or rejects s' according to an acceptance function. If it accepts, then s' is assigned to s (line 7) and the process is repeated. If it rejects, SA repeats the procedure by generating another neighbor of s.

The way the temperature schedule is implemented is generally problem-specific and it is widely studied, see HAJEK (1988) and NOURANI; ANDRESEN (1998). With regards to the acceptance function, the expression below developed by KIRKPATRICK; GELATT; VECCHI (1983) is still often used in the literature:

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

Here, T_i is the temperature at iteration i, and Ψ is an evaluation function. If $\Psi(s') \geq \Psi(s)$, then SA accepts s' with probability 1.0. Otherwise, the probability of acceptance depends on T_i and β. β is an input parameter that allows the adjustment on how greedy SA is; larger values of β result in a greedier search by more often rejecting samples with small Ψ-values. Larger values of T_i make the search less greedy since large T-values increase the chances of accepting s'.

2.3 Monte-Carlo Tree Search

The Monte-Carlo Tree Search (MCTS) developed by COULOM (2006) combines Monte-Carlo evaluations (where it plays several simulations of a Markov Decision Process (MDP) and calculates the average outcome returned by the MDP) and tree search. Given enough simulations, MCTS will store information about the utility value of an action applied to a state. A low utility value means that applying this action in that state is probable to yield a low score by an evaluation function Ψ.

The MCTS algorithm can be separated in four main routines:

1. **Selection:** The selection step starts at the root of the tree and it chooses the i-th child that maximizes a selectivity formula. COULOM (2006) indicates that an ideal selectivity formula should allocate more iterations, and therefore be searched deeper, for promising nodes. If the child selected is a leaf node, it then moves to the expansion routine. If not, it calls the selection routine again until a leaf node is reached.

2. **Expansion:** Given the leaf node retrieved in the selection stage, it then expands a random child (in order to reduce bias) by adding this node to the tree.

3. **Simulation:** With the node just expanded from the expansion stage, a simulation is applied from that node until the game ends. The simulation follows a policy applied to this node until a terminal node is reached and then a function Ψ is used to evaluate this state, returning the utility value Ψ of that state.

4. **Backpropagation:** The backpropagation step updates the X_i-values of all nodes visited in the selection step with the Ψ-value from the simulation step.

Figure 1 shows the overall iterative pipeline of the MCTS algorithm. Because it is an iterative process, the more time the algorithm is given, the more accurate the estimation will be regarding which action to choose from a state. After a certain time limit imposed by the user or any other stopping criteria, the algorithm will return the action relating to which child node had the most number of visits during the iterative process.

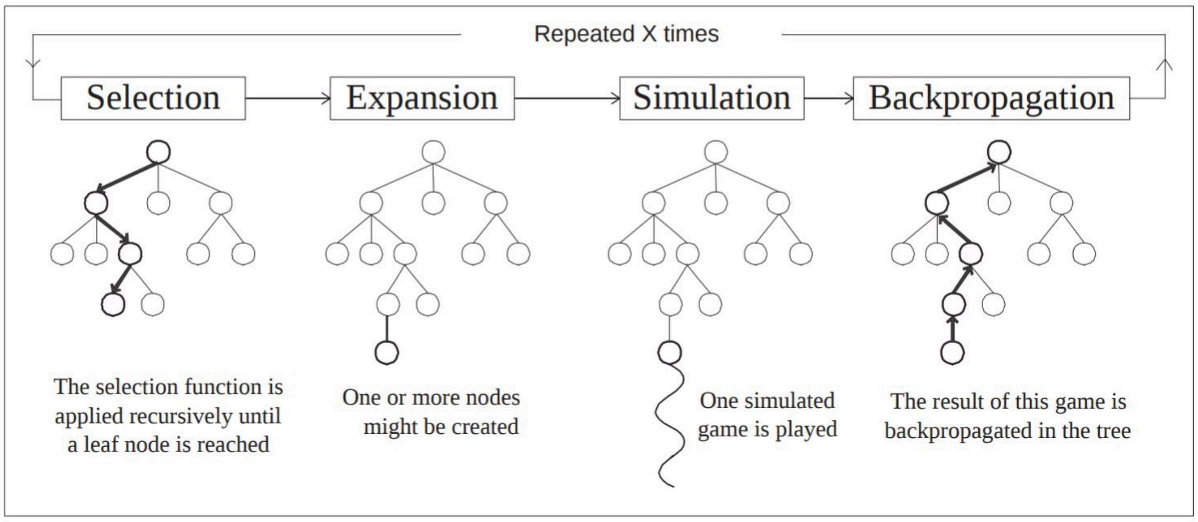

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes might be created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 1 – Overview of the Monte-Carlo Tree Search Algorithm

Source: (CHASLOT et al., 2008)

KOCSIS; SZEPESVÁRI (2006) developed a MCTS-based algorithm called *Upper Confidence Bound applied to Trees* (UCT). The utility of a state s based on this heuristic is called UCB1 (Upper Confidence Bounds) by AUER; CESA-BIANCHI; FISCHER (2002) and it is calculated as follows:

$$\text{UCB1}(s) = \bar{X}_i + C\sqrt{\frac{\ln(N(s))}{n_i(s)}}$$

Here, $\bar{X}_j$ is the average Ψ-value of the i-th child of s, N is the number of times node s was visited in previous selection steps, n_i is the number of times s was visited and the i-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 UCB1 formula is used in the selection stage as the selectivity formula in the UCT algorithm.

3 Related Work

This 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; SOLAR-LEZAMA (2018) present an algorithm called VIPER that uses a variant of the imitation learning algorithm DAgger (ROSS; GORDON; BAGNELL, 2011) to distill a high-performing neural policy into an interpretable decision tree. VIPER uses a deep neural network as an oracle to generate labeled data for DAgger-like queries in order to train the aforementioned 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; 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 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; 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; POPOVIĆ, 2017; FREITAS; SOUZA; 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).

4 Problem Definition

Let G be a two-player sequential 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 DSL. Let D be a DSL and $\llbracket D \rrbracket$ be the set of programs written in D. The best response for a strategy σ_{-i} in $\llbracket D \rrbracket$ is a strategy that maximizes player i's utility against σ_{-i}, i.e., $\max_{\sigma_i \in \llbracket D \rrbracket} U(s_{init}, \sigma_i, \sigma_{-i})$. The computation of a best response for a fixed strategy is a basic operation in game theory methods such as iterated best response for approximating a Nash equilibrium profile (LANCTOT et al., 2017).

In this work 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 $\llbracket D \rrbracket$ and returns an approximated best response σ_i to σ_{-i}. We also consider the setting in which a data set generated by an oracle with state-action pairs $L = \{(s_j, a_j)\}_{j=1}^{m}$ with the actions a_j the oracle takes at states s_j is available.

4.1 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; SCHEWE, 2016; CAZENAVE, 2013), this is the first time these approaches are applied to synthesize programmatic strategies, so we describe them in detail.

4.1.1 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 2 shows a DSL where $V = \{I, C, B\}$, $\Sigma = \{c_1, c_2, b_1, b_2 \text{ if, then}\}$, R are the relations (e.g., $C \rightarrow c_1$).

The DSL allows programs with a single command (c_1 or c_2) and programs with

$$\begin{aligned} I &\to C \mid \text{if}(B) \text{ then } C \\ C &\to c_1 \mid c_2 \\ B &\to b_1 \mid b_2 \end{aligned}$$

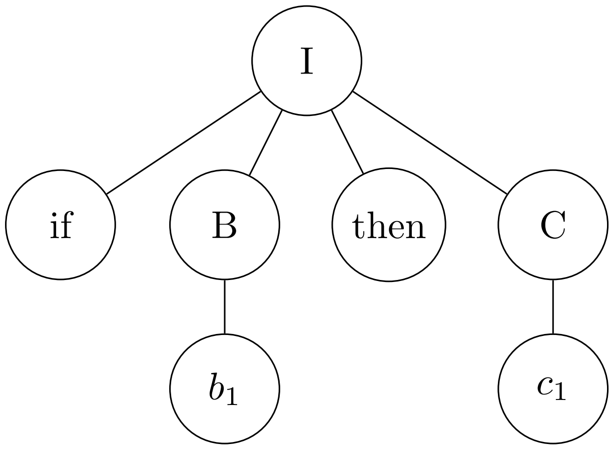


Figure 2 – DSL (left) and AST for "if b_1 then c_1" (right).

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 2 shows an example of an AST, where leaves are terminal symbols and internal nodes are non-terminals.

4.1.2 Simulated Annealing for Synthesis of Strategies

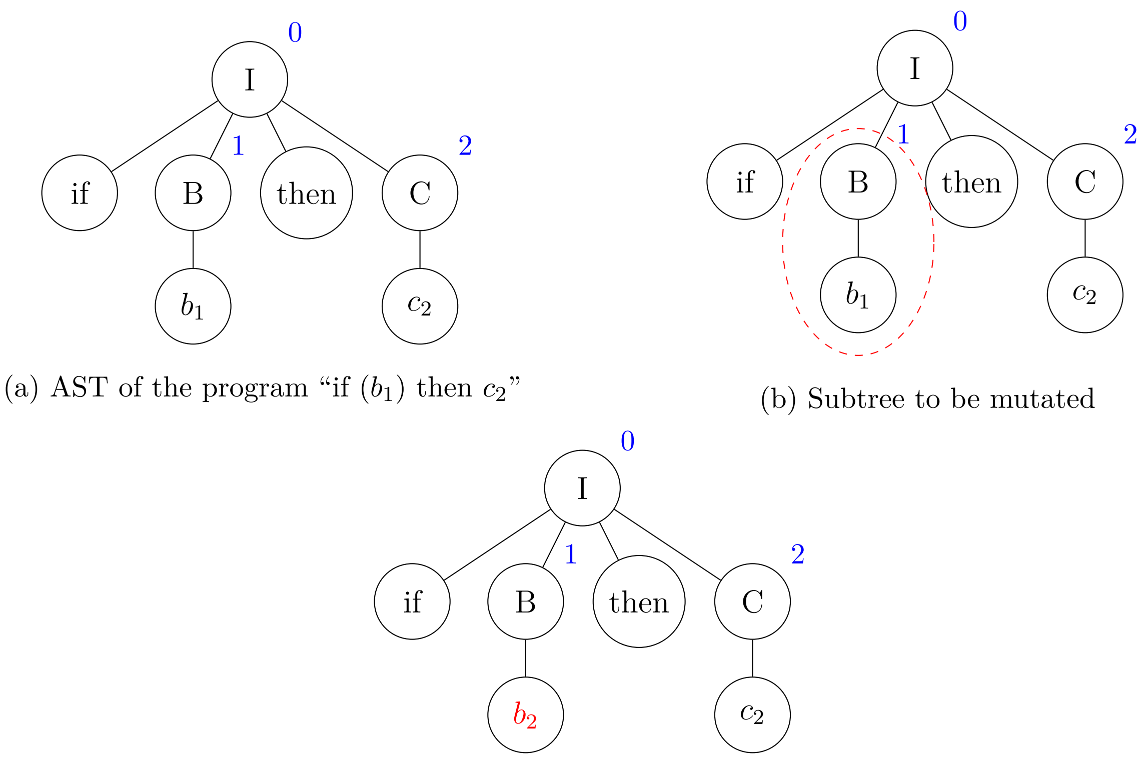
For synthesis of strategies, we use SA to approximate a programmatic best response to a target strategy σ_{-i}, i.e., SA approximates a solution to $\arg\max_{\sigma_i \in [\![D]\!]} U(s_{init}, \sigma_i, \sigma_{-i})$.

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: $I \to \text{if}(B)$ then C else C; $B \to b_1$; $C \to c_1$.

Once the initial program p is defined, SA generates a neighbor p' of p by changing a subtree in p's AST. For example, consider the AST representation of the program "if (b_1) then c_2" in Figure 3, (a). We randomly choose a non-terminal node in the AST, represented by the indexed nodes {0, 1, 2}. If the B node is chosen for mutation as shown in (b), we replace the subtree rooted at B with a subtree that is generated with the same procedure used to generate the initial program. For example, the terminal node b_1 node can become b_2 as seen in (c). SA decides if it accepts or rejects "if (b_2) then c_2". If it accepts, then this program is assigned to p and the process is repeated. If it rejects, SA repeats the procedure by generating another neighbor of "if (b_1) then c_2". The probability in which SA accepts the mutated program 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



(a) AST of the program "if (b_1) then c_2"

(b) Subtree to be mutated

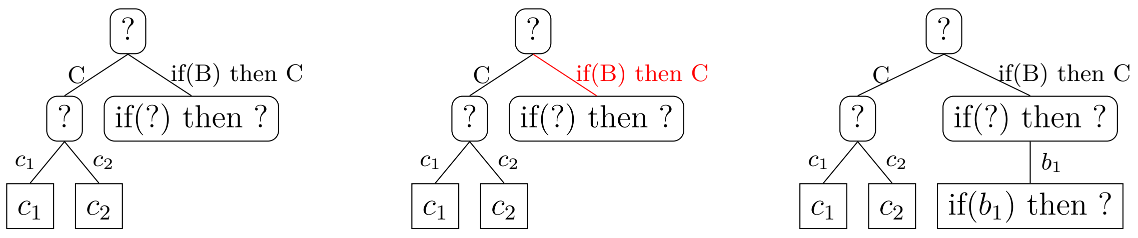
(c) Mutated program "if (b_2) then c_2"

Figure 3 – Example of SA's mutation step

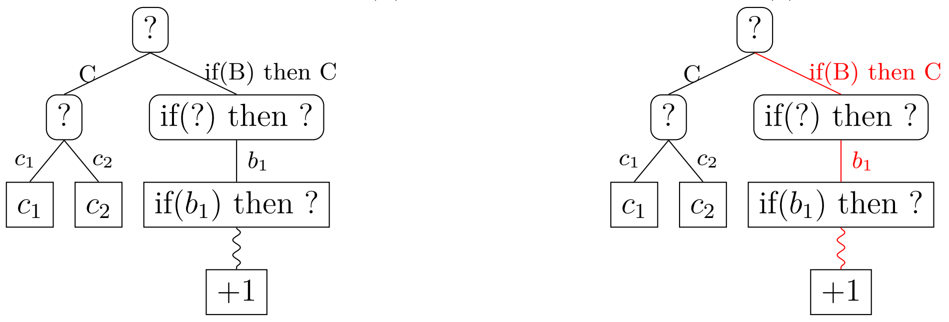
synthesis tasks, $\Psi(p)$ counts the number of input examples that p correctly maps to the desired output (ALUR et al., 2013). In the context of games, $\Psi(p)$ returns the utility of p against the opponent σ_{-i}. The initial temperature, T_1, is an input parameter and T_j is computed according to 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.

4.1.3 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. Figure 4 shows an example of a UCT iteration using the DSL shown in Figure 2. 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, consider the UCT tree as shown in (a). The selection step will choose between the two children of the root according to the UCB1



(a) Current UCT tree (b) Selection step (c) Expansion step



(d) Simulation step (e) Backpropagation step

Figure 4 – Example of an iteration of the UCT algorithm

value of each child. If the sketch "if(?) then ?" has a higher UCB1 value, it is then chosen to be expanded in the expansion step, as shown in (b). In (c), the expanded node is generated by applying a production rule to the leftmost non-terminal symbol of the program "if(?) then ?", in this case, B. After expansion, in (c) the simulation step is demonstrated, where the current sketch "if(b_1) then ?" is randomly filled in until it is a complete program, for example, "if(b_1) then c_2". This program will be evaluated according to an evaluation function and return a value, for example +1, as shown in (d). Finally, the value +1 will be backpropagated all the way up to the root, as shown in (e).

The four steps of UCT are repeated multiple times and it 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.

4.2 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.

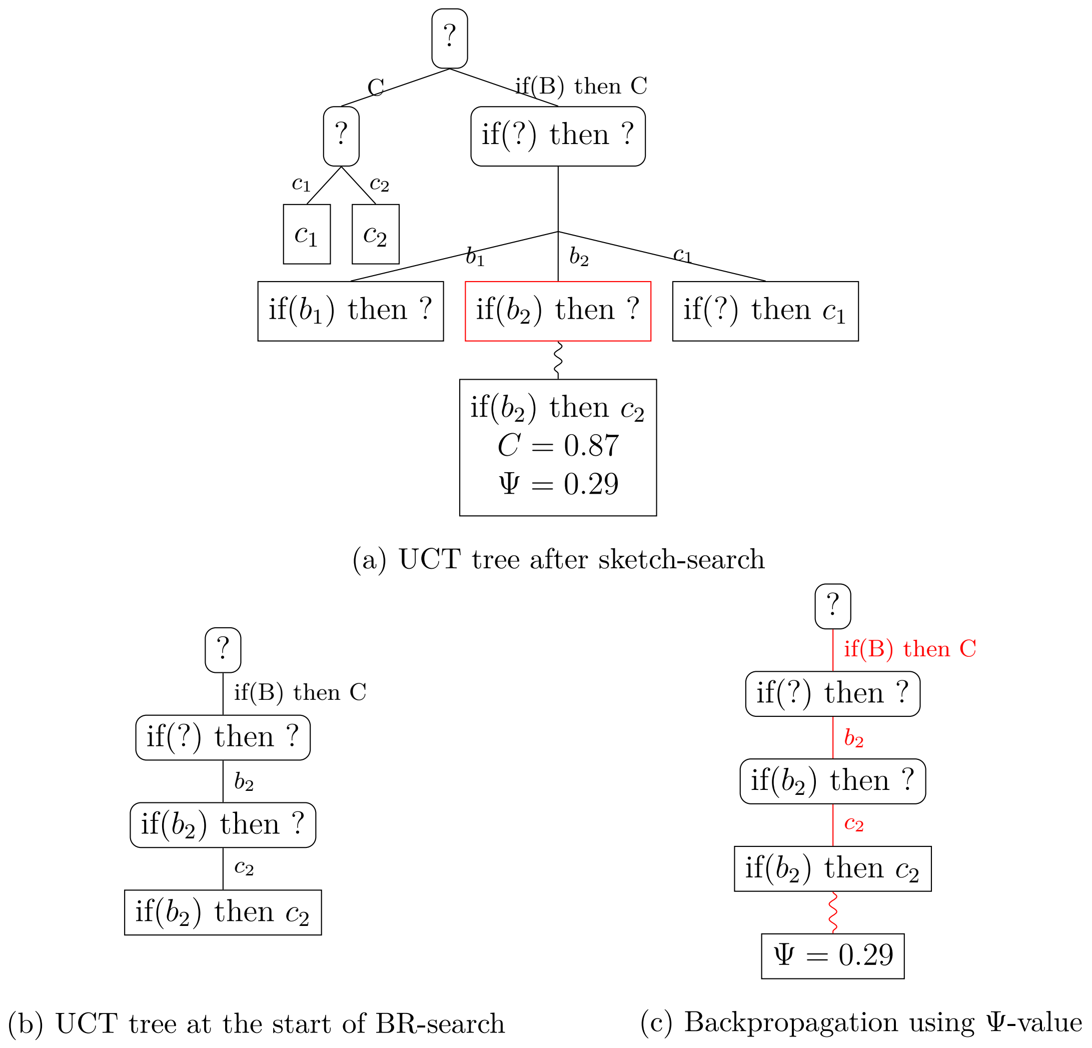
SA and UCT can be used to clone the behavior of σ_o by replacing Ψ with an evaluation function $C(L, p)$ that receives the data set L and a program p and returns a score of how well p clones σ_o. Note, however, that cloning the behavior of σ_o can result in weak strategies. This is because σ_o might not be represented in $[\![D]\!]$. Or the data set L is limited and one needs to perform DAgger-like queries (ROSS; GORDON; 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).

4.2.1 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, consider the example shown in Figure 5 where "if(b_2) then c_2" is the program with the largest Ψ-value from the sketch-search, where it achieved a Ψ-value of 0.29 and a C-value of 0.87. The UCT tree of the BR-search is initialized with the branch with nodes representing the programs: "I", "if(B) then C", "if(b_2) then C" and "if(b_2) then c_2" as shown in (b). We then perform a backpropagation step on the added branch with Ψ-value of 0.29, which was computed in the sketch-search.



(a) UCT tree after sketch-search

(b) UCT tree at the start of BR-search

(c) Backpropagation using Ψ-value

Figure 5 – Transition from sketch-search to BR-search in UCT

By adding the branch leading to "if(b_2) then c_2" to the tree of the BR-search and applying a backpropagation step with the program's Ψ-value we are biasing it to explore programs that share the structure of "if(b_2) then c_2". 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.

4.2.2 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 SA 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.

4.2.3 Score Functions for Behavioral Cloning

We use domain dependent functions $C(L, p)$ and describe 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; STONE, 2018b).

5 Results and Discussion

The primary goal of our empirical 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. We describe the domains of Can't Stop and MicroRTS, the DSLs, the score functions, and the target strategies σ_{-i}. All experiments were run on a single 2.4 GHz CPU with a limit of 8 GB of RAM and a time limit of 2 days.

5.1 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}.

5.1.1 Can't Stop

The game of Can't Stop is played with 2-4 players 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. Figure 6 presents an example of a Can't Stop board.

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 combinations 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.

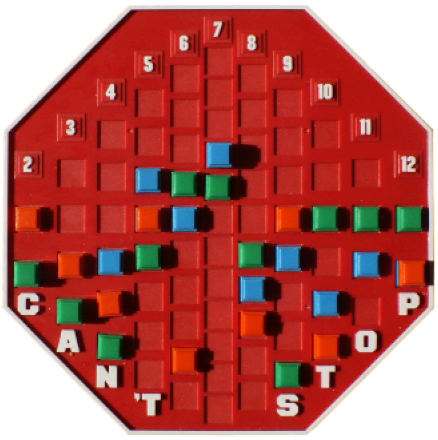


Figure 6 – Can't Stop board game. Different collors represent different players.

Source: <https://en.wikipedia.org/w/index.php?curid=4276555> (Accessed on 10/2021)

GLENN; ALOI (2009) used a genetic algorithm to improve an existing programmatic strategy for Can't Stop (KELLER, 1986). We use GLENN; ALOI (2009)'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. In particular, we developed Cant's Stop's DSL to allow the synthesizer to be able to synthesize GA, with extra functions that are not present in Glenn and Aloi's implementation. We describe the DSL in details in the appendix. 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 which is the state-of-the-art strategy for Can't Stop.

The Ψ function for Can't Stop is the average number of victories of p against σ_{-i} over 1,000 matches. We need a high number of matches against σ_{-i} due to the stochasticity of Can't Stop. 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.

5.1.2 MicroRTS

MicroRTS (ONTAÑÓN, 2013) is an implementation of a real-time strategy game, played between two players. 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.

Figure 7 shows an example of a MicroRTS game state: gray circle units represent the Worker units, cyan circle units are the Ranged units and yellow circle units represent Heavy units. Light green squares represent the resources to be collected by the Workers, white squares are the Bases and gray squares are the Barracks.

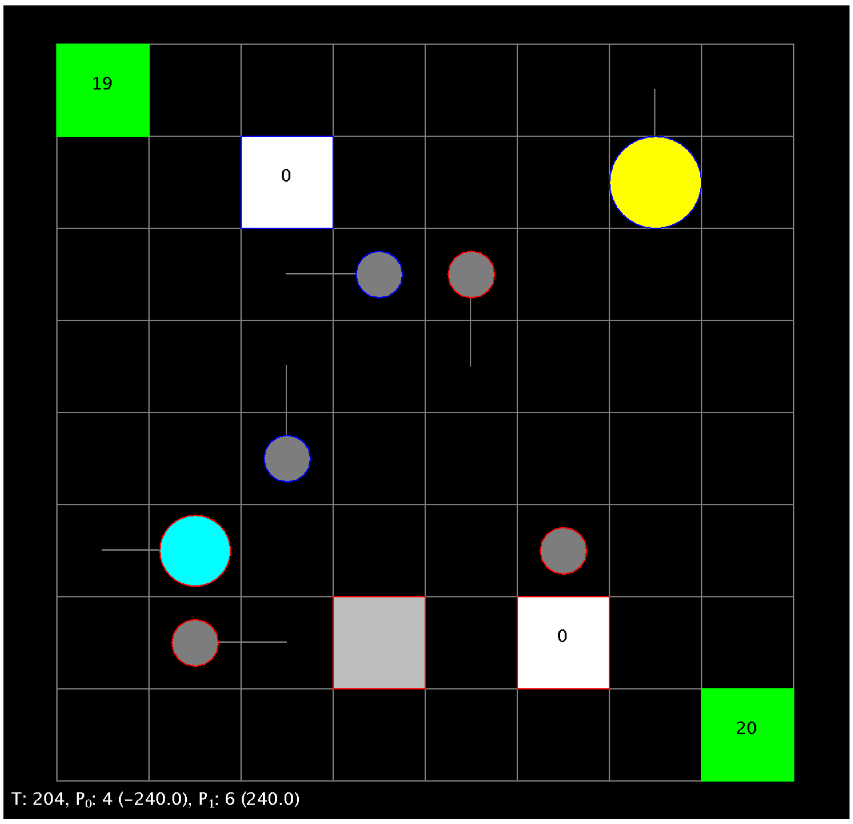
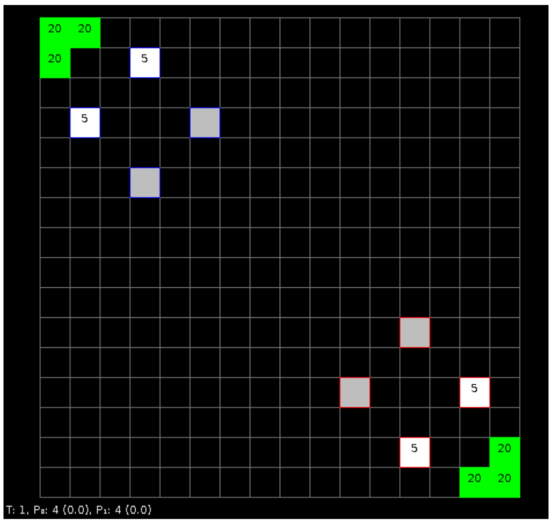


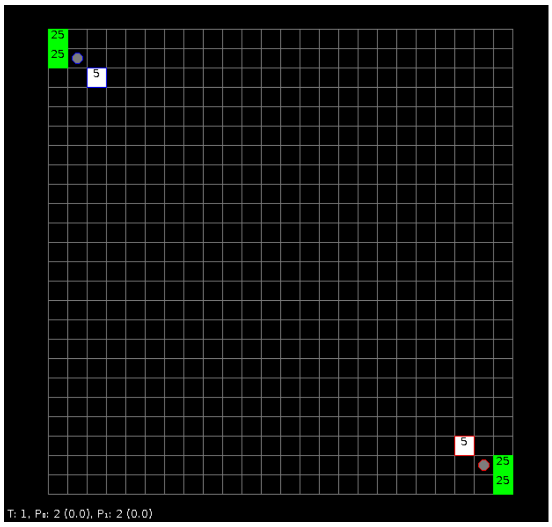
Figure 7 – *MicroRTS* match. Different outline colors represent different players.

Source: <https://sites.google.com/site/micrortsaicompetition/microrts> (Accessed on 10/2021)

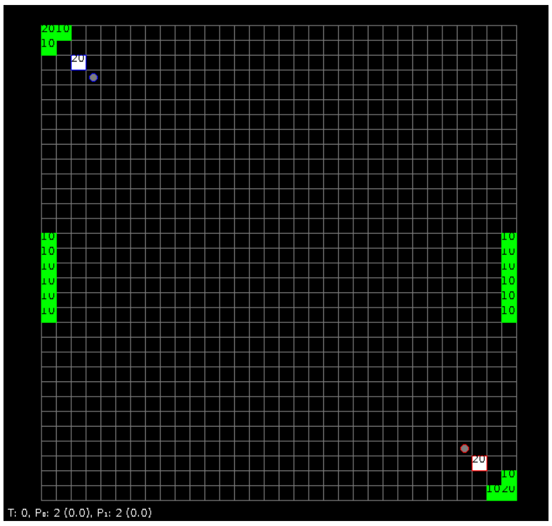
Larger maps often result in longer matches and other map features such as the distance of the player's units to the resources can significantly affect the players' strategies.



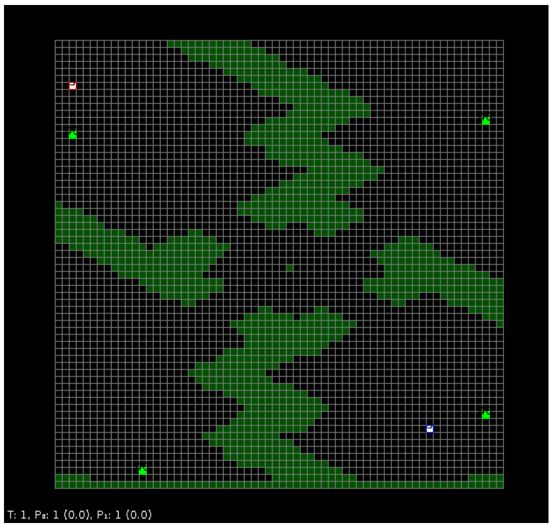
16×16 (TwoBasesBarracks)



24×24 (BasesWorkers)



32×32 (BasesWorkers)



64×64 (BloodBath-B)

Figure 8 – Maps used in the experiments.

We use four maps of different sizes, where the names in parenthesis are the maps names in the MicroRTS code base:[1] 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. Figure 8 shows the MicroRTS maps used in our experiments. We use the winner of the latest MicroRTS Competition, COAC, as σ_{-i} (ONTAÑÓN, 2020). COAC is a deterministic 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 appendix.

The Ψ function for MicroRTS is the average number of victories of p against σ_{-i} in

[1] <https://github.com/santiontanon/microrts>

two matches. Each map has two starting locations, so we run two 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.

5.2 Score Functions

5.2.1 Can't Stop & MicroRTS Action-Based Cloning Score

We consider an action-based cloning 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).

5.2.2 Can't Stop Observation-Based Cloning Score

We use an observation-based cloning 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).

5.2.3 MicroRTS Observation-Based Cloning Score

We use an observation-based cloning 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, respectively. 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).

5.3 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 player.

5.3.1 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 three self-play matches of this strategy, which only wins approximately 2.8% of the matches it plays against σ_{-i}. We call this strategy “Random”. We use a data set composed of 3 self-play matches of the GA strategy and a data set composed of three matches a human played with GA; the human won all matches.

5.3.2 MicroRTS

We also consider three data sets L for MicroRTS. The first L is composed of two 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 two 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 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 two self-play matches of COAC.

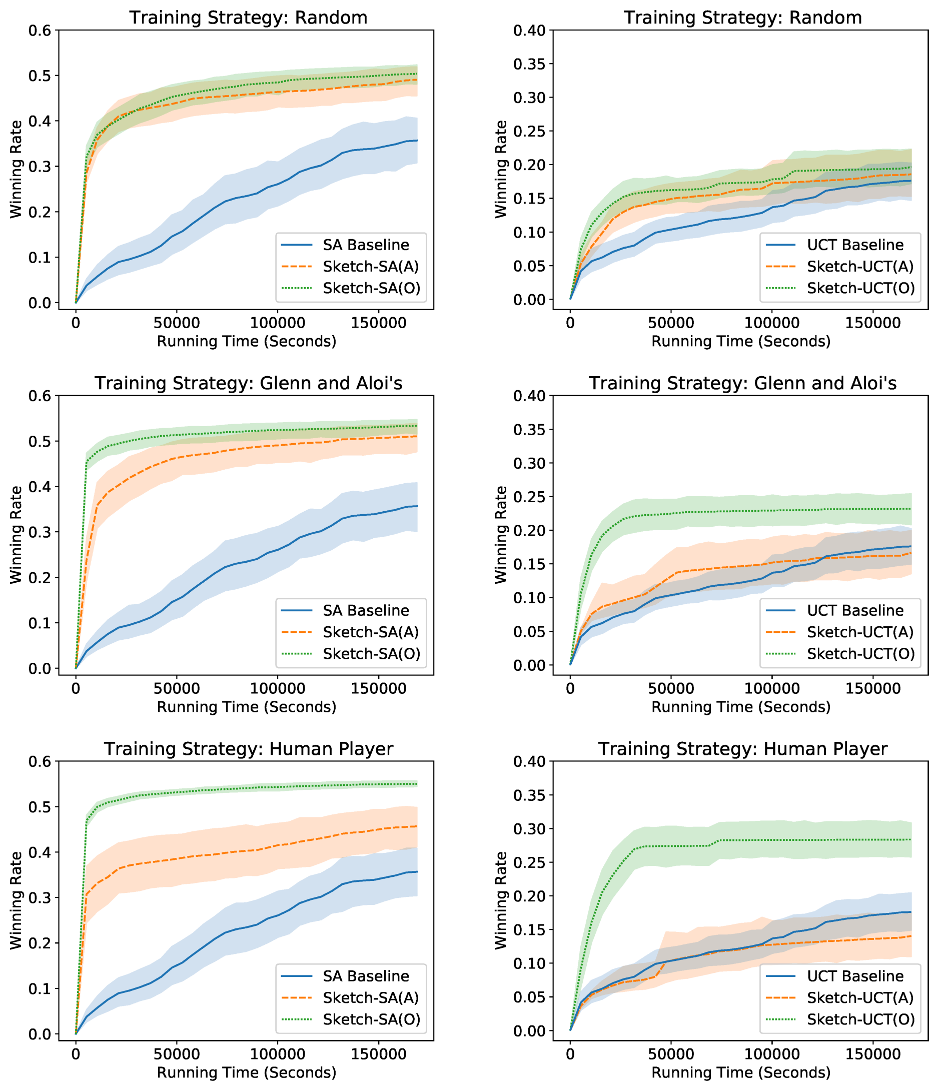


Figure 9 – Winning rate of SA (left) and UCT (right) variants.

5.4 Empirical Results: Can't Stop

Figure 9 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.

5.5 Empirical Results: MicroRTS

Figure 10 shows the results of the SA variants on MicroRTS. 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 in the game (e.g., one can train Ranged units after building a barracks, which might happen only once in a match). The plots for maps of size 16×16 and 24×24 in Figure 10 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

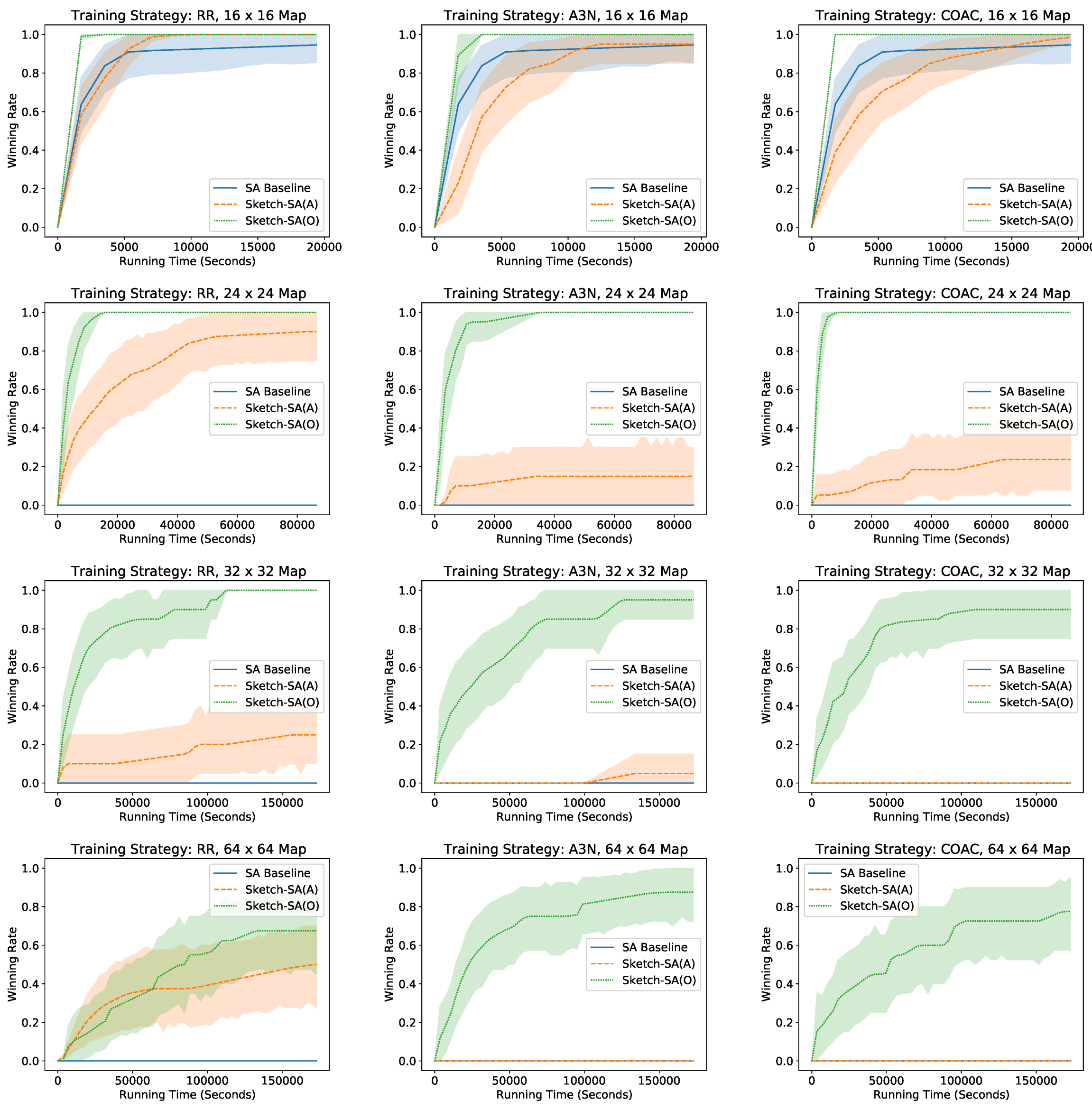


Figure 10 – Winning rate of strategies SA variants synthesize; the maps increase in size from top to bottom.

line represents the average winning rate and the shaded areas the standard deviation of 10 independent runs of each method.

Figure 11 shows the results of the UCT variants on MicroRTS. 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.

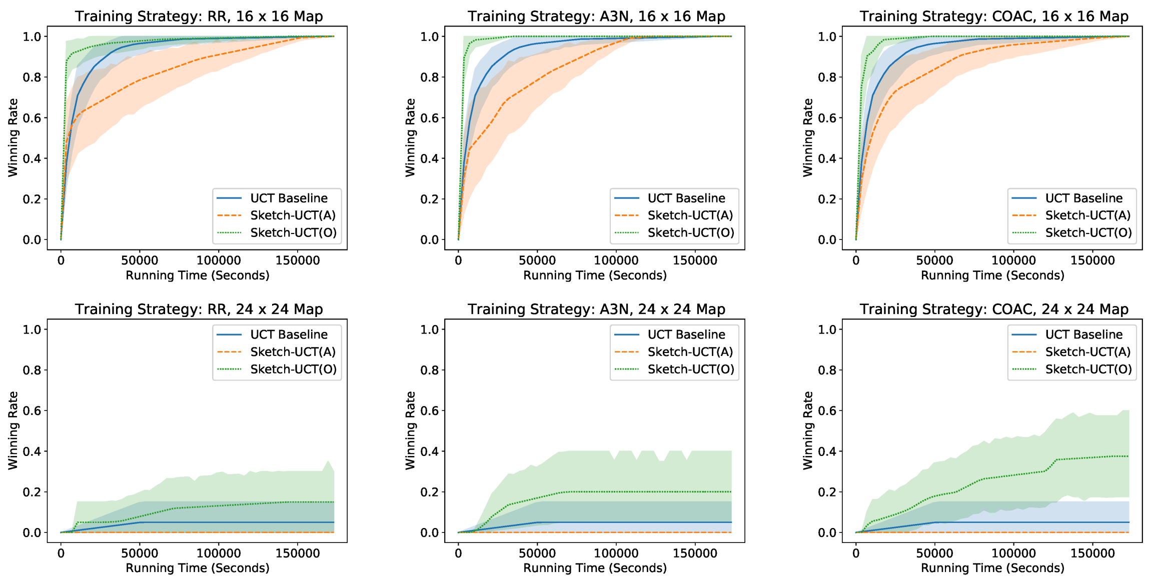


Figure 11 – Winning rate of strategies UCT variants synthesize; the maps increase in size from top to bottom.

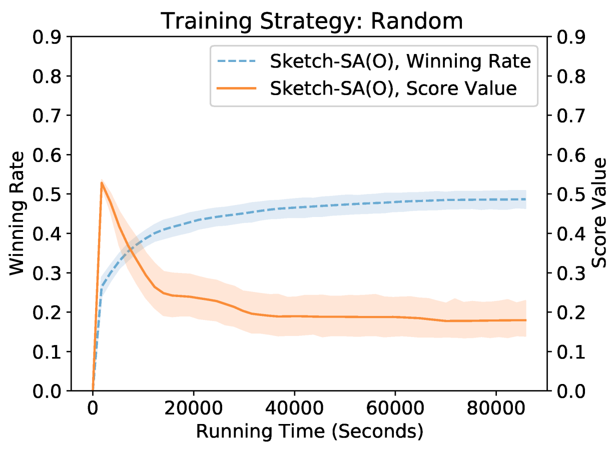


Figure 12 – Winning rate and cloning score value of the best program encountered during synthesis.

5.6 Winning Rate and Cloning Score Correlation

Figure 12 shows the relation between the winning rate (left y-axis) and the observation cloning score (right y-axis) for the best programmatic strategy encountered during the SA searches (sketch-search and BR-search) for the game of Can't Stop. The cloned strategy is Random. The spike in the score value in the beginning of the search represents the end of sketch-search, which optimizes for the cloning score. The BR-search starts where the score value drops.

During the sketch-search, where the evaluation function is C, it can be observed that while C-score grows, Ψ-value also grows. It is likely that the algorithm starts with a program that is not even compilable (and hence, with a winning rate and score value of

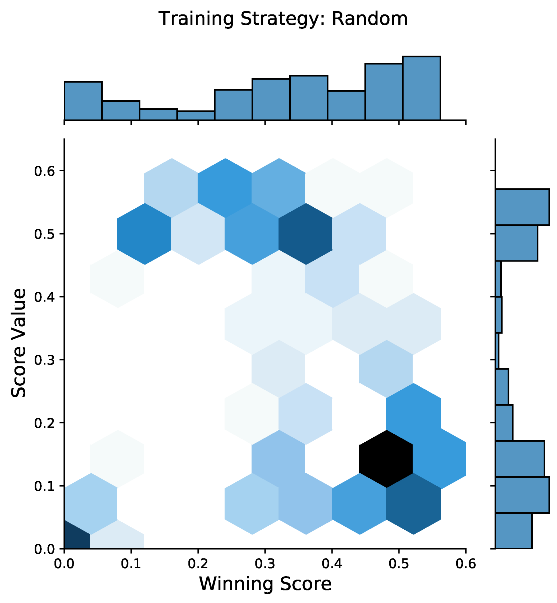


Figure 13 – Distribution of programs according to their winning score and cloning score.

0). Therefore, whenever the sketch-search is able to find a neighbor that is a compilable program and it starts to better imitate the oracle, this program is now able to possibly score a winning rate higher than 0, that is why we can observe an increase in both score value and winning rate against σ_{-i}.

However, in the BR-search where the evaluation function is Ψ, the Ψ-value increases while C-score decreases. The decrease in C-score is expected, as the strategy would become more and more similar to the Random strategy, which is a weak strategy, if the C-score value continued to grow.

Figure 13 presents a similar story by showing the distribution of a set of programmatic strategies according to their cloning score (y-axis) and winning rate against Glenn and Aloi's strategy (x-axis). The strategies with higher winning rates are not similar to the Random strategy (see the darker colors at the bottom-right corner). Despite the lack of correlation between winning rate and cloning score, cloning the behavior of Random can still help the search by allowing it to find helpful program sketches.

5.7 Sketch-Search-Only Results

Table 1 shows the results of using only the sketch-search with the behavioral cloning evaluation function $C(L, p)$ on the Can't Stop domain using SA. These results are shown in tabular mode because we play only the last program returned by the search against Glenn and Aloi's strategy.

Notice that when compared to the results achieved by sketch-search along with the BR-search, the winning rate of this approach is inferior in both types of sketch-search

Observation Type	L	WR - Sketch only	WR - Sketch & BR
A	Human	0.352 ± 0.199	0.491 ± 0.117
	Glenn and Aloi	0.424 ± 0.070	0.516 ± 0.092
	Random	0.182 ± 0.058	0.507 ± 0.085
O	Human	0.212 ± 0.216	0.555 ± 0.017
	Glenn and Aloi	0.368 ± 0.088	0.543 ± 0.024
	Random	0.244 ± 0.114	0.518 ± 0.058

Table 1 – Comparison of the winning rate against GA between the sketch-search-only approach (WR - Sketch-only) and sketch-search along with BR-search (WR - Sketch & BR) using different datasets L for cloning

(action-based (A) and observation-based (O)). This likely happens because the signal the algorithm receives from the sketch-search is only a signal representing if the algorithm is cloning the action chosen by the oracle. Therefore, the search algorithm will optimize the search towards this signal, and not towards the winning rate against Glenn and Aloi's strategy, that is, the signal given by Ψ. The BR-search with the Ψ evaluation function optimizes for programs that yield a better Ψ signal, and by the end of the execution, the synthesized program will better approximate a best response to the GA strategy.

An interesting result is that with this approach, better programs were synthesized when cloning the behavior of the data set generated by the Glenn and Aloi's strategy. We conjecture that because the DSL we used allows Glenn and Aloi's strategy to be synthesized exactly, it can then be able to synthesize scripts that clones accurately their strategy if given enough time, as opposed to the human's strategy, which is unlikely to be representable in our DSL.

5.8 BR-search UCT Tree Initialization

We experimented with two approaches to initialize the UCT tree of the BR-search: we can use the complete program p or the incomplete program given by the production rules represented by nodes in the UCT tree of the sketch-search. We empirically observed that initializing the program with the complete program sped up the synthesis step during the BR-search, as one can observe in Figure 14.

Even though both approaches are much faster than the baseline UCT, we hypothesize that initializing the BR-search by using p will bias the search towards programs that have a similar program structure to p. On the other hand, while it is true that the same applies if we initialize it by using an incomplete program, when compared to the former approach the bias is not as strong; therefore, UCT will take longer to explore the program space.

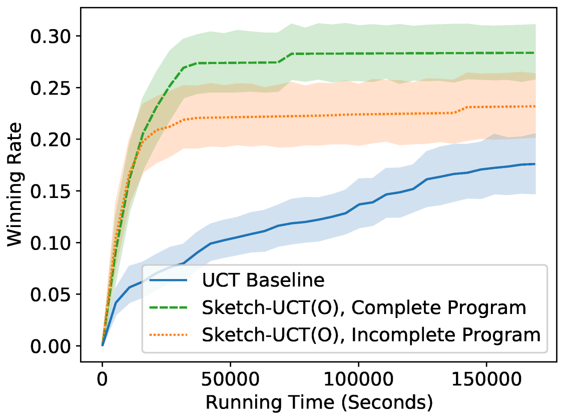


Figure 14 – Methods of initialization of the BR-search tree

5.9 Sample of Programmatic Strategy

In this section we discuss some programmatic strategies synthesized by Sketch-SA(O) that were able to defeat the σ_{-i} strategy in each domain.

5.9.1 Programs Synthesized for MicroRTS

Below it is shown 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.

```
 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)
```

The strategy receives a state s and assigns an action to each unit u 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.

Next, we present below the programs Sketch-SA(O) synthesized for the maps used in our experiments. The programs presented below are not simplified to improve readability.

```
def Sketch-SA-O-16x16(state s):
    for u in s:
        u.train(Ranged, Right, 4)
        u.harvest(9)
        u.attack(Closest)
        u.train(Worker, Up, 1)

def Sketch-SA-O-24x24(state s):
    for u in s:
        if not u.isBuilder():
            u.moveToUnit(Ally, LessHealthy)
        u.train(Ranged,Left,15)
        for u in s:
            u.idle()
        u.build(Barracks, EnemyDir, 100)
    for u in s:
        u.harvest(4)
        u.attack(LessHealthy)


def Sketch-SA-O-32x32(state s):
    for u in s:
        for u in s:
            for u in s:
              u.harvest(2)
            for u in s:
              u.train(Ranged,EnemyDir,6)
        u.train(Worker,Left,7)
        for u in s:
            u.build(Barracks,Right,1)
        u.idle()
        u.harvest(5)
        u.attack(Strongest)

def Sketch-SA-O-64x64(state s):
    for u in s:
        u.attack(Weakest)
        for u in s:
            u.train(Worker, Right, 6)
            u.harvest(7)
            for u in s:
              u.build(Barracks, Left, 10)
            if HaveUnitsAttacking(4):
               u.attack(Farthest)
            else:
              u.idle()
            u.train(Ranged, Down, 50)
```

5.9.2 Programs Synthesized for Can't Stop

We use the structure of GLENN; ALOI (2009)'s strategy and the synthesizers need to generate a program that fill the holes responsible for the yes-no and column decisions. The program with holes is shown below, with the two question marks denoting the instructions that need to be generated.

```
def  get_action(self, state):
    actions = state.available_moves()
    if actions == ['y', 'n']:
        score =  ?
        if win_after_n(state):
            return 'n'
        elif available_columns(state):
            return 'y'
        else:
            if score >= 29:
                return 'n'
            else:
                return 'y'
    else:
        index = ?
        return actions[index]
```

The method `win_after_n(state)` checks if the player will win in case they choose to end their turn and the method `available_columns(state)` checks if there are available columns for the player to choose given the current board and dice configuration. The pseudocode below presents Glenn and Aloi's strategy written with our DSL.

```
def  get_action(self, state):
    actions = state.available_moves()
    if actions == ['y', 'n']:
        score =  sum(map(lambda x: (NumberAdvancedThisRound+1)*
           progress_value, neutrals)) + DifficultyScore
        if win_after_n(state):
            return 'n'
        elif available_columns(state):
            return 'y'
        else:
            if score >= 29:
                return 'n'
            else:
                return 'y'
    else:
        index = argmax(map(lambda x: sum(map(lambda x:
           NumberAdvancedByAction * move_value - 6 * IsNewNeutral,
           locallist)), actions))
        return actions[index]
```

The following program was synthesized by Sketch-SA(O) and achieved the winning rate of 59.64% against Glenn and Aloi's strategy in 5,000 matches of the game.

```
yes-no: ((DifficultyScore * DifficultyScore) - (sum(map((lambda x :
     sum(neutrals)), neutrals)) - (DifficultyScore + sum(map((lambda
     x : (progress_value * (NumberAdvancedThisRound *
    PositionsPlayerHasSecuredInColumn))), neutrals)))))
column: argmax(map((lambda x : sum(map((lambda x : (
    PositionsPlayerHasSecuredInColumn + move_value)), locallist))),
    actions))
```

For the yes-no decision, the synthesized program has a similar structure to Glenn and Aloi's strategy. Namely, it accounts for the columns in which the neutral tokens are positioned and the position of the neutral tokens in each column. This strategy differs from Glenn and Aloi's because it places a higher weight on the difficulty of a state when deciding whether to continue or to stop playing (`DifficultyScore` appears as a quadratic term in the synthesized program, while it is linear in Glenn and Aloi's strategy). The synthesized strategy accounts for the number of rows conquered in previous rounds (`PositionsPlayerHasSecuredInColumn` is used in the synthesized strategy, but not in Glenn and Aloi's).

The synthesized strategy for the column decisions fixes a weakness in Glenn and Aloi's strategy. Namely, their strategy ignores the rows the player has conquered in previous rounds of the game. By contrast, the synthesized strategy uses function `PositionsPlayerHasSecuredInColumn` in the `argmax` operator. While both strategies prefer even-numbered columns, in some games, depending on the dice roll, the player might have to play on odd-numbered columns. Glenn and Aloi's strategy continues to prefer even-numbered columns, even if they have achieved a promising position on odd-numbered ones. The synthesized strategy eventually prefers to continue play on odd-numbered columns if they have conquered a "sufficiently large" number of rows.

6 Conclusions

In this work 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.

For future experiments, it would be interesting to analyze if the number of instances used by the sketch-search has any effects on the resulting cloning score and winrate of the strategies synthesized. A more thorough experiment on different search algorithms besides SA and UCT could bring insightful remarks on how these search algorithms behave with different grammars and domains that they are applied to when using the algorithm discussed in this work.

A more comprehensive study on the variations of the SA algorithm could bring new insights on its behavior on program synthesis, such as an extensive study on different temperature schedules and cost function implementations. The same could be done with the UCT algorithm, as it has many variations in the literature.

References

AHMED, U. Z.; GULWANI, S.; KARKARE, A. Automatically generating problems and solutions for natural deduction. In: **Twenty-Third International Joint Conference on Artificial Intelligence**. [S.l.: s.n.], 2013.

ALUR, R. et al. Syntax-guided synthesis. In: **Formal Methods in Computer-Aided Design (FMCAD)**. [S.l.]: IEEE, 2013. p. 1–8.

AUER, P.; CESA-BIANCHI, N.; FISCHER, P. Finite-time analysis of the multiarmed bandit problem. **Machine learning**, Springer, v. 47, n. 2-3, p. 235–256, 2002.

BAIN, M.; SAMMUT, C. A framework for behavioural cloning. In: **Machine Intelligence 15**. [S.l.]: Oxford University Press, 1996. p. 103–129.

BASTANI, O.; PU, Y.; SOLAR-LEZAMA, A. Verifiable reinforcement learning via policy extraction. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2018. p. 2499–2509.

BENBASSAT, A.; SIPPER, M. Evolving board-game players with genetic programming. In: **Genetic and Evolutionary Computation Conference**. [S.l.: s.n.], 2011. p. 739–742.

BUTLER, E.; TORLAK, E.; POPOVIĆ, Z. Synthesizing interpretable strategies for solving puzzle games. In: **Proceedings of the 12th International Conference on the Foundations of Digital Games**. [S.l.: s.n.], 2017. p. 1–10.

CANAAN, R. et al. Evolving agents for the hanabi 2018 cig competition. In: IEEE. **2018 IEEE Conference on Computational Intelligence and Games**. [S.l.], 2018. p. 1–8.

CAZENAVE, T. Monte-carlo expression discovery. **International Journal on Artificial Intelligence Tools**, v. 22, n. 01, p. 1250035, 2013.

CHASLOT, G. et al. Monte-carlo tree search: A new framework for game ai. In: . [S.l.: s.n.], 2008.

CHEUNG, A.; SOLAR-LEZAMA, A.; MADDEN, S. Using program synthesis for social recommendations. In: **Proceedings of the 21st ACM international conference on Information and knowledge management**. [S.l.: s.n.], 2012. p. 1732–1736.

COULOM, R. Efficient selectivity and backup operators in monte-carlo tree search. In: SPRINGER. **International conference on computers and games**. [S.l.], 2006. p. 72–83.

DEURSEN, A. V.; KLINT, P.; VISSER, J. Domain-specific languages: An annotated bibliography. **ACM Sigplan Notices**, ACM New York, NY, USA, v. 35, n. 6, p. 26–36, 2000.

FREITAS, J. M. D.; SOUZA, F. R. de; BERNARDINO, H. S. Evolving controllers for mario ai using grammar-based genetic programming. In: IEEE. **2018 IEEE Congress on Evolutionary Computation (CEC)**. [S.l.], 2018. p. 1–8.

GLENN, J. R.; ALOI, C. J. A generalized heuristic for can't stop. In: **FLAIRS Conference**. [S.l.: s.n.], 2009.

GULWANI, S. et al. Program synthesis. **Foundations and Trends® in Programming Languages**, Now Publishers, Inc., v. 4, n. 1-2, p. 1–119, 2017.

HAJEK, B. Cooling schedules for optimal annealing. **Mathematics of operations research**, INFORMS, v. 13, n. 2, p. 311–329, 1988.

HUSIEN, I.; SCHEWE, S. Program generation using simulated annealing and model checking. In: NICOLA, R. D.; KÜHN, E. (Ed.). **Software Engineering and Formal Methods**. [S.l.]: Springer International Publishing, 2016. p. 155–171. ISBN 978-3-319-41591-8.

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

KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by simulated annealing. **science**, American association for the advancement of science, v. 220, n. 4598, p. 671–680, 1983.

KOCSIS, L.; SZEPESVÁRI, C. Bandit based monte-carlo planning. In: SPRINGER. **European conference on machine learning**. [S.l.], 2006. p. 282–293.

LANCTOT, M. et al. A unified game-theoretic approach to multiagent reinforcement learning. In: **Proceedings of the International Conference on Neural Information Processing Systems**. [S.l.: s.n.], 2017. p. 4193–4206.

LELIS, L. H. S. Planning algorithms for zero-sum games with exponential action spaces: A unifying perspective. In: **Proceedings of the International Joint Conference on Artificial Intelligence**. [S.l.]: International Joint Conferences on Artificial Intelligence Organization, 2020. p. 4892–4898. Survey track.

MARIÑO, J. R. H. et al. Programmatic strategies for real-time strategy games. **Proceedings of the AAAI Conference on Artificial Intelligence**, v. 35, n. 1, p. 381–389, 2021.

MORAES, R. O. et al. Action abstractions for combinatorial multi-armed bandit tree search. In: AAAI. **Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**. [S.l.], 2018. p. 74–80.

NOURANI, Y.; ANDRESEN, B. A comparison of simulated annealing cooling strategies. **Journal of Physics A: Mathematical and General**, IOP Publishing, v. 31, n. 41, p. 8373, 1998.

NYE, M. et al. Learning to infer program sketches. In: CHAUDHURI, K.; SALAKHUTDINOV, R. (Ed.). **Proceedings of the 36th International Conference on Machine Learning**. [S.l.]: PMLR, 2019. (Proceedings of Machine Learning Research, v. 97), p. 4861–4870.

ONTANÓN, S. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In: **Ninth Artificial Intelligence and Interactive Digital Entertainment Conference**. [S.l.: s.n.], 2013.

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

ROSS, S.; GORDON, G.; BAGNELL, D. A reduction of imitation learning and structured prediction to no-regret online learning. In: **Proceedings of the International Conference on Artificial Intelligence and Statistics**. [S.l.]: PMLR, 2011. (Proceedings of Machine Learning Research, v. 15), p. 627–635.

SAMMUT, C. et al. Learning to fly. In: **Machine Learning Proceedings 1992**. [S.l.]: Elsevier, 1992. p. 385–393.

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

SINGH, R. **Accessible programming using program synthesis.** Tese (Doutorado) — Massachusetts Institute of Technology, Department of Electrical Engineering . . . , 2014.

SOLAR-LEZAMA, A. The sketching approach to program synthesis. In: SPRINGER. **Asian Symposium on Programming Languages and Systems**. [S.l.], 2009. p. 4–13.

STANESCU, M. et al. Evaluating real-time strategy game states using convolutional neural networks. In: IEEE. **Proceedings IEEE Conference on Computational Intelligence and Games**. [S.l.], 2016. p. 1–7.

TORABI, F.; WARNELL, G.; STONE, P. Behavioral cloning from observation. **arXiv preprint arXiv:1805.01954**, 2018.

TORABI, F.; WARNELL, G.; STONE, P. Behavioral cloning from observation. In: **Proceedings of the International Joint Conference on Artificial Intelligence**. [S.l.: s.n.], 2018.

VERMA, A. et al. Imitation-projected programmatic reinforcement learning. In: **Advances in Neural Information Processing Systems**. [S.l.]: Curran Associates, Inc., 2019. v. 32, p. 1–12.

VERMA, A. et al. Programmatically interpretable reinforcement learning. In: **Proceedings of the International Conference on Machine Learning**. [S.l.: s.n.], 2018. p. 5052–5061.

Appendix

APPENDIX A – Domain-Specific Languages used for Can't Stop and MicroRTS

A.1 Domain-Specific Language for Can't Stop

The DSL we developed for Can't Stop is described by the following context-free grammar.

$$
\begin{aligned}
S \rightarrow \quad & AA \\
A \rightarrow \quad & \text{if}(B < B) \text{ then } A \text{ else } A \quad | \quad \text{argmax}(L) \quad | \quad \text{sum}(L) \\
\rightarrow \quad & E \otimes E \\
B \rightarrow \quad & N \quad | \quad E \otimes E \\
E \rightarrow \quad & E \otimes E \quad | \quad N \quad | \quad \text{sum}(L) \quad | \quad L_2 \quad | \quad F_2 \\
L \rightarrow \quad & \text{map}(\lambda F_1, L) \quad | \quad L_1 \quad | \quad \text{locallist} \\
\lambda F_1 \rightarrow \quad & \text{sum}(L) \quad | \quad \text{map}(\lambda F_1, L) \quad | \quad E \otimes E \\
F_2 \rightarrow \quad & \text{NumberAdvancedThisRound} \\
\rightarrow \quad & \text{NumberAdvancedByAction} \\
\rightarrow \quad & \text{PositionsPlayerHasSecuredInColumn} \\
\rightarrow \quad & \text{PositionsOpponentHasSecuredInColumn} \\
\rightarrow \quad & \text{DifficultyScore} \\
\rightarrow \quad & \text{IsNewNeutral} \\
L_1 \rightarrow \quad & \text{neutrals} \quad | \quad \text{actions} \\
L_2 \rightarrow \quad & \text{progress\_value} \quad | \quad \text{move\_value} \\
N \rightarrow \quad & 0 \quad | \quad 1 \\
\otimes \rightarrow \quad & + \quad | \quad - \quad | \quad *
\end{aligned}
$$

This DSL allows our system to synthesize actions for both decisions in Can't Stop: yes-no decision and column decision. λF_1 represents lambda functions used as parameters for the `map` operator and F_2 is the set of domain-specific functions that provide information about a state of the game. Next, we describe these functions:

- *NumberAdvancedThisRound*: Calculates the number of cells the player has advanced in the current round

- *NumberAdvancedByAction*: Calculates the number of cells the player will advance if they take the action passed as a parameter
- *PositionsPlayerHasSecuredInColumn*: Calculates how many cells the player has secured in a column passed as parameter
- *PositionsOpponentHasSecuredInColumn*: Calculates how many cells the opponent has secured in a column passed as parameter
- *DifficultyScore*: Calculates a difficulty score (GLENN; ALOI, 2009) of a state, where the value returned depends on the position of the neutral tokens; a state is considered more difficult if the neutral tokens are on odd columns or if all tokens are either on columns with numbers smaller than 7 or greater than 7
- *IsNewNeutral*: Returns 1 if the action will use a neutral token, returns 0 otherwise

Next, we describe the lists from L, L_1 and L_2 used in our DSL:

- *locallist*: The set of actions is represented as a list of lists. When used in a `map` operator, we call the lists inside an action list a "locallist," which is represented in the DSL as l_1.
- *neutrals*: List that represents which columns the neutral tokens are located given the state passed as parameter
- *actions*: List that represents the actions available for the state passed as parameter
- *progress_value*: List that represents weights of the columns used in Glenn and Aloi's strategy. In their strategy this set of weights is used in the yes-no decision. In our DSL, this list can be used by both actions. $\text{progress\_value} = [7, 7, 3, 2, 2, 1, 2, 2, 3, 7, 7]$
- *move_value*: List that represents weights of the columns used in Glenn and Aloi's strategy in the column decision. In our DSL, this list can be used in both decisions. $\text{move\_value} = [7, 0, 2, 0, 4, 3, 3, 4, 0, 2, 0, 7]$

A.2 Domain-Specific Language for MicroRTS

Our DSL for MicroRTS is inspired on that of MARIÑO et al. (2021). The DSL is described by the following context-free grammar.

$$
\begin{array}{rl}
S \to & SS \;\mid\; \text{for S} \;\mid\; \text{if}(B) \text{ then } S \;\mid\; \text{if}(B) \text{ then } S \text{ else } S \;\mid\; C \;\mid\; \lambda \\
B \to & \text{HasNumberOfUnits(T, N)} \;\mid\; \text{OpponentHasNumberOfUnits(T, N)} \\
\to & \text{HasLessNumberOfUnits(T, N)} \;\mid\; \text{HaveQtdUnitsAttacking(N)} \\
\to & \text{HasUnitWithinDistanceFromOpponent(N)} \\
\to & \text{HasNumberOfWorkersHarvesting(N)} \;\mid\; \text{is\_Type(T)} \;\mid\; \text{IsBuilder} \\
\to & \text{CanAttack} \;\mid\; \text{HasUnitThatKillsInOneAttack} \\
\to & \text{OpponentHasUnitThatKillsUnitInOneAttack} \\
\to & \text{HasUnitInOpponentRange} \;\mid\; \text{OpponentHasUnitInPlayerRange} \\
\to & \text{CanHarvest} \\
C \to & \mathit{Build}(T, D, N) \;\mid\; Train(T, D, N) \;\mid\; moveToUnit(T_p, O_p) \\
\to & \mathit{Attack}(O_p) \;\mid\; Harvest(N) \;\mid\; Idle \;\mid\; MoveAway \\
T \to & \text{Base} \;\mid\; \text{Barracks} \;\mid\; \text{Ranged} \;\mid\; \text{Heavy} \;\mid\; \text{Light} \;\mid\; \text{Worker} \\
N \to & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 10 \mid 15 \\
\to & 20 \mid 25 \mid 50 \mid 100 \\
D \to & \text{EnemyDir} \;\mid\; \text{Up} \;\mid\; \text{Down} \;\mid\; \text{Right} \;\mid\; \text{Left} \\
O_p \to & \text{Strongest} \;\mid\; \text{Weakest} \;\mid\; \text{Closest} \;\mid\; \text{Farthest} \\
\to & \text{LessHealthy} \;\mid\; \text{MostHealthy} \;\mid\; \text{Random} \\
T_p \to & \text{Ally} \;\mid\; \text{Enemy}
\end{array}
$$

This DSL allows nested loops and conditionals. It contains several Boolean functions (B) and command-oriented functions (C) that provide either information about the current state of the game or commands for the ally units.

Next, we describe the Boolean functions used in our DSL.

- *HasNumberOfUnits(T,N)*: Checks if the ally player has N units of type T
- *OpponentHasNumberOfUnits(T,N)*: Checks if the opponent player has N units of type T
- *HasLessNumberOfUnits(T,N)*: Checks if the ally player has less than N units of type T
- *HaveQtdUnitsAttacking(N)*: Checks if the ally player has N units attacking the opponent

- *HasUnitWithinDistanceFromOpponent(N)*: Checks if the ally player has a unit within a distance N from a opponent's unit
- *HasNumberOfWorkersHarvesting(N)*: Checks if the ally player has N units of type Worker harvesting resources
- *is_Type(T)*: Checks if a unit is an instance of Type T
- *IsBuilder*: Check if a unit is of type Worker
- *CanAttack*: Checks if a unit can attack
- *HasUnitThatKillsInOneAttack*: Checks if the ally player has a unit that kills an opponent's unit with one attack action
- *OpponentHasUnitThatKillsUnitInOneAttack*: Checks if the opponent player has a unit that kills an ally's unit with one attack action
- *HasUnitInOpponentRange*: Checks if an unit of the ally player is within attack range of an opponent's unit
- *OpponentHasUnitInPlayerRange*: Checks if an unit of the opponent player is within attack range of an ally's unit
- *CanHarvest*: Checks if a unit can harvest resources

Next, we describe the command-oriented functions used in our DSL:

- *Build(T,D,N)*: Trains N units of type T on a cell located on the D direction of the unit
- *Train(T,D,N)*: Trains N units of type T on a cell located on the D direction of the structure responsible for training them
- *moveToUnit(T_p, O_p)*: Commands a unit to move towards the player T_p following a criterion O_p
- *Attack(O_p)*: Commands a unit to attack units of the opponent player following a criterion O_p
- *Harvest(N)*: Sends N Worker units to harvest resources
- *Idle*: Commands a unit to stay idle and attack if an opponent units comes within its attack range
- *MoveAway*: Commands a unit to move in the opposite direction of the player's base

T represents the types a unit can assume. N is a set of integers. D represents the directions available used in action functions. O_p is a set of criteria to select an opponent unit based on their current state. T_p represents the set of players. See the different types, integers, directions, criteria, and players we used in the context-free grammar above.