RUBENS DE OLIVEIRA MORAES FILHO

**GUIDING AND TEACHING: SYNTHESIS OF PROGRAMMATIC
STRATEGIES IN TWO CONTEXTS.**

Advancement Document presented to the Federal University of Viçosa, as part of the requirements of the Graduate Program in Computer Science, to obtain the title of *Doctor Scientiae.*

Guidance: Dr. Levi Henrique Santana de Lelis

**VIÇOSA - MINAS GERAIS
2022**

Resumo

Moraes Filho, R. O., Universidade Federal de Viçosa, junho de 2022. **Guiding and Teaching: Synthesis of Programmatic Strategies in Two Contexts.** Orientador: Dr. Levi Henrique Santana de Lelis.

RESUMO

Nesta pesquisa, propõem-se o uso de síntese de programas para geração de estratégias programáticas dominantes e interpretáveis. Em um primeiro momento, investigamos como guiar a busca para a síntese de estratégias programáticas (scripts) para jogos de soma-zero de dois jogadores. O objetivo do primeiro momento é mostrar como guiar o processo de síntese para criar estratégias programáticas puras e dominantes. Para guiar a busca local, introduzimos heurísticas baseadas in teoria de jogos, como Iterated Best Response (IBR), Fictitious Play (FP) e Double Oracle (DO). As heurísticas são importantes na síntese de estratégias programáticas por impactar diretamente a robustes da estratégia sintetizada. Se a heurística é míope e perde informação necessária para guiar a busca, a convergência para uma estratégia dominante pode ser muito devagar e impraticável. Introduzimos o método Neighborhood Curriculum (NC) como uma heurística baseada na análise de estatísticas de soluções vizinhas geradas durante a busca local. NC permite uma melhor condução da busca e síntese de estratégias que os outros métodos, como demonstramos empíricamente nos testes. Também exploramos o potencial e limitações de cada uma das heurísticas avaliadas. A heurística NC é computacionalmente menos custosa para guiar a busca, e mais robusta se comparada com as heurísticas utilizadas como *baseline*. Para alcançar essas conclusões, nós avaliamos as heurísticas em três domínios: Climbing Monkey, Poachers and Rangers, e MicroRTS. No segundo momento, exploramos a interpretabilidade das estratégias programáticas através de avaliações sobre o que humanos podem aprender com elas. Para isto, introduziremos um sistema de iteração humano-computador. Este sistema têm a possibilidade de permitir que profissionais aprendam como escrever melhores scripts através da análise e interpretação de melhores estratégias programáticas. Para trabalhos futuros, apresentamos ainda duas novas propostas: uma versão do *Alpha-League* (parte projeto Alpha-Star da DeepMind) para síntese de estratégias programáticas

para jogos de soma-zero usando busca local; e uma proposta de algoritmo pra combinar diferente estratégias programáticas em uma Abstract Syntax Tree.

Palavras-chave: Síntese de programas. Teoria de Jogos. Busca Local. Interligência Artificial Explicável. Estratégias Programáticas.

Abstract

Moraes Filho, R. O., Universidade Federal de Viçosa, June, 2022. **Guiding and Teaching: Synthesis of Programmatic Strategies in Two Contexts.** Advisor: Dr. Levi Henrique Santana de Lelis.

ABSTRACT

In this research, we propose the use of program synthesis to generate strong and interpretable programmatic strategies. In the first moment, we investigated how guide the search to synthesize programmatic strategies (scripts) for two-player zero-sum games. The goal of this first moment is shown how guiding the synthesis process for programmatic strategies to create pure dominant scripts. To guide the local search, we introduce heuristics based on game theory, such as Iterated Best Response (IBR), Fictitious Play (FP), and Double-Oracle (DO). The driving heuristics are important in the synthesis of programmatic strategies as they directly impact the robustness of the synthesized strategies. If the heuristic is myopic and loses necessary information to guide the search, the convergence for a dominant strategy might be too slow to be practical. We introduce Neighborhood Curriculum (NC), a heuristic based on the analysis of statistics from the neighbor solutions generated during local search. NC provides better guidance to the search and synthesis of strong strategies than DO, FP, and SP, as we empirically demonstrated in our tests. We also explored the potentials and limitations of each one of the heuristics evaluated. The NC heuristic was computationally less costly to guide the search, and more robust compared to the guidance provided by the other heuristics. To reach these conclusions, we evaluate the heuristics in three domains: Jumping Monkey, Poachers and Rangers, and MicroRTS. On the second moment, we will explore the interpretability on programmatic strategies by evaluating what humans can learn from these strategies. For this, we will present the human-computer interaction system. This system has the possibility to allow professionals to learn how to write better scripts through the analysis and interpretation of improved programmatic strategies. For work in the furute, we present two proposals: a version of Alpha-League (part of DeepMind's famous Alpha-Star project) to synthesize programmatic strategies for zero-sum games using local search; and a proposal for algorithm to combine different programmatic strategies in an Abstract Syntax Tree.

List of Figures

List of Tables

Acronyms

$\mathcal{BR}$ Best Response. 18, 32, 34, 36

AI Artificial Intelligence. 24, 25, 64, 72

AST Abstract Syntax Tree. 2, 3, 22, 33, 63, 67

CEGIS Counter-Example-Guided Inductive Synthesis. 24

CM Climbing Monkey. 1, 5, 13, 44, 45, 49–51, 58

DO Double Oracle. 1, 20, 21, 35, 36, 38

DRL Deep Reinforcement Learning. 25

DS Dynamic Scripting. 30

DSL Domain-Specific Language. 13, 21, 22, 25, 28–33, 47, 48, 66, 70–72

FP Fictitious Play. 1, 19, 20, 34

HC Hill-Climbing. 6, 11, 26, 27, 32, 49–52, 54, 55, 59, 66

HCI Human-Computer Interaction. 13

IBR Iterated Best Response. 1, 3, 19, 29, 34–37, 40, 44, 49–51, 53, 58, 59, 63, 65

Lasi Domain-Specific Language Simplifier. 28, 29

LP Linear Programming. 35–38

ML Machine Learning. 25

NC Neighborhood Curriculum. 1, 5, 13, 37, 40–44, 49–51, 53, 58, 59, 65, 66, 72

P&R Poachers and Rangers. 1, 5, 7, 13, 37, 38, 44, 45, 49, 50, 58, 59

PBE Programming by Example. 21–24

PIRL Programmatically Interpretable Reinforcement Learning. 25

PRL Programmatically Reinforcement Learning. 29, 30

PSRO policy-space response oracles. 35

Contents

1 Introduction

Program synthesis is defined as the process of automatically synthesizing computer programs that satisfy constraints and requirements based on examples provided by users. By searching in a specific program language, or domain-specific language, the program synthesizer can find a solution as per the requirements (GULWANI; POLOZOV; SINGH, 2017). However, the search space can be extremely large, and the program synthesizer needs to deal with all possibilities to find the desired solution, which is costly in time and computer resources. This challenge emphasizes the importance of developing smart heuristics to guide search algorithms. Another interesting part of the synthesis process lies in the interpretability and explainability of the solutions produced during the process. Considering the applicability of program synthesis, we focus on two components: the usage of program synthesis to guide and generate programmatic strategies for two-player zero-sum games; the interpretability aspect of the programmatic strategies to teach humans. We tested these hypothesis directly in game scenarios, and in doing so, we hope to discover highly applicable strategies that will advance the synthesis of programmatic strategies and how they are interpreted by humans.

In the current researches on game scenarios, the same used on this project, learning-based and self-play algorithms have achieved results that surpassed humans and professional players. For example, AlphaGo (SILVER et al., 2016; SILVER et al., 2017), AlphaStar (VINYALS et al., 2019a), OpenAI Five (BERNER et al., 2019), and Deep-Mind's FTW (JADERBERG et al., 2019) are intelligent systems, composed by one or more algorithms that surpass human players in Go, StarCraft II, Dota 2, and Quake III, respectively. However, deep neural networks like AlphaStar are costly and complicated to be trained. Moreover, these algorithms, implemented as black-box solutions, suffer from a lack of explainability.

Algorithms for program synthesis develop intelligent agents (DAVID; KROENING, 2017) that avoid the black-box issue commonly found in techniques like deep learning. The primary advantage of program synthesis methods relies on their trend of explainability. Programmatic strategies, or *scripts*, are agents that play a sort of games, specially two-player sum-zero games. They tend to be more interpretable and are easily modified by experts or professionals in the domain. However, the strategies discovered by the methods for synthesis of script do not have the same potential compared other search algorithms, such as Alpha-Beta Tree-Search (CHURCHILL; SAFFIDINE; BURO, 2012) or Monte Carlo Tree-Search (ONTAÑÓN, 2017), or reinforcement learning algorithms (SILVER et al., 2016; SILVER et al., 2017).

Guidance over large search space is a crucial aspect of the program synthesis

for programmatic strategies. As Medeiros, Aleixo e Lelis (2022) have explored, guidance during the search process to synthesize programmatic strategies is challenging considering the search space. Mariño et al. (2021) developed an algorithm that uses program synthesis techniques which guides the search using self-play. The authors show that its method has the potential to be competitive compared with strategies developed by professionals in the domain of MicroRTS.

The main contribution of this work lies in the guidance process to synthesize pure and dominant strategies using program synthesis. Following the previous work, we explore how self-play, also called Iteration Best Response (IBR), can be used as a heuristic to guide the synthesis process. In addition, we introduce two new heuristics based on game theory. The first, Simple Cumulative Fictitious Play (SCFP), is based on the Fictitious Play method, and the second, Script-Space Double Oracle (SSDO), is based on the Double-Oracle method. Both heuristics guide local search algorithm more efficiently than IBR for script synthesis. However, we discovered possible weaknesses in the heuristics evaluated so far. These weaknesses allow the search to rewind and make the search slower. Based on these discoveries, we propose a new method called Neighborhood Curriculum (NC).

Neighborhood Curriculum is a heuristic focused on composing a strategy profile, also called curriculum, able to guide a local search algorithm in the task of synthesizing programmatic strategies. To create this curriculum, NC uses information from the adjacent solutions (or neighbor solution) generated during the local search. NC uses the neighborhood statistics to select the next subgroup of strategies. This set of strategies comprises the informative strategies used to guide the search. The method used by NC to select the set of strategies is a greedy method based on the neighborhood statistics collected during the search. In order to test our approach, we evaluate the four heuristics on three test-beds: Poachers and Rangers, Climbing Monkey, and MicroRTS. In all test-beds, Neighborhood Curriculum guides the search efficiently by using less budget than its counterparts.

The aspect of interpretability might allow codified knowledge from a solution to be informative and understood by humans. This interaction between humans and algorithms can be used to produce technical artifacts through collaboration (ZHANG et al., 2020). To evaluate interpretability, we used the programmatic strategies in a Human-Computer Interaction style to help programmers identify weaknesses in their code. To do this job, we developed a Domain-Specific Language for MicroRTS to be used as a common search space. We also developed an interface for programmers to write scripts using DSL.

We intend to evaluate how professionals can learn from our method by allowing the programmer to interact with our interface and the algorithm that is used to suggest improvements. During this interaction, the programmers will write their own programmatic strategies using a common interface and submit those scripts to the experiment.

Next, the programmers receive feedback from our algorithm about their initial code. After receiving the feedback, the programmer answers a questionnaire and submits it to us. Once we receive the answers, we measure how much knowledge can be retained by the programmers based on the answers on the questionnaire. We also introduce in this document two other ideas for future works: an Alpha-League algorithm for the synthesis of programmatic strategies, and a programmatic strategy combiner.

1.1 Objectives

The main goal of this research is to explore the program synthesis methods for two different contexts: Guide the search to find a dominant script, and measure the interpretability over programmatic strategies. To achieve these objectives, we propose the following:

- Develop and analyze heuristics capable of guiding local-search algorithms in the process of synthesizing programmatic strategies;

- Combine different algorithms and heuristics to reproduce the Alpha-League method to synthesize programmatic strategies;

- Use the potential of interpretability inherent to the program synthesis methods used by us here to allow knowledge transfer between computers and humans;

- Develop algorithms to combine programmatic strategies, allowing a single script to perform mixed-strategies.

1.2 Motivation

Synthesizing strong and robust programmatic strategies can help the games industry, providing smart scripts for games with low cost and flexibility, as reinforced by Summerville et al. (2017). These programmatic strategies can be interpretable and modifiable, characteristics that can be used to train professionals and adapt the behavior of the scripts according to the necessary specifications. To avoid the issues inherent to black-box solutions, Zhang et al. (2021) argue for the requirements of interpretability and explainability in the AI community. Therefore, one of our motivations lies in guaranteeing that the solutions synthesized by our methods satisfy those two requirements.

1.3 Document Structure

This document is organized as follows: Chapter 2 introduces the background and related work, providing basic definitions and related literature. Chapter 3 introduces

the problem related to programmatic strategies. In that chapter, we also describe the search space, local-search algorithm for synthesis, and heuristics for the search. Chapter 4 presents and describes our approach, Neighborhood Curriculum, and includes our motivation, along with the algorithm. Chapter 5 presents the domains used in this research and the empirical results. Chapter 6 is dedicated to the discussion of the results. Chapter 7 describes the future works with partial results. Chapter 8 provides the schedule for the next months of my doctoral studies. Chapter 9 enumerates contributions made so far. Finally, Chapter 10 concludes the document with necessary conclusions.

2 Background and Related Works

In this Chapter, we introduce necessary concepts to explain the algorithms and strategies described in this document.

2.1 Game Theory

The initial concept of game theory and the first important document was introduced by Neumann e Morgenstern (1947), and it has evolved and spread to different fields. Some important contributions come from John Nash, who introduced fundamental concepts like Nash Equilibrium (NASH, 1950). After 1970, the game theory concepts began to be used and influenced fields like economics, politics, business, and biology. In general, game theory handles situations where two or more agents make decisions, and the outcome is calculated from a combination of player's decisions (BONANNO, 2018).

The concept of game theory can be defined as a technique to analyze relations and situations between two or more agents. These situations between the agents do not depend on the action that one gets alone. The outcomes depends on the individual's actions and how these actions can influence each other. These actions are selected according to the expectations of the other agents and their strategies. Agents in such situations do not execute decisions in isolation. These agents consider the strategies of the other agents because the decisions are interdependently related, which is called a *strategic interdependence.* Here, the situations described are called *games of strategy*, and the individuals are referred to as players. Players need to follow strategies in the game with limited information about what the other individuals are doing (CARMICHAEL, 2008).

The *Prisoner's Dilemma* is one example of strategic games, in which two suspects in a crime are interrogated separately, and their answers determine who will be arrested or released. The investigation has sufficient evidence to condemn the suspects, but there are three possible outcomes to the situation based on the suspects' choices. The first possible outcome is when one acts as a snitch e other silent. The silent suspect is arrested for a major crime and the snitch is free. The second possible outcome is when both suspects keep silent. Both will spend one year in prison. The third possible outcome is when one betray and the other decide to be in silent. One will be freed and the other will get five years in the prison. The last outcome is when both decides to betray, and they get 3 year in the prison. This scenario can be represented as a matrix game, as we illustrated in table 1.

In Table 1, we have two players, suspects 1 and 2. The two actions that each player can perform are silent and snitch, being the rows to player 1 and the columns to player

		Suspect 2	
		Silent	Snitch
Suspect 1	Silent	(2,2)	(0,3)
	Snitch	(3,0)	(1,1)

Table 1 – A matrix representation of the Prisoner's Dilemma with the outcomes from each pair of actions. Based on (OSBORNE, 2004).

2. In Table 1, we have the outcome, or the player's payoff values. The first value is the outcome for player 1. For example, if player 1 selects silent and player 2 selects snitch, the payoff is (0,3), which means that player one gets zero, and player 2 gets 3, the high payoff. This simple game models an example where cooperation between the players has gains. However, the suspects have an incentive to select snitch and be freed from prison. But considering the risk of both players deciding to select snitch and get three years in prison, selecting silence sounds like the safest option.

2.1.1 Strategies

Strategy is defined as a player's action plan for a given game (CARMICHAEL, 2008). Considering the Prisoner's Dilemma, in a naive example, if Suspect 2 has an informant about which action Suspect 1 will take, Suspect 2's strategy could be: If Suspect 1 decides to be silent, Suspect 2 should choose snitch. Aside from the Prisoner's Dilemma, another example of strategy can be considering what to do depending on the weather. One simple strategy is, if the temperature is $+18^{\text{o}}$ degrees, we will go camping. However, if the weather is not sunny, we will have a dinner at home. If the temperature is less than $+18^{\text{o}}$ degrees, we will go to the home theater.

For a game $\mathcal{G}$, where $\mathcal{P} = \{i, -i\}$ represents the players, a strategy for player i is a function $\sigma_i(s) \to a$ that returns the action a for a specific state s. The set of all strategies for player i is equal to Σ_i .

2.1.2 Dominant Strategies

One notation to represent games is called normal form, or matrix form. This representation is the same as in the Prisoner's Dilemma (see Table 1). The normal form is used to compose combinations of strategies that each player can choose, and presents the payoff accordingly. To define such concepts, we use the normal form game introduced on Table 2.

Using Table 2 as the outcomes to select strategies for Player i, without any knowledge about the strategy of Player 2 is assuming, certainly that the strategy L1 is the best option that we can select. Regardless of the strategy that player 2 chooses, player 1 will get the highest score. For example, if Player 2 selects T1, Player 1 will receive 10

	Player 2	
	T1	T2
L1	$(10^\star,4^\dagger)$	$(1^\star,5^\dagger)$
L2	$(9,9^\dagger)$	$(0,3)$

Table 2 – A normal form game with the indications for best response strategies for each player, being $\star$ and $\dagger$ (superscript) the marks of best response. Based on (PETROSYAN; ZENKEVICH, 2016).

as payoff. This is a better payoff than L2. The same is valid for T2 and L2. Considering that situation, we can say that L1 is the best response for Player 2's strategies T1 and T2, and it is marked with $\star$ in Table 2 to indicate that dominance.

A strategy σ_i, for player i, is a best response $\mathcal{BR}_i(\sigma_{-i})$ for a strategy σ_{-i}, for player $-i$, if σ_i gives to player i the highest payoff given that player $-i$ has used σ_{-i}. Being valid the same concept to player $-i$.

However, to Player 2, there is more than one best response strategy: T1 is $\mathcal{BR}(L1)$, and T2 is $\mathcal{BR}(L2)$. In such situations, Player 2 will depend on the strategies of Player 1 to define his $\mathcal{BR}$. We have $\dagger$ in almost all the boxes on Table 2 to indicate the $\mathcal{BR}$ for Player 2 accordingly the strategy selected by Player 1. Using these examples, we can define the concepts of dominated and dominant (PETROSYAN; ZENKEVICH, 2016).

If a strategy σ never is the best response $\mathcal{BR}$ for any strategy that the opponent can choose, it is called dominated strategy. On the other hand, if σ is the best strategy for all opponent's strategies, it is called dominant.

Considering the game represented in table 2, L2 is considered a dominated strategy and L1 is a dominant strategy for player 1, because this player will select L1 regardless of the strategy selected by player2.

2.1.3 Two-Player Zero-sum games

A two-player zero-sum game can be formulated as $\mathcal{G} = (\mathcal{P}, \mathcal{X}, \mathcal{Y}, \mathcal{U})$, where:

- $\mathcal{G}$ represents the game.

- $\mathcal{P} = \{i, i\}$ is the set of players, being player i and $-i$ his opponent.

- $\mathcal{X}$ and $\mathcal{Y}$ are nonempty sets of strategies for player i and player $-i$, respectively.

- $\mathcal{U} : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ is a utility or payoff function where the value for player i, playing a strategy $x \in \mathcal{X}$, given a strategy $y \in \mathcal{X}$ for the opponent, is $\mathcal{U}_i(x, y)$. It is respectively for player $-i$, the payoff function is $-\mathcal{U}_{-i}(x, y)$.

Considering that the payoff function is represented by $\mathcal{U}_i(x, y) = -\mathcal{U}_{-i}(x, y)$, G is called zero-sum game. The definition of a two-player zero-sum game requires that G

has only two players, represented by $\mathcal{P}$ in the formulation, and be a zero-sum game (PETROSYAN; ZENKEVICH, 2016).

2.1.4   Iterated Best Response

The Iterated Best Response (IBR) method is a simple algorithm in which the players play against each other in a self-play fashion style until their strategies converge to a best response solution (HO; CAMERER; WEIGELT, 1998). The IBR is an optimization technique to approximate the best response for a strategy profile of the player against his opponent. The algorithm works interactively by selecting a player to optimize a strategy against a fixed opponent strategy (CHEN et al., 2017).

Algorithm 1 Self play with local search

Require: Game $\mathcal{G}$, DSL D, number of steps n.
Ensure: Script p for playing the game $\mathcal{G}$.
1: $p \leftarrow$ random-script(D)
2: **for** $k = 1$ to n **do**
3:      $p_{BR} \leftarrow$ local-search($\mathcal{G}, D, p$)
4:      **if** p_{BR} defeats p in $\mathcal{G}$ **then**
5:          $p \leftarrow p_{BR}$
6: **return** p

Figure 1 – A self-play algorithm with local search to synthesize programmatic strategies introduced by Mariño et al. (2021).

Mariño et al. (2021) used a self-play with local search to show how IBR can be used to synthesize programmatic strategies for MicroRTS domain, as shown in Figure 1.

2.1.5   Fictitious Play

Introduced by Brown (1951), Fictitious Play (FP) was initially presented as an algorithm and method to compute the value of a zero-sum game, and it was studied in depth by Robinson (1951), a fact that gave it the name of Brown-Robinson learning process. In this popular game theory model, players play a finite game repeatedly. In each confrontation, one needs to choose a best response against the opponent's strategy (HEINRICH; LANCTOT; SILVER, 2015). The process of play, at each iteration, a myopic pure best response against the distribution of opponents' strategies always converge to a Nash equilibrium after some point and rounds of matches (BERGER, 2007).

The fictitious play introduced by Brown (1951) and Robinson (1951) is defined as a pair of players $\mathcal{P} = \{i, -i\}$ being $(\mathcal{I}, -\mathcal{I})$ a representation of an $n \times m$ bimatrix game. Player i assumes the player row of the bimatrix and has access to the pure strategies $k \in \mathcal{N} = \{1, 2, ..., n\}$. Player $-i$ assumes the player column and uses the pure strategies

defined by $j \in \mathcal{M} = \{1, 2, ..., m\}$. $\mathcal{I}$ and $-\mathcal{I}$ are the $n \times m$ payoff matrices for players i and $-i$, once the first selects k, and the second selects j, the respective payoff for both players is a_{kj} and a_{jk}.

Player i's payoff when playing strategy k versus a mixed strategy y adopted by $-i$ is $(\mathcal{I}y)_k$. For player $-i$, the payoff is $(-\mathcal{I}^t x)_j$, where j is the pure strategy played by $-i$ against the mixed strategy x, and t (superscript) corresponding to the transpose of the matrix. $\mathcal{BR}_i()$ and $\mathcal{BR}_{-i}()$ stands for the pure strategy best response for i and $-i$, respectively. Finally, a strategy profile (x^*, y^*) is a Nash equilibrium if for all $k, x_k^* > 0$ and hint $k \in \mathcal{BR}_i(y*)$, and the analogous to $-i$. If x^* and y^* are pure strategies, there exists a pure Nash equilibrium. In this work, we used a mixed strategy profile, in an FP fashion method, to guide the synthesizer to find a pure strategy for the profile composed online by each iteration of the algorithms.

The FP introduced by Brown (1951) forces the player to find the best decision against the opponent's profile strategy. This aspect guarantees that the FP finds convergence in some games,such as two-person zero-sum games. However, it turns the convergence slow. Genugten (2000) introduced a Weakened Fictitious Play (WFP). In each iteration, WFP searches for a progressively better decision against the strategy profile of the opponent. The generalised weakened fictitious play introduced by Leslie e Collins (2006) is a WFP that allows ϵ-best responses and perturbed average strategy updates. The generalised weakened fictitious play converges in games that contain the fictitious play property Leslie e Collins (2006), as potential and two-player zero-sum games. We adopted the ϵ-best response in our implementation for FP by guaranteeing that in each iteration of the local-search algorithm there is almost a ϵ improvement ($\epsilon >= 0$) of the strategy when FP is applied.

2.1.6 Double Oracle

The Double Oracle (DO) algorithm was introduced by McMahan, Gordon e Blum (2003). The algorithm assumes two best response oracles, $\mathcal{O}_1$ and $\mathcal{O}_2$ for two players, row and column, respectively, being that game is a normal-form matrix. The algorithm starts with two sets of strategies, $\mathcal{R}$ for row player and $\mathcal{C}$ for column player. Both sets are initialized with a random row and a random column. At each iteration, the algorithm solves the matrix game considering that player row only uses strategies from $\mathcal{R}$ and player column strategies from $\mathcal{C}$. To solve the game, they use linear programming as a solver. After solving the game, the solver provides distributions r_i over $\mathcal{R}$, and c_i over $\mathcal{C}$, being i the current iteration. Assuming that the player column will play c_i, the player row computes an optimal pure strategy and adds this strategy to $\mathcal{R}$. The same is valid for player column. The algorithm stops when the optimal pure strategies for $\mathcal{R}$ and $\mathcal{C}$ are already inside of the sets. McMahan, Gordon e Blum (2003) proves that DO converges to
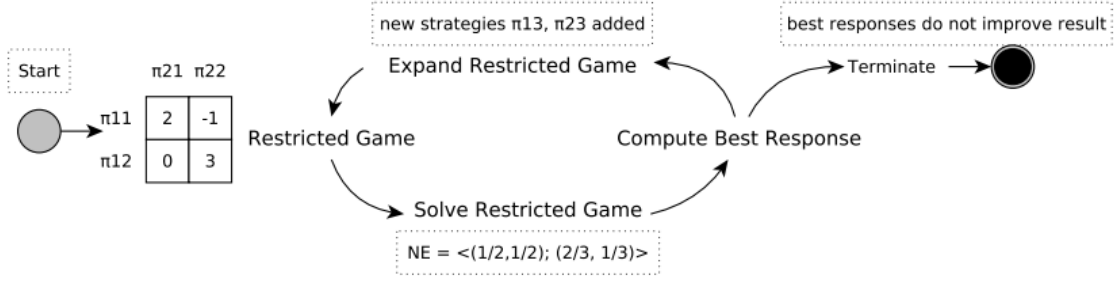
Figure 2 – Representation of the double oracle algorithm used to solve a normal-form game(BOSANSKÝ et al., 2016).

a minimax equilibrium.

The main goal of the double oracle algorithm consist on finding a solution for a normal-form game without needs to construct a solution for the entire game (BOSANSKÝ et al., 2016). Figure 2 shows how the double oracle scheme works for a normal-form game. Lanctot et al. (2017) introduced a generalization of the DO. The meta-game, or partial game, constituted by $\mathcal{R}$ and $\mathcal{C}$ sets of strategy, is composited by policies, while original DO the strategies are actions. We used the DO generalization in our project by considering that each program we synthesized is a full agent to play the game.

2.2 Program Synthesis

Pu et al. (2018) define the synthesis as a specific group of regression problems that uses a search algorithm to find a source-code program that satisfies constraints. These constraints are commonly input/output examples, and used by the synthesizer to ensure correctness. Another aspect of the synthesis is associated with the grammar used to compose the programs. This grammar is frequently represented as a Domain-Specific Language (DSL). The synthesis task needs to search in the space of solution represented by the grammar, respecting the relations and syntax expressed by it, while synthesize the program that satisfy the constraints. These two points, grammar, and constraints, are used by a large quantity of modern program synthesis projects (ALUR et al., 2013; GULWANI, 2011; PERELMAN et al., 2012).

The program synthesis task can receive as constraints a variety of entries, such as input/output examples, demonstrations, natural language, partial programs, assertions, and functions (GULWANI; POLOZOV; SINGH, 2017). Depending on the specifications of the users and their intentions, the program synthesis process can be classified as a deductive, inductive, enumerative, and natural language-based synthesizer. Programming by Example (PBE), one representation of inductive program synthesis, is a definition for synthesizing a program that uses a domain-specific language and tries to synthesize programs consistent with the input/output examples given as constraints (POLOZOV,

2017). Due to the importance of the PBE, we explain it in detail in Section 2.2.1.

The domain-specific language is a grammar that provides rules and notation for a specific domain, and it is commonly used by inductive program synthesis. The DSL contains particularities that can be used to produce programs and families of codes without demanding professional knowledge from the users. End-users have the advantage of using the DSL as a high abstraction of the environment, a fact that avoids the necessity of knowledge about a specific program language (KOSAR et al., 2008). In general, the DSL has a disadvantage because it requires time, expertise, and skilled professionals to codify. To avoid the cost of producing a DSL, some works use an entire programming language as the grammar. The DSL defines the search space and the programs that can be produced. If the DSL is expressive enough, the number of possible programs that can be synthesized is unlimited. The huge number of possibilities brings a trade-off between the complexity of finding, testing, and evaluating all programs synthesized during the search and the expressiveness of the search spaces (GULWANI, 2010). Considering how complex the search space is, the synthesis of programs is a challenging task, considered by some authors as one of the most central problems in the theory of programming (PNUELI; ROSNER, 1989). The DSL defines a language that the inductive program synthesizer needs to follow. For example, we can consider a grammar for arithmetic expressions, represented as a context free-grammar, that defines a DSL for arithmetic expressions.

$$S \rightarrow C \mid C - C \mid C + C$$
$$C \rightarrow (S) \mid C * C \mid N$$
$$N \rightarrow 0 \mid 1 \mid 2 \mid .. \mid 100$$

Normally, the program that is generated by the inductive program synthesis using the DSL follows a representation called Abstract Syntax Tree (AST). That tree represents different constructions, or derivation tree, composed by nodes that reflect the DSL's specification. For example a 8-3*4 calculation, we represent the derivation tree in Figure 3.

The program synthesis can be used in a large number of applications and different kinds of users. The applicability comes from helping in debugging codes(JACINDHA; ABISHEK; VASUKI, 2022), improving codes(GULWANI; POLOZOV; SINGH, 2017), engineers, and even teaching people(DAVID; KROENING, 2017). Here, we listed some possibilities:

- **Data Wrangling:** It is the task of applying cleaning, preparing, and/or transformations under data. That data comes in a raw format, or pre-structured format. The wrangling process transform the data in a new format, or more structured for-
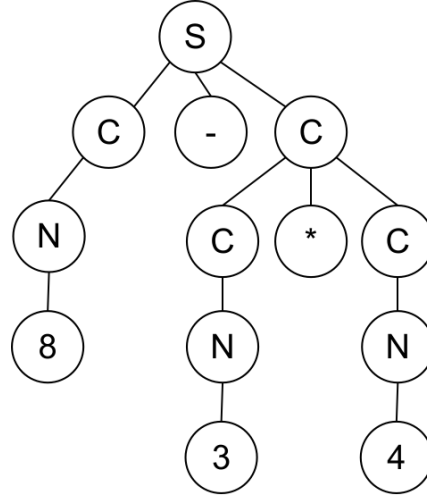
Figure 3 – Derivation tree for "8-3*4" example.

mat, being this format aligned with the interest of the end-user. Some examples of data wrangling are syntactic and semantic string transformations (GULWANI, 2016), information extraction under files, and many more.

- **Graphics:** Program synthesis can be used to draw, construct, and validate graphical objects for applications drawing applications. Cheema et al. (2014) introduced a framework that uses repetitive elements as examples, the program can predict the next element of the draw.

- **Code Repair:** This application of the synthesis helps programmers in improve codes by removing bugs, by receiving a bugging program and constraints, and producing a new program that guarantees the coverage of all constraints (D'ANTONI; SAMANTA; SINGH, 2016; NGUYEN et al., 2013).

Methods and techniques common in program synthesis use brute-force search (ALUR et al., 2013), constraint satisfaction (JHA et al., 2010), machine learning (BALOG et al., 2017), and hybrid systems that combine search with learned functions through a range of techniques (LIANG; JORDAN; KLEIN, 2010; MENON et al., 2013; MURALI; CHAUDHURI; JERMAINE, 2017).

2.2.1   Inductive Program Synthesis

The inductive program synthesis, or PBE, is one of the simplest forms of program synthesis (POLOZOV, 2017). In this sub-field of the program synthesis, the synthesizer needs to search in the language looking for programs that respect the inputs/outputs, provided as examples, to verify that the behavior of the solution respects these constraints. Table 3 shows a typical set of input/outputs examples:

Input	Output
José Carvalho	Carvalho, J.
André Pereira	Pereira, A.

Table 3 – Typical set of input/outputs examples for an inductive program synthesis algorithm.

In this PBE problem, the program needs to transform the entries in a typical last surname/abbreviation format, frequently used in citations. These examples exhibit unique properties that can be used by the search to satisfy the users' intent. However, considering the examples above, if a new and different entry with a new format is introduced, the program can break. For example, if the input is *Rubens de Oliveira Moraes*, the program synthesized may not work for this particular one because it has never experienced examples like that. This particular situation shows how important is the examples cover many situations, as suggested by Pu et al. (2018).

Gulwani (2016) uses Microsoft Excel spreadsheet as a domain to build a synthesis system that helps the end-user synthesize macros based on simple interactions with the users. In Flash Fill research, the algorithm is considered a real-time solution. It can run an entire interaction in approximately 0.1 ms, allowing the solution to be fast to the final user. The solution can recognize particular inputs that need more examples and requires them from the users. Another approach under inductive synthesis is the Counter-Example-Guided Inductive Synthesis (CEGIS) (ALUR et al., 2013). In this method, the authors used oracles to generate, automatically, counter-examples used for the search to derive the correct problem for a PBE problem.

All these works demand some form of supervision, which can be provided as a set of input/output examples (KITZELMANN, 2009). These resources lead learning, training, and synthesis process. The algorithms proposed in this work do not demand previous examples or data. Our synthesizer acts directly over the environment, in this case, a virtual one, while trying to harmonize a strategy to resolve a specific task. Tasks, in our research, are defined as programmatic strategies able to play an entire game in a domain (MARIÑO et al., 2021). We initially synthesized our solutions without any specification, and we improve upon our solutions by using the synthetic data produced by our algorithm. The quality of the program synthesized is verified by the game theory methods and our approach.

2.2.2 Interpretability

Investments in solutions that use Artificial Intelligence (AI), like Machine Learning, have grown considerably. According to the International Data Corporation (IDC), the investments in AI jumped from $24 billion U.S. dollars in 2018 to $85.3 billion in 2021,

and it will jump to \$204 billion by 2025 (IDC, 2021). AI has spread around the world and it is part of strategic solutions and technologies like medicine (SZOLOVITS, 2019; BRIGANTI; MOINE, 2020), robots and autonomous systems (HE et al., 2020; HAMET; TREMBLAY, 2017), AI-driven vehicles (MANOHARAN et al., 2019), and in the field of law (NUNEZ, 2017). The most significant AI's solutions use Machine Learning (ML) methods are based on deep learning. Unfortunately, even though these solutions reach superhuman skills (SILVER et al., 2016; VINYALS et al., 2019a), they lack clarity and transparency. In general, these solutions are trained to archive high precision in decision-making, but they are characterized as black box solutions (CARVALHO; PEREIRA; CARDOSO, 2019).

It is a fact that AI improves human capabilities when applied in healthcare, criminal justice, monetary decisions, and other situations. The knowledge of the AI solution is, in general, part of his internal logic and it keeps hidden from experts to verify, interpret and these decisions (ADADI; BERRADA, 2018). This particular issue rises a discussion about the importance of the interpretability and explainability of these solutions. The Explainable Artificial Intelligence (XAI), initially used by Lent, Fisher e Mancuso (2004), was introduced as the ability of a system to explain its choices in a game platform. Currently, XAI has become an area of study and interest for those who want to guarantee that solution can be validated, studied and verified (DOŠILOVIĆ; BRČIĆ; HLUPIĆ, 2018).

The field of program synthesis emerges as an opportunity to provide explainable, interpretability, and transparency for AI solutions, allowing experts, professionals, and society to understand and learn with the knowledge codified by them. Zhang et al. (2021) show how interpretability allows humans to interact with the application and they can provide feedback and corrections. This feedback improves the synthesizer and increases the quality of the program synthesized. Furthermore, program synthesis can be used to explain other AI methods. Verma et al. (2018) introduced a framework called Programmatically Interpretable Reinforcement Learning (PIRL), which produces agent policies using a DSL, and might provide interpretability and verifiability for the policies synthesized. Bastani, Inala e Solar-Lezama (2022) expands the PIRL ideas by exploring how the program synthesis can be used to mimic DRL's solutions. In my project, we use a DSL that allows the synthesized program explainability and modification by the users.

2.3   Local Search

Local Search algorithms work by starting from a current solution. Starting from this solution, the algorithm generates neighbor solutions. The algorithm evaluates each neighbor solution moving from the next best neighbor solution, if it exists, and it follows the search by repeating the same steps. In general, the local search algorithms do not retain the neighbor information or the path the algorithms used to reach the best solution. This

aspect brings one advantage to local search, which is the usage of little memory, or a constant quantity.
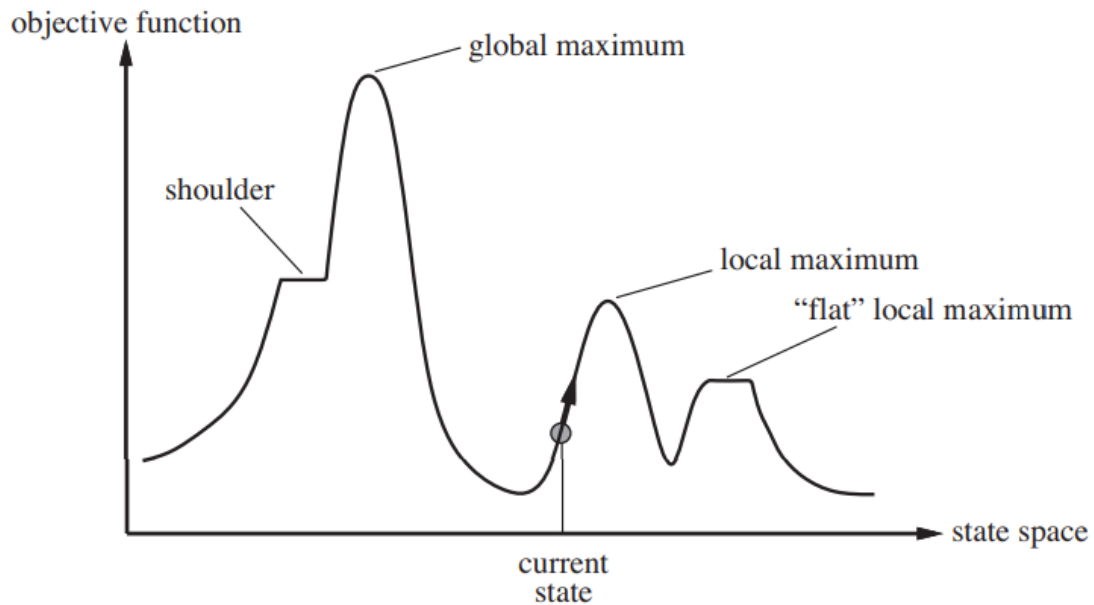
Figure 4 – Example of a state-space landscape composed of elevations that represent the objective function. Figure from (NORVIG; RUSSELL, 2004)

.

To illustrate how local search works, we consider the state-space landscape Figure 4. The local search algorithm explores the space of solutions represented by this one-dimensional landscape. The algorithm tries to find the global maximum solution or the global minimum solution according to the objective. The location is defined by the state and the elevation is defined by the value of a function. This function can be a heuristic or an objective function. The function defines whether the algorithm is looking for a maximum or minimum solution. If a local search algorithm always finds a goal, it is called complete. If the local search algorithm always finds a global maximum/minimum solution, it is called optimal.

2.3.1 Hill-Climbing

The Hill-Climbing (HC) algorithm is a local search algorithm that moves the search toward the highest value/elevation. At each step of the search, the algorithm tries to find the highest neighbor solution from an initial solution. Each new best neighbor found moves the search in a direction toward reaching the peak of the closest mountain. This representation of space-state landscape is shown by Figure 4. The hill-climbing algorithm is presented in Figure 5 as a simple loop that continuously moves the current solution in the direction of the highest value of the hill. The search stops when it reaches a peak or plateau in the landscape.

```
1 def Hill-Climbing(problem):
2     current = problem.initial_state()
3     while True:
4         neighbor = best_neighbor(current, problem)
5         if neighbor.value() <= current.value():
6             return current
7         current = neighbor
```

Figure 5 – Hill-Climbing Search Algorithm. At each iteration of the search, the current solution is replaced by the best neighbor until it stops.

Hill-Climbing is also called greedy local search because it keeps moving just in the direction of the best local solution. This greediness can perform well in some problems because HC can easily progress in the direction of a good state, even if the algorithm starts from a bad point. However, the HC algorithm gets stuck in the search if the landscape contains local maxima, ridges, or plateaus. To avoid these problems, some variations of the hill-climbing search were introduced: Stochastic hill-climbing, first-choice hill-climbing, and random-restart hill climbing. The HC algorithm is incomplete if it never walks a "downhill" in the search, it means never perform restarts or random walk. It is said incomplete considering the HC might stuck in a local maximum (RUSSELL; NORVIG, 2002). On the other hand, if the algorithm performs a pure random walk, moving from one successor to another by random choices, it is complete but inefficient. The next algorithm, Simulate Annealing, tries to combine the best of both solutions: Hill-climbing and random walk.

2.3.2   Simulated Annealing

Simulated Annealing (SA) is an algorithm that combines hill-climbing with random walks. SA combines the efficiency of the HC with the completeness of the random walks method (RUSSELL; NORVIG, 2002). The term annealing comes from metallurgy and it is used to define the process of tempering a metal. Initially the material gets high temperature and it is placed to cool down slowly. SA follows the same idea: When the temperature parameter in the algorithm is high, it accepts the solution to move downhill and performs some random walks in the landscape of states. As the temperature parameter goes down, the SA reduces the random walk movements, being similar an HC algorithm. The SA algorithm is shown in Figure 6.

The algorithm receives the variables problem and schedule as input. The problem defines the space of solutions. The schedule is used as a function to change the temperature parameter according to the function iterations/time (see Line 1). The loop in line 3 is used to represent the time (the number of iterations t) in the problem, and it is similar to the loop in the HC algorithm (see Figure 5). The input variable schedule defines

```
1 def Simulated-Annealing(problem,schedule):
2     current = problem.initial_state()
3     for t in range(1,inf):
4         T = schedule(t)
5         if T == 0:
6             return current
7         neighbor = get_random_neighbor(current, problem)
8         E = neighbor.value()−current.value()
9         if E > 0:
10            current = neighbor
11        elif exp^{E/T} >= rand.random():
12            current = neighbor
```

Figure 6 – Simulated Annealing is a stochastic version of the hill-climbing algorithm where, based on the temperature, some downhills are made. These downhills happen with more probability when the search temperature is high (at the beginning of the search) and less frequently at the end when the temperature is cold. The schedule function is used to define the temperature T as a variant of time.

the temperature T as a function of time represented by t (see Line 4). The algorithm stops when the temperature reaches zero (see Lines 5-6). Then the algorithm moves by generating random neighbors from the current solution (see Line 7). The neighbor is always accepted if the solution is better them the current solution (see Line 9). However, the algorithm can accept worse solutions (the downhill trend) with some probability (see Line 11). The probability is influenced by the temperature and the quality of the neighbor. If the temperature is low, the chances of accepting the downhill trend are low. The same happens if the solution quality, represented by E, in line 9, is poor.

2.4   Related Works

In this section, we present several relevant previous works associated with our research, which rely on game theory, program synthesis, programmatic strategies, programmatic reinforcement learning, automatic scripting, and planning in zero-sum domains with real-time constraints.

Mariño et al. (2021) define a novel algorithm to reduce the DSL D' used in the experiments by applying the set cover algorithm under sequences of state-actions pairs, called traces. These traces are generated by an agent that is able to play an entire game from the state s_{init} until s_{end}. The algorithm, called Domain-Specific Language Simplifier (Lasi), acts by selecting a subset of rules $D"$ from D' that is more restricted than the original. Lasi guarantees that the rules in $D"$ are sufficient to reproduce the same traces provided as input to the algorithm. After defining $D"$, a local search algorithm is used to

synthesize a script for different maps in the MicroRTS. An iterated best response algorithm is used by the local search to synthesize programmatic strategies. The scripts produced by this project achieved high scores in matches against search algorithms. In a qualitative experiment, Mariño et al. (2021) show that the algorithm can encode scripts similar to strategies written by programmers, in some cases better by adding improvements in the strategies that are not easily discoverable to humans. We expand this work by evaluating fictitious play and double oracle as heuristics for the local search, and we propose a new heuristic that help local search algorithms in the synthesis task. Lasi can be combined with our novel to reduce the DSL used and helps the local search algorithm. However, our heuristic can be used in any two-player virtual environment with any DSL. Lasi can not be used in a virtual environment which the DSL's simplification is not possible, like in games that all actions can be required.

Medeiros, Aleixo e Lelis (2022) introduced learning sketches based on behavioral cloning that allows local-search algorithms to effectively synthesize strategies for playing Can't Stop and MicroRTS games. They used sketches to reduce the computational cost of learning the basics fragments from scratch. They introduced an UCT and a Simulated Annealing method for synthesis of strategies and cloning of the strategies in a sketches fashion model. The sketch generation introduced by them is innovative by defining the sketch with a small quantity of samples, a fact that turns the neural model techniques to generate sketches impractical. These algorithms differ from the work presented in this research because they use only Iterated Best Response as a heuristic to guide the search methods, while we investigated the potential of Fictitious Play, Double Oracle, and our method to guide the synthesis of programmatic strategies. These methods can be combined with the novel introduced by Medeiros, Aleixo e Lelis (2022) to boost the synthesis process.

Mariño e Toledo (2022) introduced a novel based on genetic programming to the synthesis of computer programs (scripts) for zero-sum games by using population-based self-play evaluation to improve the strategies generated by a genetic algorithm. Gesy, as called by the authors, is a genetic programming approach that evolves initial populations of random scripts. To define the best strategies, Gesy runs five times in each map. The best individual for each test in Gesy's test is collected and evaluated against the opponents. These opponents are composed of search algorithms and Gesy's instances. This work shows that the final scripts are interpretable and competitive against search-based methods. Furthermore, the contributions provided by our work might be combined with Gesy to improve the results in an Alpha League (DEEPMIND, 2019) method.

In Programmatically Reinforcement Learning (PRL), policies are characterized as computer programs (BASTANI; PU; SOLAR-LEZAMA, 2018). These programs can be interpreted and formally verified. PRL's methods have been applied in previous works to mimic a policy codified by a neural network. (VERMA et al., 2018; VERMA et al.,

2019). The policies generated by PRL's methods are commonly represented as high-level solutions defined by a domain-specific programming language, an aspect that allows interpretability. Instead of learning policies, as is commonly done in PRL's works, we propose algorithms that use heuristics based on game theory. These heuristics guide the search in a virtual environment (game) and synthesize programmatic strategies. Our algorithms can interact with the game to evaluate the solutions, while navigating through the search space. The strategies produced by our solutions are high-level solutions with interpretability and verifiable characteristics.

Dynamic Scripting (DS) is a novel learning technique for game AI, which uses reinforcement learning to synthesize scripts for role-playing and fights games (MAJCHRZAK; QUADFLIEG; RUDOLPH, 2015; SPRONCK; SPRINKHUIZEN-KUYPER; POSTMA, 2004). This technique uses an adaptive rulebase of previous rules codified by experts as a database to compose strategies for games. The rulebase splits the rules, according to the game, agents, types, classes, or classification. These rules have a weight in the rulebase. This weight is used to define the probability to selected the rules for a character's strategy. The weight is updated online after each match and guarantees adaptability for the agents in-game, corresponding to the player's skill. Variations of the DS' methods were introduced to zero-sum real-time strategy games (RTS) (PONSEN; SPRONCK, 2004; DAHLBOM; NIKLASSON, 2006), where the synthesis process is limited by a fixed number of clauses if-then. We proposed a more robust solution by adopting an expressive DSL with nested if-then and for clauses, allowing the strategies to be abstract and robust.

Benbassat et al. (BENBASSAT; SIPPER, 2011; BENBASSAT; SIPPER, 2012) introduced a novel algorithm based on genetics programming to synthesize computer programs for zero-sum tabletop games. The programs synthesized by that novel were used to evaluate and prune functions. This method did not generate full programs able to play the entire game, as we proposed. Algorithms that generated evaluation function can be combined with our algorithms to increase the quality of the heuristic used by our solutions in the synthesis task to guide the search.

Program synthesis have been applied in collaborative games. Canaan et al. (2018) used evolutionary techniques to produce strategies to play a cooperative game called Hanabi. By searching in sequences of pre-coded blocks of if-then rules, the algorithm was able to determine the best sequence of rules and set it as a strategy. Freitas, Souza e Bernardino (2018) defined grammar-based genetic programming to evolve controllers to play the popular Mario AI game. Butler, Torlak e Popović (2017) introduced a method to synthesize strategies able to resolve nonograms' puzzles. The algorithm introduced in (FREITAS; SOUZA; BERNARDINO, 2018; BUTLER; TORLAK; POPOVIĆ, 2017) differ from our methods to be applied in the context of a unique player. Our methods are designed to generate strategies for two-player zero-sum games.

3 Problem Definition

In this Chapter, we define the synthesis of scripts as a program synthesis task for two-player zero-sum games. After, we introduced the search space conditioned to a DSL. Also, we described how we based on variations of the iterated best response, fictitious play, and double oracle to implement heuristics. These were used to guide algorithms of local search, such as Hill-Climbing, and Simulated Annealing, in the process of scripts synthesis.

3.1 Programmatic Strategies Synthesis Formalization

We define the synthesis of programmatic strategies formally assuming a zero-sum game $\mathcal{G}$, where $\mathcal{P} = \{i, -i\}$ represents the players for the game $\mathcal{G}$. The set of states is represented by $\mathcal{S}$, being s_{init} the initial state of $\mathcal{G}$. Each player can perform a set of legal actions $\mathcal{A}_i(s_t)$ for a state s_t in time t. A decision point is a state $s_t \in \mathcal{S}$ where player i can perform an action $a_i \in \mathcal{A}_i(s_t)$. During the game, each decision point requires player i to take an action a_i. Player i provides the action a by playing a player strategy. This strategy is a function $\sigma_i : S \to \mathcal{A}_i$ that returns the action a for a specific s_t. We define a script as a function $\psi(s_t)$ that maps a state s_t to a specific action a for player i in a decision point, being that action legal and available to be executed in the game $\mathcal{G}$. $\psi(s_t)$ is a programmatic strategy that encoding σ in a computer program style. The utility function $\mathcal{U}_{\mathcal{P}}(s_t)$ returns the value of the game for a specific player in $\mathcal{P}$ for a state s_t. Considering that the game $\mathcal{G}$ is a zero-sum environment, we have that $\mathcal{U}_i(s_t) = \mathcal{U}_{-i}(s_t)$. For s_t, the value of the game is denoted by $\mathcal{U}(s_t, \psi_i, \psi_{-i})$ when player i follows the script ψ_i and player $-i$, ψ_{-i}, respectively.

3.2 Defining the Search Space

We use a DSL D to define the search space of programmatic strategies. The set of scripts that can be synthesized by using D is limited by $[\![D]\!]$. One of our tasks in this project is synthesize a script ψ_i that maximizes the utility function for player i against player $-i$, when $-i$ plays the script ψ_{-i}. We formulate the problem of program synthesis as,

$$\max_{\psi_i \in [\![D]\!]} \min_{\psi_{-i} \in [\![D]\!]} \mathcal{U}(s_{init}, \psi_i, \psi_{-i}). \tag{3.1}$$

The strategies σ encoded by the scripts $\psi_{\mathcal{P}}$ for each player in $\mathcal{P}$ able to solve

the Equation 3.1 defines a Nash equilibrium profile when considering just scripts that are limited to the set $[\![D]\!]$. Trying to synthesize a strategy that can be a Nash profile might be troublesome, particularly when D is abstract enough to produce a large set of scripts for the set $[\![D]\!]$. In the meantime, we compute a programmatic strategy that can be a pure dominant strategy, or an approximated Best Response ($\mathcal{BR}$), by using the approaches iterated best response (see 2.1.4), fictitious play (see 2.1.5), and double oracle (see 2.1.6).

3.3 Hill-Climbing for Program Synthesis

In Section 2.3.1, we defined a version of the Hill-Climbing algorithm to synthesize programmatic strategies, as described in Figure 7. The HC is used to approximate a programmatic best response to a target heuristic (see Section 3.5). HC starts with an initial program ψ_{best}, that can be generated randomly or can be the best script produced by another iteration of HC. Also, HC receives an instance of the game $\mathcal{G}$, a DSL D, an heuristic Ψ, and the value n that represent the number of neighbors that will be visited to define the best one.

```
1 def HillClimbingPS(DSL D, Game G, script ψ_best, heuristic Ψ,
    neighbors n):
2     best_score = Ψ.evaluate(G, ψ_best)
3     while stop_condition(score, Ψ) and has_time():
4         best_neighbor = ψ_best.clone()
5         neigh_score = −inf
6         for i in range(n):
7             neighbor = best_neighbor.next_neighbor(D)
8             score = = Ψ.evaluate(G, neighbor)
9             if score > neigh_score:
10                best_neighbor = neighbor.clone()
11                neigh_score = score
12         if neigh_score > best_score:
13             best_score = neigh_score
14             ψ_best = best_neighbor
15     return ψ_best
```

Figure 7 – Hill-Climbing Algorithm for Program Synthesis.

The initial script ψ_{best} defines the best solution until now, and if the HC does not find a better script, it returns the same initial instance. The HC iterates until it exhausts the time or reaches a stop condition according to the score and the heuristic Ψ. For each iteration of the outer loop (line 3), the inner *for* evaluates n neighbors, generating the next neighbor based on the script defined as *best_neighbor*. After the entire execution of the inner loop, if a better script is found, the algorithm updates ψ_{best}. The method *next_neighbor* is defined accordingly the DSL D. It can be a small modification in the

terminal nodes of the AST (see section 2.2). Otherwise, the modification can be deep, which will modify the entire AST, depending on the DSL and if the game imposes some limitations. We go deeper on the DSLs used for each domain in Section 5.

3.4   Simulated Annealing for Program Synthesis

The Simulated Annealing (SA) is a probabilistic algorithm (see Section 2.3.2) based on a local search to approximate a global function. It was first used to synthesize programmatic strategies by Medeiros, Aleixo e Lelis (2022) in the test-bed MicroRTS and Can't Stop. We introduced the SA's version in Figure 8.

```
1 def SimulatedAnnealingPS(DSL D, Game 𝒢, script ψ_best, heuristic Ψ,
     α, β):
2     best_score = Ψ.evaluate(𝒢, ψ_best)
3     local_solution = ψ_best.clone()
4     local_score = best_score
5     while stop_condition(best_score, Ψ) and has_time():
6         T = schedule(α, T)
7             if T == 0:
8                 return ψ_best
9         solution = local_solution.next_solution(D)
10        score = Ψ.evaluate(𝒢, solution)
11      if accept(score, local_score, T, β):
12          local_solution = solution
13          local_score = score
14        if score > best_score:
15            best_score = local_score
16          ψ_best = local_solution
17      return ψ_best
```

Figure 8 – Simulated Annealing Algorithm for Program Synthesis.

The SA receives the DSL D, a Game $\mathcal{G}$, the initial script ψ_{best}, the heuristic Ψ and two variables, α, and β as input. α, and β are used as temperature controller and for the accept function, respectively. SA uses the temperature T to define when the algorithm stops (see Lines 6 to 8) and to consider if a solution will be accepted or not as the next solution (see lines 11 to 13). The *accept* function control the greediness of the algorithm. SA behaves like a random walk when the temperature T is high, which allows movements for solutions that are not always better than the current solution. When the temperature T is low, SA behaves more like a hill-climbing. We define the *accept* function as

$$\min\left(1, \exp\left(\frac{\beta \cdot (\kappa' - \kappa)}{T}\right)\right).$$

Here, T is defined by the schedule in Line 6. κ is the score of the local solution, and κ' it the score of the next solution generated by the mutation of the local solution. The heuristic Ψ calculates these scores in Lines 2 and 10. If $\kappa' \geq \kappa$, the new solution will be accepted with a probability of 1.0. If not, the probability of the new solution be accepted depends on temperature T and the parameter β.

3.5   Heuristics

The heuristics $\Psi(\mathcal{G}, \psi)$ are utility functions in the local search algorithms to help the search in the way to find a better approximated programmatic strategic best response. To perform the evaluation, the heuristics receive the game $\mathcal{G}$ and the script ψ which represents the programmatic strategy to be evaluated. We introduce three heuristics (baselines) based on the iterated best response, fictitious play, and double oracle.

3.5.1   Iterated Best Response

The heuristic, called Iterated Best Response, retains just one strategy used to define the payoff of the script introduced as input. The algorithm returns $\mathcal{U}(s_t, \psi_i, \psi_{-i})$. s_t is the initial state of the game $\mathcal{G}$, ψ_i is the neighbor solution , and ψ_{-i} the script that represents the opponent which the algorithm is trying to find a $\mathcal{BR}$.

3.5.2   Simple Cumulative Fictitious Play

The Simple Cumulative Fictitious Play (SCFP) is an heuristic based on the FP method in a simple cumulative form. To define the payoff of the strategy ψ_i, SCFP performs an individual evaluation against all the opponents that represent a mixed strategy profile. Figure 9 shows how SCFP performs the evaluation.

```
1 class SCFP:
2     list Ψ
3     def evaluation(Game 𝒢, script ψ):
4         payoff = 0
5         for ψₜ in Ψ:
6             payoff += 𝒰(𝒢.sₜ, ψ, ψₜ)
7         return payoff/len(Ψ)
```

Figure 9 – A pseudo-code that represents the simple cumulative fictitious play.

The SCFP works accumulating each solution in Ψ list. The solution in our context are represented by the scripts found in each iteration of the local search algorithm. Using the stored scripts, the SCFP provides a signal to the search for better spots in the search space. Even the weak strategies, discovered in the beginning of the local search, can

helps the synthesis by providing small information about the correct spot in the search space. This is better than keep just the best strategy found during the search, as the IBR heuristic does. However, the signal provided by all the solution stored is a computational costly.

3.5.3   Script-Space Double Oracle

The Script-Space Double Oracle (SSDO) heuristic is based on the original DO (MCMAHAN; GORDON; BLUM, 2003) and policy-space response oracles (PSRO) (LANC-TOT et al., 2017) (see Section 2.1.6). Instead of using policies to replace the action as PSRO, the SSDO uses the scripts as strategies. Figure 10 shows how SSDO is implemented as an heuristic.

```
1  class SSDO:
2      list Ψ^{meta}
3      def resolve_meta_game(Ψ, payoffs):
4          Ψ^{meta}.clear()
5          ρ = LPSolver(payoffs).solve()
6          Ψ^{meta} = get_scripts(ρ, Ψ)
7      def evaluation(Game 𝒢, script ψ):
8          for ψ_t in Ψ^{meta}:
9              payoff += 𝒰(𝒢.s_t, ψ, ψ_t)
10         return payoff/len(Ψ^{meta})
```

Figure 10 – A pseudo-code that represents the script-space double oracle heuristic.

Before each iteration of the local search, the heuristic SSDO needs to define the Ψ^{meta} profile that will be used to evaluate the current player. This setting is executed by the function **resolve_meta_game**, which receives two lists of scripts Ψ and a payoff matrix. Ψ is composed of all the strategies discovered for a specific player after each iteration of the local search. The *payoffs* is a matrix of overcome, where the rows are composed by strategies Ψ_i, and columns by Ψ_{-i}. Ψ^{meta} is the list of strategies that represents the mixed profile of strategies. If $\Psi_i = \Psi_{-i}$, it is a symmetric game. Using the payoff, the SSDO uses a Linear Programming (LP) solver (see Line 6) to solve the payoff matrix, resulting in a list of probabilities, ρ. These probabilities are used to define the opponents Ψ^{meta}. If a strategy has a probability > 0.0, it is included in the Ψ^{meta} list for the specific player (see Lines 9 and 10). The evaluation method (see Line 8) returns the payoff by evaluating all the Ψ^{meta} opponents selected for the current player.

3.6   Limitations

Each heuristic has some limitations, which can slow the search. The Iterated Best Response (IBR) is a myopic method that focuses on just one strategy, and it tries to find

$\mathcal{BR}$ for this one. To illustrate this problem, let's suppose four strategies A, B, C and D, and the following situation: A is the $\mathcal{BR}$ for B. B is the $\mathcal{BR}$ for C. C is the $\mathcal{BR}$ for A, and D is the $\mathcal{BR}$ for A, B, C. The IBR can comes to a loop without converge to a D, by keeping looking just for one single strategy at time.

The SCFP method is able to solve the example above. To reach it, the method keeps the strategies A, B, and C as part of the set of solution, allowing the search to find the solution D that is a best response for the set. However, if the number of strategies synthesized during the search is great, SCFP is a costly heuristic, and slows down the search.

The Double Oracle (DO) method offers a better solution than SP and FP by resolving the game via LP and selecting the strongest strategies of the matrix game. Between the heuristics, IBR and SCFP, the SSDO is able to select only the best strategy (if it is dominant) and perform like IBR. Or, in the worst case, SSDO will behave as SCFP. However, in the average case, SSDO will select a small group of strategies to guide the search. However, depending on the strategies in the matrix, the SSDO can loses important strategies that can help the search to get a fast convergence. This aspect will be explained in Section 4.1.

4 Neighborhood Curriculum

The Neighborhood Curriculum (NC) is supported by the information provided neighborhood's solutions evaluated during the local search. These information collected is used to define the group of opponents necessary to lead the synthesis in the process to generate an approximated script best response. We have been organizing the content of this approach in a paper [1], and we intent to submit it for Journal of Artificial Intelligence Research (A2) or IEEE Transactions on Games (A3).

4.1 Motivation

In Chapter 3, Section 6, we introduced some limitations to the IBR and SCFP. SSDO can behave as IBR or SCFP by solving a meta game, which allows the algorithm to select a specific number of strategies. In the best case, SSDO will select the dominance strategy, and it will behave as IBR. In the worst case, SSDO needs all the strategies, and it will behave as SCFP. In the average case, SSDO selects a small group of strategies that is sufficient to provide enough signal to help the synthesizer to find an approximated best response. However, SSDO can lose some crucial strategies when resolving the meta game using LP. This lack of selection impact directly on the next group of strategies chosen, and it quality to guide the search.

To explain this SSDO limitation, we introduce a scenario that represents a time when SSDO selects a group of strategies that leave out some crucial information for the search. Let's call the game *Poachers & Rangers*. In this game, Poachers need to find one free gate in the park to get in to steeling the resources. The government ordered his Rangers to protect the park's entrance. When, Poachers try to invade the park, the Rangers must decide which gate to protect. Table 4 shows 6 iterations of the algorithm in the game Poachers and Rangers scenario, in which each line shows the gates. Each gate on the park is labeled with a number, that the Poachers used to invade the park and which gates the Rangers were protecting at that time. Considering the gates invaded, the Rangers decide on the gates that they will defend. Each strategy receives a label c (for Poachers) and s (for Rangers) to facilitate identification.

Using the game and the strategies labeled in the Table 4, we introduce a payoff matrix, in which the strategies used by the Rangers are the rows, and the Poachers' strategies are the columns in Table 5. For example, if the Poachers choose the strategy $c2$ and Rangers $s3$, the payoff is 1 for Rangers, and -1 for Poachers. It happens because $s3$ cover all the gates on the strategy $c2$.

---

[1] https://pt.overleaf.com/read/bxxpprjsvxnn

		Gates [Invaded/Protected]	
Attempts	**Poachers**	**Rangers**	
1	c1=[4]	s1=[14]	
2	c2=[14, 8, 5]	s2=[4, 5, 6, 7, 13, 14, 15]	
3	c3=[14, 7, 1]	s3=[2, 3, 4, 5, 6, 7, 8, 12, 13, 14, 15]	
4	c4=[14, 10]	s4=[1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15]	
5	c5=[14, 10, 1]	s5=[2, 3, 4, 5, 6, 7, 8, 10, 12, 13, 14, 15]	
6	c6=[14, 9]	s6=[1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 13, 14, 15]	

Table 4 – A table that shows 6 attempts of the algorithm on Poachers and Rangers. Each row shows the numerate gate used by Poachers and each numerate gate protected by the Ranger.

		Poachers					
	Strategies	c1	c2	c3	c4	c5	c6
	s1	-1	-1	-1	-1	-1	-1
	s2	1	-1	-1	-1	-1	-1
Rangers	s3	1	1	-1	-1	-1	-1
	s4	1	1	1	-1	-1	1
	s5	1	1	-1	1	-1	-1
	s6	1	1	1	1	1	-1

Table 5 – Payoff matrix for the strategies on table 4.

We solve the Table 5, using an LP solver, as DO perform. This gave us the probabilities on Table 6. Using these probabilities, the mixed strategy for player Poachers is the set of strategies c4=[14, 10], c5=[14, 10, 1], and c6=[14, 9].

	Strategy	Probability
	c1	0.00
	c2	0.00
Player	c3	0.00
Poachers	c4	0.23
	c5	0.27
	c6	0.50

Table 6 – Probabilities of strategies for player Poachers. Values rounded to two houses to simplify.

Considering these examples, we can see the limitation of the SSDO approach lies in the loss of information that comes from strategies c2 and c3: gates 5, 7, and 8 are not more part of the mixed profile selected. However, these gates 5, 7, and 8 exist in the original set of strategies used in the past. This forgetfulness happens because the majority group of Rangers' strategies has already covered these gates.

The limitation in the SSDO approach, considering those examples, is the loss of

information that comes from the strategy c2 and c3: gates 5, 7, and 8 are not more part of the mixed profile selected. However, these gates exist in the original set of strategies, and it happens because the majority group of soldier strategies covered these gates.

4.2 Formalization

Neighborhood Curriculum (NC) uses the neighborhood's solutions evaluated during HC to define the group of opponents necessary to guide the synthesis of pure dominant scripts or an approximated script best response for the heuristic h. NC keeps a set C of strategies (i.e., scripts) composed of the current best strategy b_i and a subset of previous strategies selected according to their win rates against b_i. NC also keeps a set C_f of all strategies b_i returned by HC at each iteration. We start the search with $C = \{b_0\}$, where b_0 is a random solution. During the first HC iteration, we generate neighbor solutions until we find one that wins against b_0, which becomes b_1. We then include b_1 in C, forming $C = \{b_0, b_1\}$. At the following iterations $i > 1$, we generate neighbor solutions $N^i = \{n_0^i, n_1^i, \ldots, n_T^i\}$ until we find one b_i that wins against all solutions in C.

While generating the neighbor solutions n_t^i, we keep statistics of each match $b \times n_t^i$, for all $b \in C$. These statistics decide whether or not b will stay in the curriculum. To produce the game statistics for a neighbor solution n, we compute the utility value $r_{i-1} = \mathcal{U}(s_t, n, b_{i-1})$, where b_{i-1} is the best solution of the last HC iteration $i-1$. Next, for each $b \in C$, we compute the utility value $r_c = \mathcal{U}(s_t, n, b)$. If n wins against b_{i-1} according to r_{i-1} and losses to b according to r_c, we store one (1) for the entry (n, b), otherwise, zero (0). For example, consider the game statistics in Table 7.

A. Game Statistics ($i = 2$)		
	b_0	b_1
n_0	0	0
n_1	0	0
n_2	1	0
n_3	1	0

B. Unique Statistics	
$\{b_0 \quad b_1\}$	v
$\{0 \quad 0\}$	2
$\{1 \quad 0\}$	2

Table 7 – Example of game statistics and a their unique entries.

The entry (n_2, b_0) is equal to 1 because n_2 won against b_1 and lost against b_0. On the other hand, the entry (n_2, b_1) is equal to 0 because n_2 lost against b_1. After computing these statistics for N, the HC iteration i is complete and we count the number of unique entries in a table T, as shown in Table 7 (B). After a HC iteration i, we evaluate b_i against all strategies $b_f \in C_f$. If we find a strategy b_f that defeats b_i, we add b_f in C and b_i in C_f, and restart the search. Otherwise, we perform a greedy search on T to reduce C.

The greedy search starts calculating the score K_{b_j} for each strategy b_j in C. The score is calculated by the entries 1 and the values v on T, accordingly the Equation 4.1:

$$K_{b_j} = \sum_{j=0}^{|C|} \sum_{t=0}^{|T|} T_t^{b_j} * T_t.v \qquad (4.1)$$

For example, the $K_{b_0} = 2$ and $K_{b_1} = 0$ for Table 7 B. The search continues by selecting the strategy b_j with the highest value on K_{b_j}. After that, all entries 1 on T for b_j is removed from T, and the greedy search is restarted. The search stops when there is no value in K_{b_j} bigger than zero. All the b_j selected during the search compose the new C.

The NC is able to booster the synthesis by tracking crucial strategies that, even weak, can booster the search on the direction of a dominant programmatic strategy. Collecting meta-data provided by the neighbor solutions evaluation, the NC can fix the SSDO's problem because its methodology recognize gaps of necessary strategies.

4.3   Algorithm Description

The SSDO approach suffers from the lack of information, as shown in section 4.1. Here, we introduces the Neighborhood Curriculum (NC) as a novel based on the idea of a curriculum, which lies on first: the best strategy will be always part of the curriculum (main goal). Second, we add other strategies which can be seen as stepping stones to boost the search. To decide which strategies will be selected, the Neighborhood Curriculum (NC) uses information that is provided by the evaluation of the neighbor solutions performed during a local search.

The main concept of the NC lies on the last strategy synthesized. This single and strong strategy is sufficient to provide enough signal to the search to converge in a dominant strategy. The same idea is observed in the IBR when there is a dominance strategy. However, the presence of other strategies in the scenario helps the heuristic have a strong information signal that can boost the synthesizer in the direction of the dominance script. Using the example in section 3.6, the NC can notice that gates 5, 7, and 8 are important to the search, allowing the algorithm to avoid loops of strategies that can slow the search. To make up the curriculum of strategies as part of the evaluation, NC saves information provided from the evaluation of the neighbor strategy against each strategy in the curriculum. The evaluation process and how the information is stored can be seen in Figure 11.

The NC algorithm lists the current selection of opponents' strategies stored in an ordered list $\Psi^{curriculum}$. All the data necessary to select the important opponents for $\Psi^{curriculum}$ is stored in the dictionary *data_points*. During the local search, neighbor solutions are generated as part of the evaluation process. Each time the NC heuristic is called to evaluate a neighboring solution (see line 4), information about the strategies on

```
1 class NSA:
2     list Ψ^{curriculum} #Selected opponents
3     dict data_points
4     def evaluation(Game 𝒢, script ψ):
5         neighbor_data_boolean = []
6         payoff^{last} = 𝒰(𝒢.s_t, ψ, Ψ^{curriculum}[-1])
7         payoff = 0
8         for ψ_t in Ψ^{curriculum}:
9             payoff^t = 𝒰(𝒢.s_t, ψ, ψ_t)
10            payoff += payoff^t
11            if payoff^{last} == 1 and payoff^t == -1:
12                neighbor_data_boolean.append(1)
13            else:
14                neighbor_data_boolean.append(0)
15        data_points[neighbor_data_boolean]+=1
16        return payoff/len(Ψ^{curriculum})
```

Figure 11 – A pseudo-code in python representing the evaluation method in the NC heuristic.

$\Psi^{curriculum}$ is recorded. Initially, the algorithm evaluates the script score ψ against the last opponent on $\Psi^{curriculum}$ (see line 6), and the payoff is recorded on payoff^{last}. The evaluation continues iterating for each strategy on $\Psi^{curriculum}$ (see lines 8-10). The score payoff^t, which represents the score of the neighbor against one solution in the curriculum, is used to characterize the information that will be collected. If the current strategy ψ wins against the last strategy in the $\Psi^{curriculum}$ but loses against the strategy ψ_t, the score is 1. Otherwise, the score is 0. These scores are used to compose the key-value for the dictionary *data_points*. The intuition on line 11 is to identify the strongest opponents in $\Psi^{curriculum}$ that can be stronger than those in the last strategy, or that provide unique strategies that is stronger than the last inserted for a particular group of scripts.

After an iteration of the local search algorithm, the NC uses the data recorded in the dictionary to select the group of opponents on $\Psi^{curriculum}$. This new group will be used in the next iteration of the local search algorithm. The algorithm to select the new group is shown in Figure 12.

The algorithm begins by selecting the last strategy from the portfolio $\Psi^{curriculum}$ and use it as the initially selected strategy (see line 5) for the new group. For each pair of data and values in data_points, the algorithm calculates the score for each strategy in $\Psi^{curriculum}$. If a strategy has the value 1 in any stored data, its value is incremented accordingly to the *value* calculated by the function *evaluation* (see lines 8-11). If, after measuring the score for each strategy, there is any strategy with a value bigger than zero, the algorithm stops and updates the number of strategies in $\Psi^{curriculum}$ by using the indexes stored at *index_strategies*(see lines 12-13). Otherwise, the algorithm selects

```
1 class NSA:
2     list Ψ^curriculum #Selected opponents
3     dict data_points
4     def select_curriculum():
5         index_strategies = [Ψ^curriculum[-1]]
6         while True:
7             scores = list_zeros(len(Ψ^curriculum))
8             for data, values in data_points.items():
9                 for i in range(len(data)):
10                    if data[i] == 1:
11                        scores[i] += value
12             if get_max_score(scores) == 0:
13                 return update(Ψ^curriculum, index_strategies)
14             i_selected_strategy = argmax(scores)
15             index_strategies.append(i_selected_strategy)
16             new_data = dict()
17             for data, values in data_points.items():
18                 if data[i_selected_strategy] == 0:
19                     new_data[data] = value
20             data_points = new_data
```

Figure 12 – A pseudo-code in python that represents the method select_curriculum on the NC heuristic.

the strategy with the biggest value, adds it to the list *index_strategies*, and remove all the entries with value 1 from the dictionary data_points. Then, the greedy algorithm continues adding strategies until the remaining strategies do not provide any score bigger than zero.

To illustrate the selection process, let's suppose the method is called after a local search iteration with 33 evaluations and the size of $\Psi^{curriculum}$ being three. The *data_point* dictionary contains the entries in table 8:

#	Key	Value
1	(0,0,0)	31
2	(1,1,0)	1
3	(0,1,0)	1

Table 8 – Entries on data_points dictionary.

The algorithm starts by adding the variable *index_strategies* the index two, representing the last strategy included. After this, the score is calculated by using the key values of the dictionary. We have three strategies for $\Psi^{curriculum}$, represented by the index [0,1,2]. Strategy 0 has an entry value equal to 1 in the second key, which computes the value 1 for this strategy. Strategy 1 has two entries with value 1, entries number 2 and 3, and receives the value 2. Strategy 2 has a value of zero for not having any entry with

value 1. These values are stored in the list *scores.Scores* has values [1,2,0], and strategy 1 has the biggest value, being that strategy add in *index_strategies* (current values [2,1]). After this, all the entries for the strategy 1 with value 1 are removed from the *data_point* and remain the key-value $\{(0,0,0)|31\}$. The greedy search is finished because any new strategy is necessary to be included. The curriculum $\Psi^{curriculum}$ will be updated to retain the strategies with indexes 1 and 2, respectively.

Escrever como o método foi utilizado em associação com as buscas locais.

4.4   Analysis of the Algorithm

The heuristic NC has two important points, the data stored and the greedy selection method. First, the dictionary to store all the necessary information; Second, the greedy algorithm used to select the strategies to compose the new $\Psi^{curriculum}$. In the worst situation, the algorithm will have to stock all the evaluated strategies in one local search iteration Υ, each key the size of the number of strategies in $\Psi^{curriculum}$, represented by k. In terms of memory, the worst situation is $O(\ |\ \Upsilon\ |\ \cdot k)$. About the greedy method select_strategies, in the worst case, it needs to evaluate all the entries in the data (Υ) (see Figure 12, line 8). For each entry, it is necessary to count the score for each strategy in $\Psi^{curriculum}$, which is a constant value k. The time complexity for this evaluation will be $O(\ |\ \Upsilon\ |\ \cdot k)$. If the algorithm needs to select all the strategies, the loop will repeat k-times, to guarantee coverage, the worst case will be $O(\ |\ \Upsilon\ |\ \cdot k^2)$.

5 Empirical Evaluation

In this chapter we evaluate NC as a heuristic to guide a local search synthesizer to generate a dominant script, or a strong approximated script best response. We compared NC against SCFP, SSDO, and IBR in three domains: Poachers and Rangers (P&R), Climbing Monkey (CM), and MicroRTS. P&R and CM are toy games we introduced to facilitate the evaluation of the heuristics. MicroRTS is a standard benchmark for game playing algorithms.

5.1 Problems Domains

In this section, we describe the three problem domains we use to evaluate our heuristic.

5.1.1 Poachers and Rangers

Poachers and Rangers is a toy game where *rangers* need to protect the gates of a national park to avoid *poachers* getting into the park. The poachers need at least one unprotected gate to enter the park, and rangers succeed if they protect all the gates attacked by poachers. The game is developed as a two-player zero-sum game without ties. Its utility function returns 1 if the poachers reach at least one unprotected gate and -1 if rangers protect all the gates. The game runs for a budget β of calls to the utility function. Players $P = \{poacher, rangers\}$ play the game alternatively, trying to find the best response to the opponent's strategy. The game starts with a random strategy for each player. Rangers start using the local search to find the best strategy against the current poachers' strategy. After that, the poachers perform the same search looking for a new strategy that can defeat the current rangers' strategy. The game stops immediately when the rangers find the dominant strategy where all gates are protected.

Strategies for both poachers and rangers are represented as a list l of gate numbers. To generate the next (or neighbor) solution ζ' from a current solution ζ, we (1) truncate ζ at a random point and (2) iteratively flip a coin to decide whether or not a new gate number should be added to ζ'. Assuming heads as True, we add (uniformly) sampled gate numbers to ζ' for as long as the coin flip returns heads. ζ' is guaranteed to have at least one gate. Given that we know the dominant strategy for P&R (maximizing the number of protected gates), we use P&R to evaluate the performance of different heuristics in finding such strategy.

5.1.2   Climbing Monkey

Climbing Monkey is a single-player game where a monkey needs to reach the highest branch of a tree. However, the branches are slippery, and the monkey needs to climb branch-by-branch, without skipping anyone. The game starts with a random initial solution l, and it runs for a budget β of calls to the utility function. The only stop condition is when $\beta = 0$. The utility function returns 1 when the current solution has more branches than the old solution and zero otherwise. At each iteration of the game, the monkey's goal is to find a better strategy than his old one, which means to keep moving up. A strategy is represented by an ordered list of branch numbers and a neighbor solution is generated following the approach used in P&R. We use Climbing Monkey as a benchmark to evaluate if heuristics, such as SCFP, unnecessarily keep saving old solutions.

5.1.3   MicroRTS

MicroRTS is a small Real-time strategy (RTS) game designed for research purposes, having an active AI competition[1]. MicroRTS is a deterministic game played with real-time constraints and the action space grows exponentially with the number of units in the state (LELIS, 2021). Figure 13 shows an example of a MicroRTS game state played on a 8×8 grid map. Circles and squares represent units, where a unit's contour color (blue or red) represents the player who controls it.

5.1.3.1   Unit types

MicroRTS has the following types of units: worker, light, ranged, heavy, base, and barracks. The units that can move and attack are represented by a circle, the other units that don't move are represented by a square. Small dark-gray circles represent workers, large yellow circles represent heavy units, small light-blue circles represent ranged units, and small orange circles represent light units. Light-gray squares represent bases (the number in the base shows the amount of resources available to the player). The dark-gray squares represent barracks.

5.1.3.2   Map layout

The dark-green squares on the map are walls that cannot be traversed by units. Light-green squares are resources that can be collected by workers (the amount of resources that can be collected is displayed in the square). The layout of walls on the map might influence the strategies chosen by the players. For example, players can choose to build their structures where they are surrounded by walls as a way of protecting their base against attacks of the opponent.

---

[1]   https://github.com/santiontanon/microrts/wiki
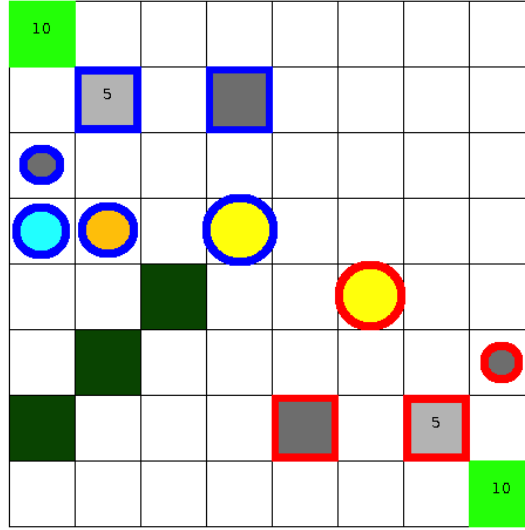
Figure 13 – A MicroRTS game state played on a 8×8 map. The light-green squares at the top-left and bottom-right corners represent resources that can be harvested by worker units, which are represented by small dark-gray circles. Dark-gray squares represent barracks that can be used to train military units. Large yellow circles represent heavy units, small orange circles light units, and small light-blue circles ranged units. Dark-green squares represent walls that units cannot traverse.

5.1.3.3 Hit points

Every unit has hit points that indicate the amount of damage the unit can suffer before being eliminated from the game. Workers and ranged units have fewer hit points than light units, heavy units, and barracks. The base has more hit points than any other unit. Some of the units can attack an enemy unit. If unit u attacks enemy unit u', then u reduces the hit points of u' according to its inflicted damage, which is determined by the unit's type. Workers and ranged units cause the least amount of damage. Light and heavy units cause more damage per attack. Workers, light, and heavy units u can only attack enemy units adjacent to the grid cell u occupies. Ranged units u can attack any enemy unit at max three grid cells away from u.

5.1.3.4 Action scheme

Most of the unit actions require a single game cycle to be executed (RTS games typically have from 10 to 50 game cycles per second (ONTAÑÓN, 2017)), but some of the unit actions (e.g., build a base) take several game cycles to complete (i.e., actions have different durations). Any unit can take a no-op action, which means that the unit waits until the next game cycle. All units other than base and barracks can move one grid cell at a time (up, down, left, and right). Only one unit can occupy a given grid cell at a time. If two units move simultaneously to the same grid cell, then the game server overwrites their actions with the no-op action, which means that the units will perform no action in

the game cycle.

5.1.3.5 Collecting and spending resources

Bases and barracks cannot move nor attack, but the former can train workers and the latter can train light, heavy, and ranged units—all at a cost of resources. Workers can build bases and barracks at the cost of resources. Workers can also collect resources (one resource unit at a time). Once collected, the resource must be delivered to the base by the worker. Once the collected resource is delivered at the player's base, it can be spent to train other units or build bases and barracks.

5.1.3.6 Domain-Specific Language for MicroRTS

The DSL we use for MicroRTS was introduced by Medeiros, Aleixo e Lelis (2022). We use their DSL instead of the one showed in Section 9.1 because theirs accept nested-if's and nested-for's. Their DSL is described by the following context-free grammar:

$$S \to SS \mid \text{for S} \mid \text{if}(B) \text{ then } S \mid \text{if}(B) \text{ then } S \text{ else } S$$
$$\to C \mid \lambda$$
$$B \to b_1(T, N) \mid b_2(T, N) \mid b_3(T, N) \mid b_4(N) \mid b_5(N)$$
$$\to \; \mid b_6(N) \mid b_7(T) \mid b_8 \mid b_9 \mid b_{10} \mid b_{11} \mid b_{12} \mid b_{13} \mid b_{14}$$
$$C \to c_1(T, D, N) \mid c_2(T, D, N) \mid c_3(T_p, O_p) \mid c_4(O_p)$$
$$\to c_5(N) \mid c_6 \mid c_7$$
$$T \to \text{Base} \mid \text{Barracks} \mid \text{Ranged} \mid \text{Heavy} \mid \text{Light}$$
$$\to \text{Worker}$$
$$N \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 10 \mid 15 \mid 20 \mid 25$$
$$\to 50 \mid 100$$
$$D \to \text{EnemyDir} \mid \text{Up} \mid \text{Down} \mid \text{Right} \mid \text{Left}$$
$$O_p \to \text{Strongest} \mid \text{Weakest} \mid \text{Closest} \mid \text{Farthest}$$
$$\to \text{LessHealthy} \mid \text{MostHealthy} \mid \text{Random}$$
$$T_p \to \text{Ally} \mid \text{Enemy}$$

This DSL 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.

- $b_1(T, N)$: Checks if the ally player has N units of type T (HasNumberOfUnits).

- $b_2(T, N)$: Checks if the opponent player has N units of type T (OpponentHasNumberOfUnits).

- $b_3(T, N)$: Checks if the ally player has less than N units of type T (HasLessNumberOfUnits).

- $b_4(N)$: Checks if the ally player has N units attacking the opponent (HaveQtdUnitsAttacking).

- $b_5(N)$: Checks if the ally player has a unit within a distance N from a opponent's unit (HasUnitWithinDistanceFromOpponent).

- $b_6(N)$: Checks if the ally player has N units of type Worker harvesting resources (HasNumberOfWorkersHarvesting).

- $b_7(T)$: Checks if a unit is an instance of Type T (is_Type).

- b_8: Check if a unit is of type Worker (IsBuilder).

- b_9: Checks if a unit can attack (CanAttack).

- b_{10}: Checks if the ally player has a unit that kills an opponent's unit with one attack action (HasUnitThatKillsInOneAttack).

- b_{11}: Checks if the opponent player has a unit that kills an ally's unit with one attack action (OpponentHasUnitThatKillsUnitInOneAttack).

- b_{12}: Checks if an unit of the ally player is within attack range of an opponent's unit (HasUnitInOpponentRange).

- b_{13}: Checks if an unit of the opponent player is within attack range of an ally's unit (OpponentHasUnitInPlayerRange).

- b_{14}: Checks if a unit can harvest resources (CanHarvest).

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

- $c_1(T, D, N)$: Trains N units of type T on a cell located on the D direction of the unit (Build).

- $c_2(T, D, N)$: Trains N units of type T on a cell located on the D direction of the structure responsible for training them (Train).

- $c_3(T_p, O_p)$: Commands a unit to move towards the player T_p following a criterion O_p (moveToUnit).

- $c_4(O_p)$: Commands a unit to attack units of the opponent player following a criterion O_p (Attack).

- $c_5(N)$: Sends N Worker units to harvest resources (Harvest).

- c_6: Commands a unit to stay idle and attack if an opponent units comes within its attack range (Idle).

- c_7: Commands a unit to move in the opposite direction of the player's base (Move-Away).

T represents the type 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. We define the different types, integers, directions, criteria, and players we used in the context-free grammar above.

5.2   Experiments and Results

In this section, we compare the performance of NC, SCFP, SSDO, and IBR as a heuristic to guide a local search algorithm to generate a dominant script, or a strong approximated script best response in P&R, CM, and MicroRTS. For P&R and CM,, we used HC as the local search algorithm. For MicroRTS, we compared six different maps with two local search algorithms: HC and SA. Next, we present the evaluation methodology for each domain.

5.2.1   Poachers and Rangers

In P&R, HC receives as input a heuristic to be evaluated (e.g., NC, SCFP, SSDO, and IBR). The same heuristic will be used by the rangers player Ψ_{ranger} and by the poachers player $\Psi_{poachers}$. However, each heuristic keeps its own independent set of strategies, each starting with a single random strategy. We call $\psi_{poachers}$ the initial solution of the poachers player and ψ_{ranger} the initial solution of the rangers player. At this point, all the heuristics behave as IBR since they have only one strategy. The game runs iteratively until the budget $\beta = 0$. β is decremented based on the evaluation cost of each heuristic. Initially, the rangers player runs the search until it finds the best response ψ'_{ranger} to the set of strategies in $\Psi_{poachers}$. The heuristic $\Psi_{rangers}$ then updates its set of strategies considering this new strategy ψ'_{ranger}. Note that each heuristic performs this update differently. The game ends if ψ_{ranger} protects all the gates. If not, the same process is executed to the player poachers.

Each iteration, a player runs HC until it finds a best response to its opponent strategy. To avoid local maximum solutions, we implemented restarts after a constant
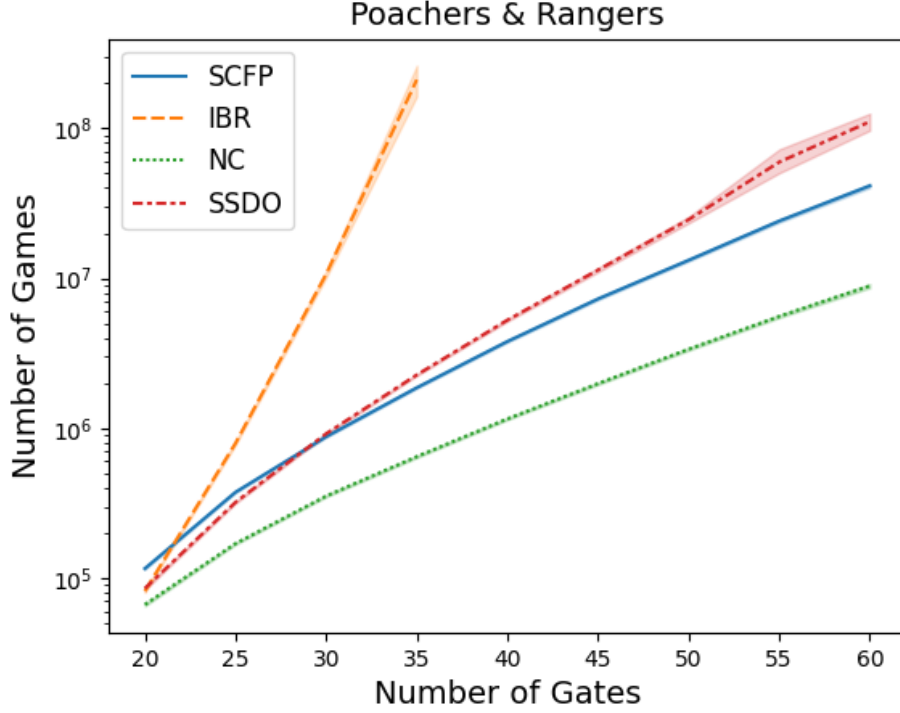
Figure 14 – Average number of gates reached by each heuristic on Poachers and Rangers by number of games played, on 10000 simulations for each method.

value p, initially started with a budget of 10.000 strategy evaluations. After each re-initialization, p is multiplied by a factor of 1.5 of his current value.

Figure 14 shows the results of 10.000 simulations for each heuristic with $\beta = 10^9$. IBR could not solve problems with more than 35 gates and performed worst than all other heuristics. SSDO outperformed SCFP in games with 20 and 25 gates. However, in games with more than 30 gates, SCFP performed better than SSDO. NC outperformed all the other heuristics, solving games with 60 gates using the same amount of evaluations that SSDO and SCFP used to solve games with approximately 45 gates. We performed the same test with HC without restarts. All the heuristics benefited from the restart, and we decided to present only the best result to avoid repetition.

5.2.2 Climbing Monkey

In Climbing Monkey (CM), we also run HC with and without restarts. Similar to P&R, both variations provide the same conclusions, so we show only the variation with restarts, which found solutions with higher number of branches. HC receives as input a heuristic to be evaluated (e.g., NC, SCFP, SSDO, and IBR) and a initial random solution. At each HC iteration, a solution is returned and it is passed thought the heuristics. A new HC iteration is started until the budget $\beta = 10^6$ is exhausted.

In this test-bed, a strategy with the highest branch is a dominant strategy, and it
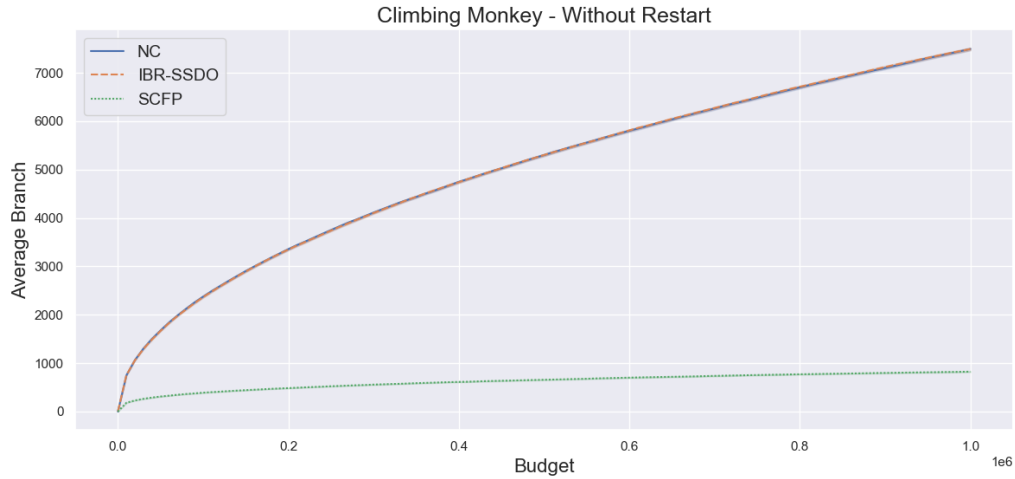
Figure 15 – Average number of branched by each heuristic on Climbing Monkey by budget's limit of 10^6, on 300 simulations for each heuristic.

is enough to provide enough guidance to the search. The next best solution is a solution with, at least, one high branch. The heuristics IBR, SSDO, and NC are statically equal on this domain because they keep using the same best strategy at each iteration. This can be seen on Figure 15. SCFP reaches the small average branch and as the image suggests, the solution grows linearly slow. It happens because of the huge number of solutions in its support group. At each HC iteration, even if the new solution has just one more branch than the previous one, SCFP will include a solution in his group. In the worst case of this scenario, SCFP will add one solution for improvement. For a domain where just one strategy is enough to guide the search, heuristics such as SSDO, SCFP, and NC waste computational resources by keeping the track of old strategies or statistics.

5.2.3 MicroRTS

In the MicroRTS domain, we used six maps from size 8x8, 9x8, 16x16, 24x24 and 32x32, with the specific maximum number of cycles for each map, following the same definitions of the MicroRTS competition:[2].

- **8x8**=Bases Workers 8x8A: 3000 cycles

- **9x8**=No Where To Run 9x8: 4000 cycles

- **16x16**=Bases Workers 16x16A: 4000 cycles

- **24x24A**=Bases Workers 24x24A: 5000 cycles

- **24x24B**=Double Game 24x24: 5000 cycles

- **32x32**=Bases Workers 32x32A: 6000 cycles

---

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

We show the maps in the Figure 16. Maps *NoWhereToRun9x8* and *DoubleGame24x24* (both in the middle of the figure, one top and another bottom) were selected by bringing a different context where both players are separated by one removable obstacle. All tests here performed with the same specification from the MicroRTScompetition: 100 milliseconds for each decision point, full observable game, and the definitions of attack, life, damage of each unit defined by the *Unit Type Table* called **VERSION ORIGINAL FINETUNED**.
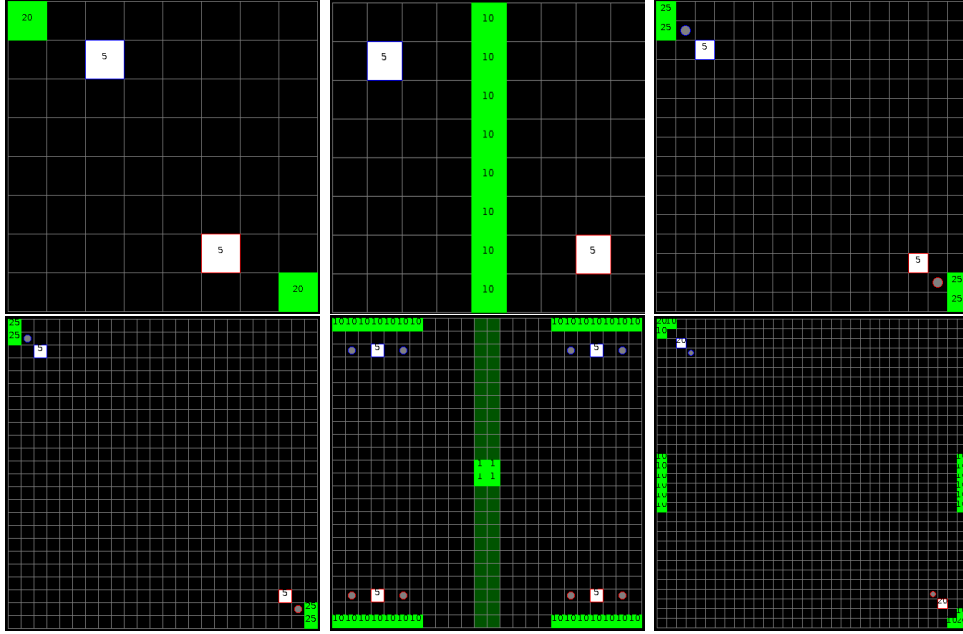
Figure 16 – The six maps used in the MicroRTS experiments. From size 8x8, upper left size, to size 32x32, bottom right side.

We used two local search algorithms, HC and SA for program synthesis introduced in chapters 2.3.1 and 2.3.2, to evaluate the four heuristics. Each heuristic were tested 30 times, in each map, for each algorithm, with the time limit of 72 hours of computation in a single 2.4 GHz CPU with 2 GB of RAM.[3] For each test, we computed the number of matches performed during the 72 hours of computation. To guarantee fairness, for each map, we defined the same budget for each heuristic, in each map. It was necessary because in our experiments, the most costly function is defined by the match used to evaluate the scripts. Unfortunately, it is not a constant time for all matches. We defined a equal budget for each map based on the lower higher value reached by the tests. Strategies synthesized by the algorithm after the defined budget were unconsidered.

5.2.4 Hill-Climbing Results

Figures 17 and 18 shows the average winning percentage of the strategies synthesized during the search accordingly each heuristic. Each heuristic has 30 independent runs,

---

[3] The implementation of all algorithms used the MicroRTS experiments is available at https://github.com/rubensolv/MicroRTS

and the shaded area is the Standard Deviation (SD). Each point in the plot is calculated by evaluating the heuristic against the last strategy of the opponents. For example, in the map 8x8 (first map in the Figure 17), the budget 0.2^{10^6} to NC is calculated against the last strategies (budget equal to 1.0^{10^6}) synthesized by IBR, SCFP, and SSDO.

In all the maps, NC appears as the best heuristic, followed by SSDO, SCFP, and IBR. In some maps, as 9x8, 24x24B, and 32x32, the SD, even with 30 runs, is slightly high, but statically, the results in these maps are expressive. On map 8x8, due to the simplicity of the strategies and the spatial limitation of the map, the difference between NC and SSDO are difficult to observe in the plot. However, in maps with complex strategies, as 9x8 that requires a ranged-rush behavior, the NC is strictly dominant over the other heuristics.

5.2.5 Simulated Annealing Results

Figures 19 and 20 shows the results to four heuristics using the Simulated Annealing in the same six maps. The results with SA shows the NC superiority in all the maps. However, with SA, the results of SCFP and SSDO are not so clear any more. In all maps, the SCFP shows better or equal results when compared against SSDO.

Figure 17 – Results of each heuristic in the maps 8x8, 9x8, and 16x16, against themselves, with Hill-Climbing.

Figure 18 – Results of each heuristic in the maps 24x24 and 32x32, against themselves, with Hill-Climbing.
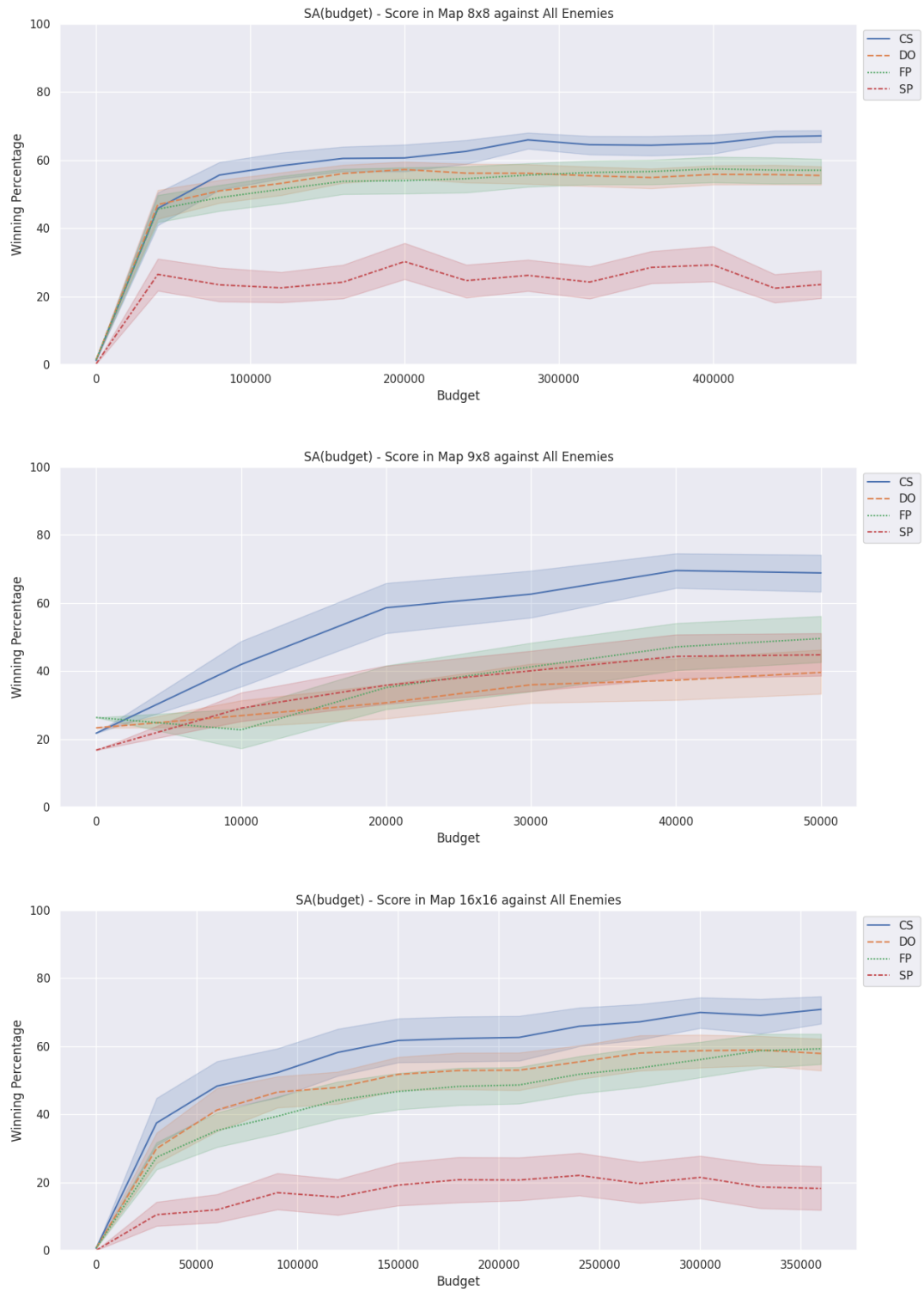
Figure 19 – Results of each heuristic in the maps 8x8, 9x8, and 16x16, against themselves, with Simulated Annealing.
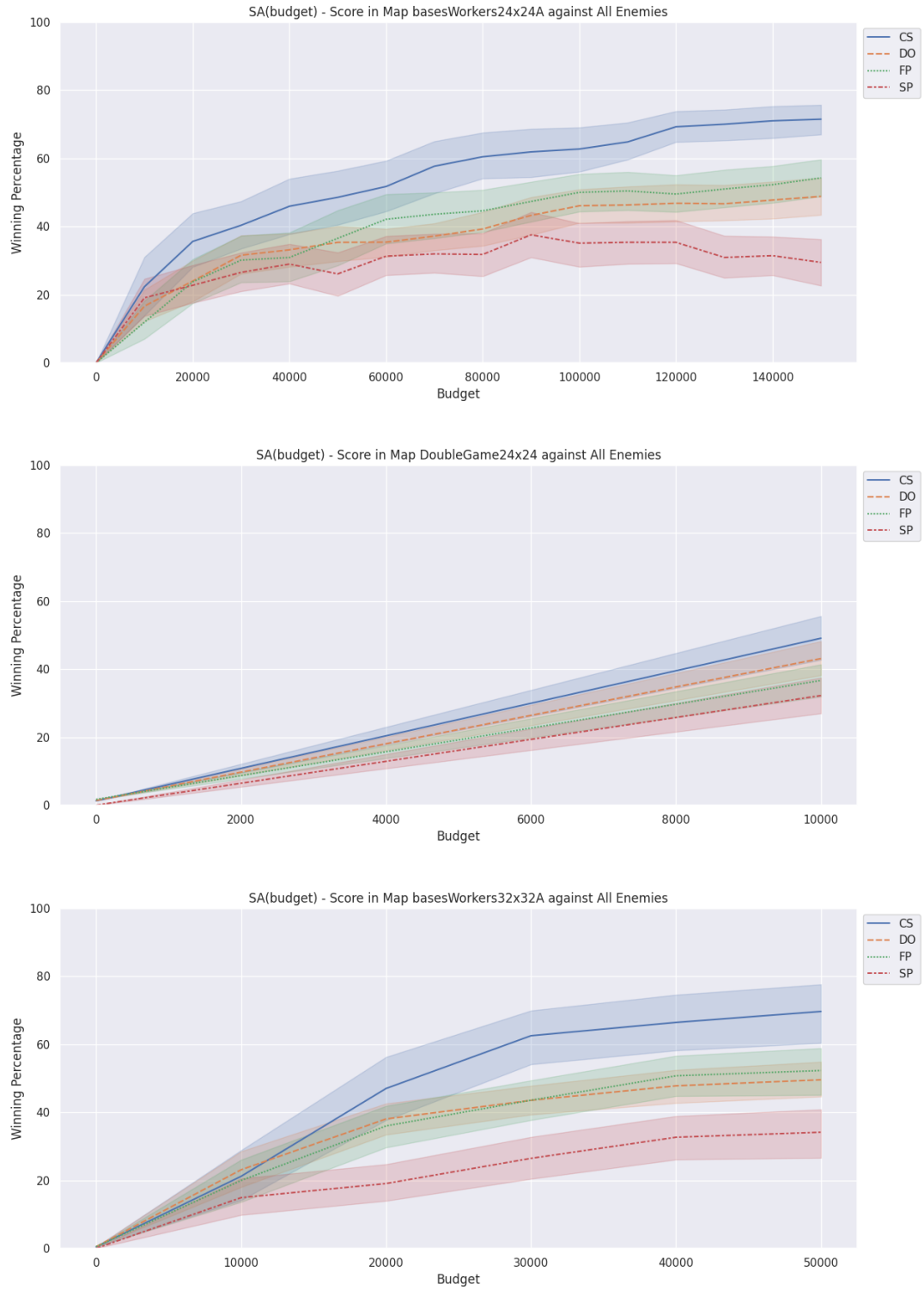
Figure 20 – Results of each heuristic in the maps 24x24 and 32x32, against themselves, with Simulated Annealing.

6 Discussion

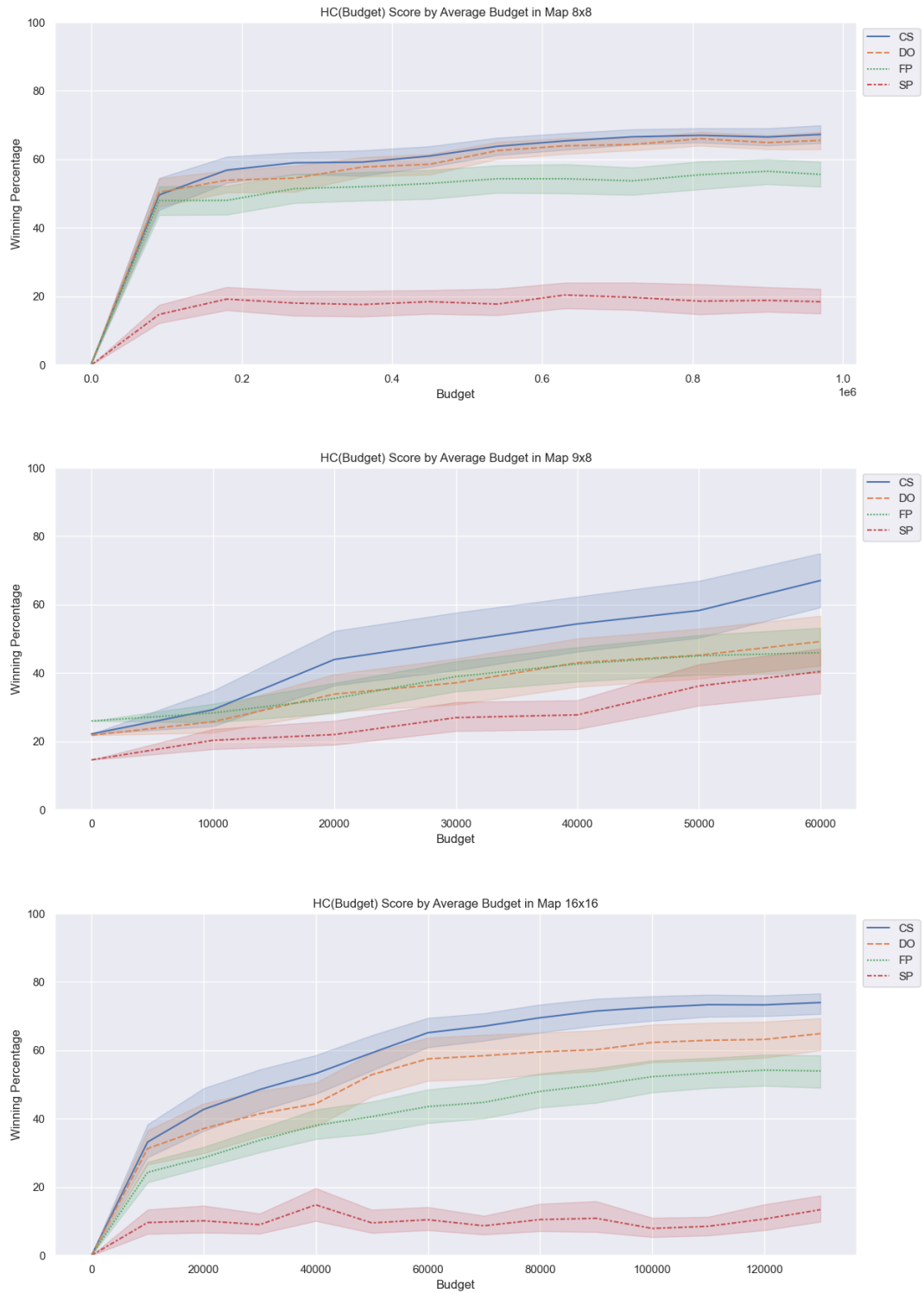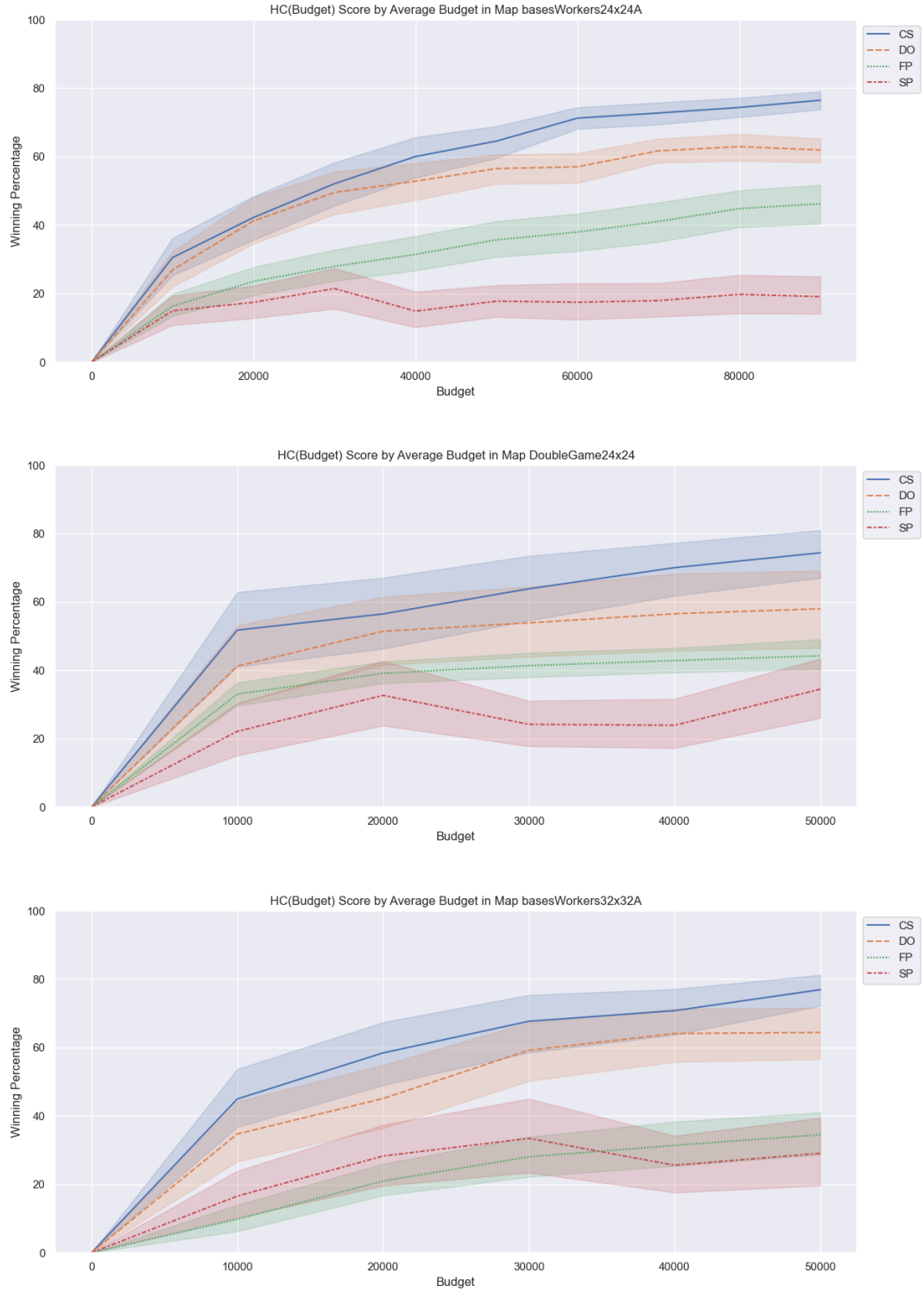In this document, we presented the Neighborhood Curriculum (NC), a heuristic to guide local search algorithms to synthesize programmatic strategies. NC uses previous neighbor solutions evaluations to define a set of support strategies which are representative enough to enhance the synthesis of an dominant strategy for a set of strategies. We performed tests in three different domains, Poachers and Rangers (P&R), Climbing Monkey (CM), and MicroRTS. These tests evaluated the NC, and compared it with three baselines: Iterated Best Response (IBR), Simple Cumulative Fictitious Play (SCFP), and Script-Space Double Oracle (SSDO). IBR was previously used in some works and proved to work for tasks to find an approximate best response for strategy (MARIÑO et al., 2021; MEDEIROS; ALEIXO; LELIS, 2022; MARIÑO; TOLEDO, 2022). However, in our project, we attempt to find a dominant script, which the IBR is not a good heuristic to guide the search.

Based on game theory approaches, we introduced two new heuristics, Simple Cumulative Fictitious Play (SCFP) and Script-Space Double Oracle (SSDO), to be used as baselines. These two baselines are necessary because they have vital concepts to improve upon the IBR's weakness. The Simple Cumulative Fictitious Play (SCFP) guarantees that any strategy synthesized by the search is missed.Simple Cumulative Fictitious Play (SCFP) stored all the strategies synthesized during the search in the support group. All strategies support providing a better guidance signal to the synthesizer. However, Simple Cumulative Fictitious Play (SCFP) is expensive computationally and fail to converge to a dominant strategy under restriction. SSDO appears as the logical solution for IBR and SCFP. While IBR kept just one strategy, and SCFP stored all the previous ones, SSDO is flexible to online select a strong and representative group of strategies. These strategies guide the search by solving the game and selecting the strongest solutions according to probabilities. To investigate the heuristic, we developed two toy games that help us understand the problem.

The first toy game in this domain, Climbing Monkey (CM), was formulated because only the last strategy is sufficient to efficiently conduct the search, as observed in Section 5.2.2. The heuristics IBR and SSDO have the same results since the last strategy synthesized is the strongest compared to previous one. When SSDO solves the game, it selects the same strategy that IBR uses, which is a strong indication that the search shoul be conducted. In this scenario, SCFP uses the computational resources inefficiently by keeping unnecessary strategies in the support group, and it does not archive more than 800 branch size. However, NC has results similar to IBR and SSDO because the selection process is able to pick up the same strategies than the other two methods.

The second toy game, Poachers and Rangers (P&R), was formulated to emphasize how a better group of strategies can guide the search to converge faster to a dominant strategy, being that strategy found when rangers protect all the gates. As observed in Section 5.2.1, the IBR was not able to solve problems with more than 35 gates. This happens because IBR allows the search to converge to a better solution for just one strategy at a time, and that convergence rarely finds a dominant strategy with more gates. For a small number of gates, SSDO is a strong heuristic when compared with SCFP because even a small group of support strategies can help the search find the dominant strategy. However, for more than 30 gates, SSDO uses more computational resources than SCFP to solve a problem of the same size. This is because SSDO does not select all necessary strategies for the group of support strategies and that gap costs more computational resources to allow the search find a dominant strategy.

The MicroRTS domain shows how the NC provides a strong signal to guide the search towards a pure dominant strategy. In the tests with the HC, the NC was superior on almost all maps. For map 8x8, NC, and SSDO have similar performance with a slight predominance of NC. This happens because of the simplicity of the map and the strategies that are found in it. Even a simple WorkerRush strategy is able to achieve high scores on this map, as we observed in the MicroRTS AI Competition in previous results. In 2018, WorkerRush strategy was the hyphenate strategy on an 8x8.[1]

In the tests with SA, the NC is superior in all maps, even in the 8x8. The SCFP and SSDO have statistically equal scores in all the maps. It is hard to define which one is the second best because the SD makes the two strategies statistically equal. We believe the SA allows the search to synthesize better strategies for this domain. Even the IBR had increased the results on maps 9x8, 16x16, and 24x24A. We hypothesize is that the SA synthesizer could produce better strategies than the HC by exploring the search space more effectively because of because of the addition of random walks, intrinsic to the algorithm.

---

[1]   https://docs.google.com/spreadsheets/d/1RTw6UxpLLnjx172mewsg-XHX8NDZAtT1HTwss8I8tMo/edit#gid=0

7 Future Works

In this Chapter, we show future works that are already under development and some to describe the next paths to be followed in the doctorate.

7.1 Interpretability of Programmatic Strategies

This work consists of a project to measure and evaluate the interpretability of programmatic strategies and how that feature can help humans to recognize weakness on their strategies and learn with it. This project is already in the final stages of human experiments, with all the technological apparatus developed. Initially, we made an interface to help humans write their own programmatic strategies supported by the μLanguage, as shown in Section 9.1. The interface is presented in Figure 21.
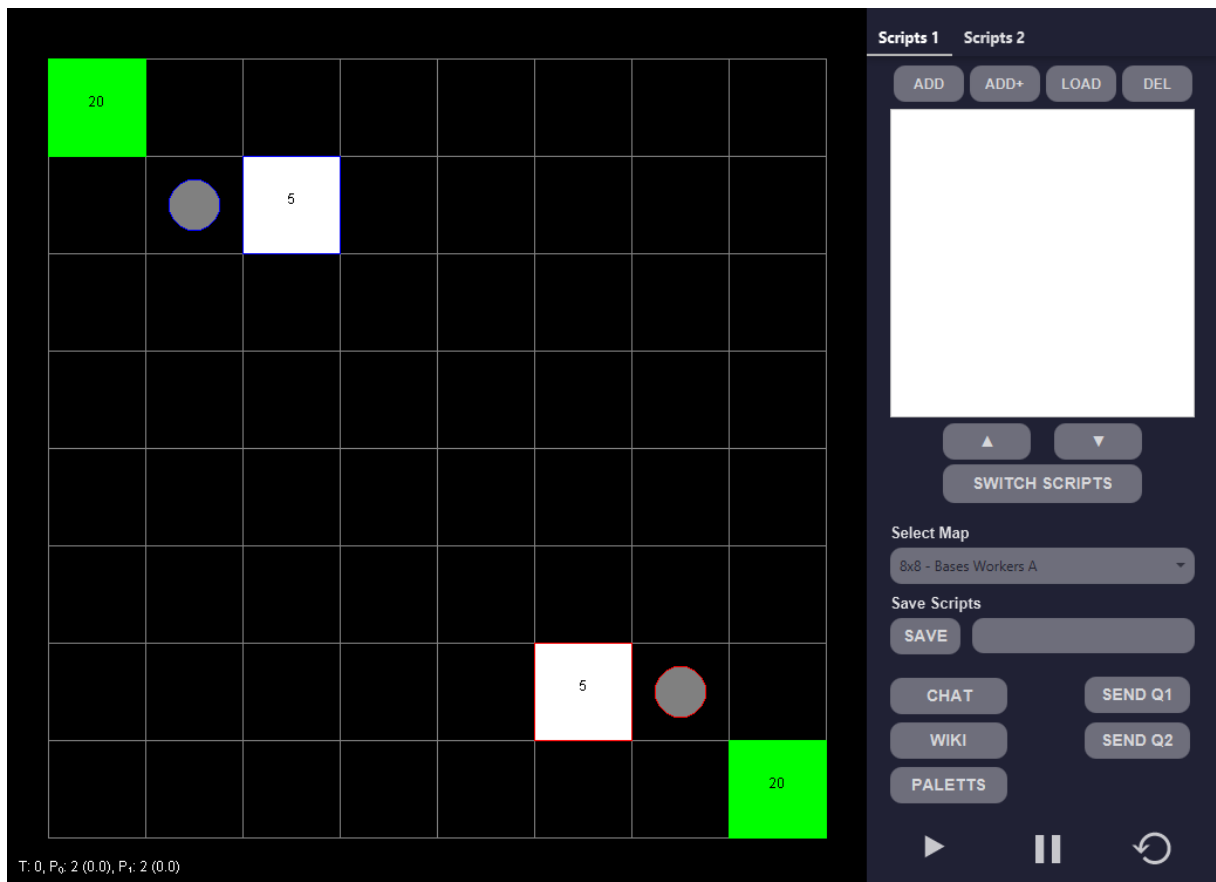
Figure 21 – Interface for humans to write programmatic strategies for MicroRTS using μLanguage.

The interface has entire documentation to help users understand how the interface works, and explains all the facilities.[1] We recorded two tutorials on **Youtube** to provide

---

[1] https://github.com/Tathy/Interface_microRTS/wiki

examples of usage and contextualization. These videos can be found in tutorial one and tutorial two. To perform tests with humans, we submitted the project to the ethics committee of the University of Alberta Research Ethics Board with the title Assistive Tool for Writing Scripts for RTS Games. It was approve under number Pro00103548. The experiments with humans consisted of six steps:

1. We sent the invitation via email including instructions for the students of Universidade de Viçosa. This letter can be seen in Appendix C;

2. With the interface, we asked the participants to create scripts for three different maps with sizes 8x8, 9x8, and 15x14;

3. When the scripts were finished, the participants need to submitted the scripts, and then they will answer Questionnaire 1;

4. Our algorithm performs improvements in the scripts, by trying to synthesize strategies able to surpass the on send by the participant. When improvements are found, they are returned to each participants, by email-response, in the format shown in Appendix B;

5. The participants will answer a second questionnaire based on the email-response sent to them;

6. The answers are used to evaluate the interpretability of program synthesis and how it can help humans improve their codes.

The three maps selected for the experiments: Bases Workers 8x8, No Where To run 9x8, and It's Not Safe 15x14. These are the original names from the MicroRTS' repository. Figure 22 shows the maps. The participants, initially focused on Brazilian community, were asked to write at least one programmatic strategy for one map to participate in the experiment.
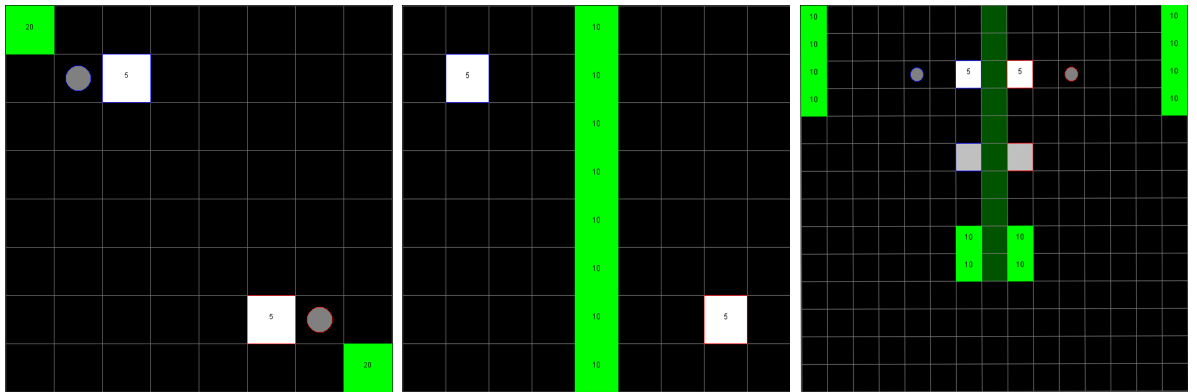
Figure 22 – Maps 8x8, 9x8 and 15x14, from left to right, were chosen to be used in the experiments.

During the concept of our experiment, we identify some key statistics that should be collected from each participant. These data are collected according to the participant works on the interface. Above, we describe the statistics collected:

1. Number of times the interface was opened;

2. Number of times the participant pressed the buttons **play**, **pause**, and **restart**;

3. Time expended on each one of the maps;

4. Color pallet used by each participant.

All the data collected are stored at the University of Alberta, as required by the ethics committee. The participants' data are anonymously saved, even the email is encrypted on the database. When the participants finished developing the programmatic strategies, they will attach the scripts and filled in Questionnaire one, shown in Figure 23. After Questionnaire One is submitted, the button for it is deactivated. The questions can be seen in Appendix E.

Figure 23 – Questionnaire One, used to collect initial information and the scripts written by the participant.

Once answers and scripts are collected and stored, our algorithm automatically starts to return the data for each participant. For each script, we synthesized a best

response for it. When the best response is found, we run the algorithm a second time, trying to synthesize a third script that generates the best response for the two previous scripts. We used a Simulated Annealing as the synthesizer (see Section 2.3.2), and in the first run, the script submitted by the participant is used as the reference strategy for the heuristic IBR. On the second run, we aggregate the initial script and its best response, and we feed the SCFP's heuristic with both scripts. The third strategy synthesized is the best response for an SCFP execution. We performed eight runs for each map, and the best run was selected as participants answer. We prioritize the scripts with a high score and less AST's size to be the answer for the participant.

Figure 24 – Questionnaire Two with specific questions for each map and one general question about the experiment.

Once the algorithm finish the search for scripts, describe above, we sent the answer-email with a document for the participant a txt file with all the scripts synthesized. The document can be seen in Appendix B. The text used to advertise the participant for the second part of the experiment can be seen in Appendix D. Once the email is sent, the button to answer the Questionnaire Two is activated on the interface used by the participants to perform the test. The Questionnaire Two is presented in Figure 24, which one tab for each map, and it contains the questions shown in Appendix E.

We performed a pilot test with one guest to evaluate the flow and clarity of the experiment. The collected data of this participant says that he is a gamer. However, he does not play RTS games. Regardless, the participant could wrote strong scripts against

the existing examples in the interface, just using the wiki and tutorials. The answers to Questionnaire Two show that the participant could explain the weakness in his code and how the other two strategies could explore the weakness on it.

As the final part of this project, we plan to perform a final test with more humans to evaluate the interpretability on programmatic strategies. Considering the results, we believe the experiment will give directions about how this process can help humans to improve the understanding system through examples. We plan to submit this work on ACM CHI Conference on Human Factors in Computing Systems (A1) or ACM CHI PLAY (A3) depending on the results.

7.2   A Local-Search Alpha-League for Program Synthesis

AlphaStar (VINYALS et al., 2019b) was presented in 2019 by the DeepMind as the first AI able to defeat a professional player in one of the most challenging RTS' game: Starcraft II. To reach this achievement, AlphaStar uses a novel multi-agent learn algorithm shown in Figure 25. The initial agents, represented by deep neural networks, are trained with replays of human players. The agents are able to mimic strategies used by players on the real game. These initial agents are used as seed to a the AlphaStar League, a multi-agent reinforcement learning process. The representation of the league is shown in Figure 26. The agents leans from games against other competitors of the league. The agents are trained playing with selected players on the league. It is necessary to ensure that each agent perform well against strongest strategies without forget how to defeat earlier ones. The league has three roles in the process, and each role has the following function:
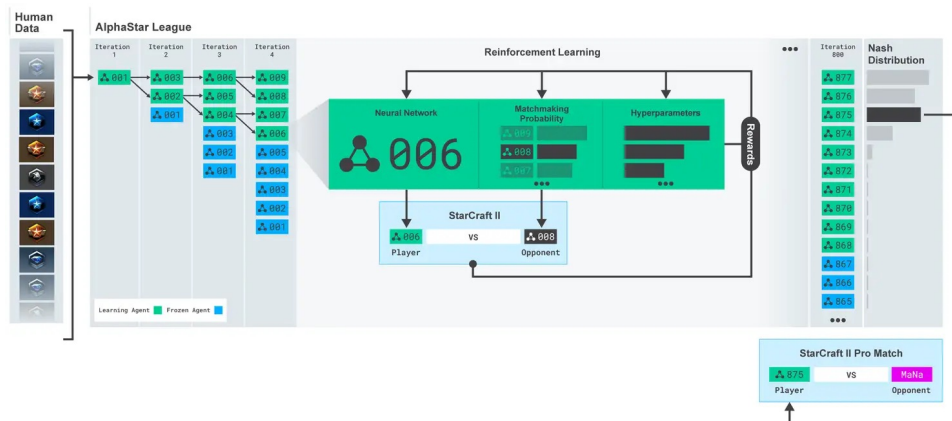
Figure 25 – AlphaStar representation. The agents on iteration one are trained with human game replays, and then they are trained inside the league. At each iteration, new agents are trained and old versions are frozen. After n iteration, the agents with high Nash distribution are used to play against professional players.

- Main Agents: These agents have the goal of win against every strategy in the league. They are trained against all the past players and themselves;

- Main Exploiters: They have the goal of finding flaws in the Main Agents. By exposing these flaws, the Main agents can grow stronger. These agents are trained just against versions of the Main Agents;

- League Exploiters: They have the goal to find a strong strategy while training with all the players in the league.
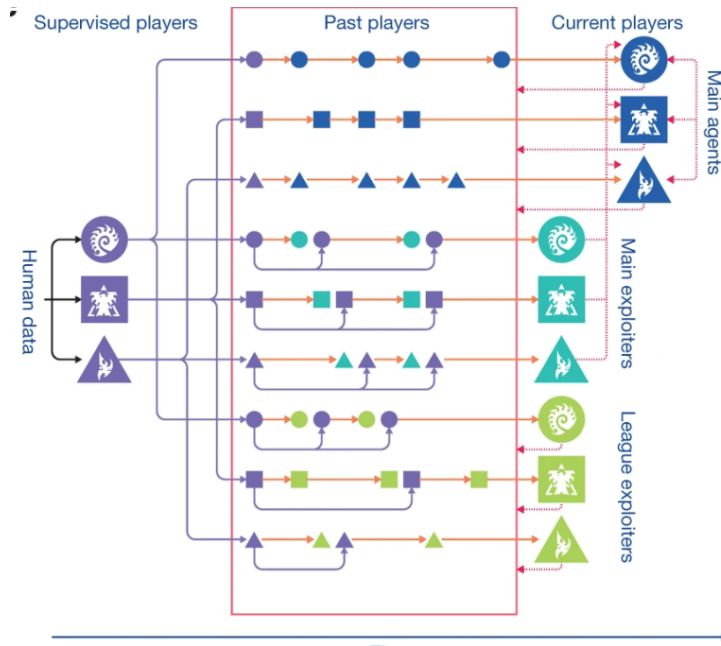
Figure 26 – AlphaStar League in detail. Each initial player (supervised) is an agent trained from replays. They are trained in the league through a multi-agent learning process. The league has three roles: Main agents, Main Exploiters, and League Exploiters.

In this future work, we are interested in developing an AlphaStar League with Local Search algorithms to synthesize programmatic strategies. The initial scheme, called AlphaDSL League, is shown in Figure 27. The first phase of AlphaStar League trains the agents with replays of human players. We suggest replacing this phase by using a sketch cloning algorithm introduced by Medeiros, Aleixo e Lelis (2022). The replays will be replaced by instances of scripts and agents that already exist in the MicroRTS ' repository. We called this Script Phase. These mimic agents will be trained to respect the same three roles:

- Main Agents: They will be synthesized by playing against versions of themselves with a IBR, or SSDO, or NC. If the statistic indicates a dominant strategy in the league, the IBR heuristic is used. If not, and the quantity of memory to store neighbor solutions is not prohibitive, the main agents will be trained with NC heuristic.
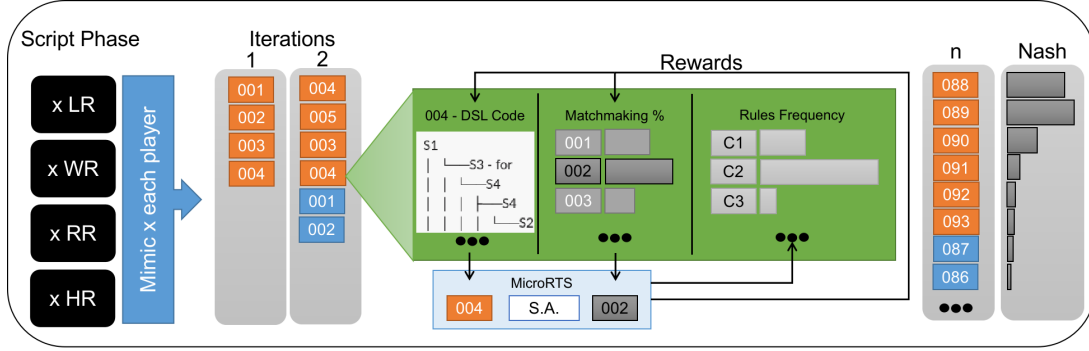
Figure 27 – AlphaDSL League is composed by the script phase and iterations of a local
search algorithm to synthesize programmatic strategies.

- Main Exploiters: They will be trained with batches of past versions of the main
  agents. The NC heuristic will be used if the memory allows it. If not, SSDO.

- League Exploiters: We need to investigate if the SCFP can be used as heuristics for
  that situation.

We initially investigate the AlphaDSL league without the script phase using the
DSL presented on Section 9.1. In this initial tests we used the Simulated Annealing as local
search algorithm, and the results are shown on AlphaDSL League Website. Evaluating
these tests we identified some challenges:

1. Which local search algorithms to use?

2. Can we use the method Lasi (MARIÑO et al., 2021) to reduce the DSL in the script
   phase?

3. How to ensure diversity within the league?

4. Would it be possible to generate strategies for multiple maps?

Trying to answer the first challenge, we performed tests with SA, and HC to pro-
gram synthesis. As Mariño e Toledo (2022) showed, the genetic algorithms We need to
investigated if typical algorithms to synthesis of programs can be used in this domain.
Some of these algorithms are Bottom-Up, Top-Down, Symbolic search, and other. Follow-
ing the next challenge, we pretend to use the script phase to reduce the DSL, as suggested
by Mariño et al. (2021). Which this reduction the AlphaDSL League can reduce the search
space. The third challenge is important because diversity in the league guarantees con-
vergence for a strong strategy. To the best of our knowledge, all the actual researches to
synthesize programmatic strategies, the scripts are map-dependent. We intend to investi-
gate if the league can be used to generate script for maps with the same size or different
sizes. We initially suggested an architecture as shown in Figure 28. This architecture can

solve the problem about multiples maps by keeping a central league of statistic. Statistics can be used to define the best solutions discovered by multiples instances of the league. The instances can receive strategies from the central league to improve diversity and robustness.
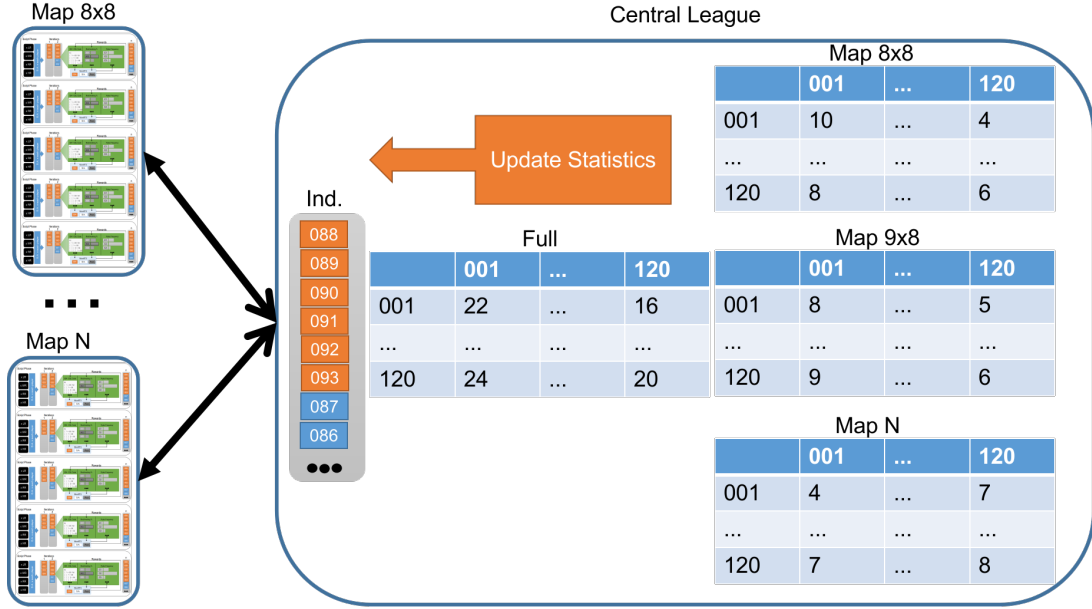
Figure 28 – AlphaDSL League architecture for multiples maps.

7.3 Combining Programmatic Strategies to Improve Abstraction

The question introduced in the last Section, "Would it be possible to generate strategies for multiple maps?", is the main motivation of this future work. Each programmatic strategy is a AST with multiples nodes, and sometimes, strategies to different maps has common branches. Our idea is propose an algorithm that is able to merge two or more different strategies keeping both behaves. Which means, the strategies combined should be strong has the originals. We enumerate some challenges on this research, that can be used to guide the project.

1. Can we use algorithms to combine binary search trees for AST, as suggested by Demaine et al. (2013)?

2. Algorithms for code comparison (BAXTER et al., 1998) might be useful to identify fragment of AST that can be combined.

3. How to identify and generated conditionals, automatically, that can be used to combine fragments of nodes from two different AST's.

We believe that these combinations can allow the generation of strategies with multi-behaviours that will be able to compose strong portfolios of strategies as used in

the Portfolio Greedy Search (CHURCHILL; BURO, 2013), Stratified Strategy Selection (LELIS, 2017), and A3N (MORAES; LELIS, 2018).

8 Study Schedule

In this chapter we show how the next 24 months of PhD is planned to achieve all the projects and future works introduced. We divided the 24 months in 12 bimonthlies to facilitate the display. We enumerated each work and future work accordingly with the following specification:

1. **Project 1:** Guiding Local Search Algorithms for Synthesis of Programmatic Strategies

2. **Project 2:** Teaching Humans to learn with Programs: The interpretability skills

3. **Project 3:** A local-search Alpha-League for Program Synthesis

4. **Project 4:** Combining Programmatic Strategies to Improve Abstraction

Project	Tasks	Budget											
		1	2	3	4	5	6	7	8	9	10	11	12
# 1	Preparing article Extend article Submission and improvements												
# 2	Experiments with humans Preparing article Submission and improvements												
# 3	Project update Empirical Tests Preparing article Submission and improvements												
# 4	Design and code Evaluation and improvements Preparing article Submission and improvements												
Conclusion	Thesis preparation												

Table 9 – Action schedule for the next 12 bimonthlies, divided into four projects.

9 Contributions

In this section, I briefly describe a few of the contributions already made until the present date.

9.1 μLanguage: a Domain-Specific Language for MicroRTS

I developed a script language for MicroRTS called μLanguage, which is an integral part of the MicroRTS' repository.[1] The DSL is specified as follows:

$$S_1 \to C\,S_1 \mid S_2\,S_1 \mid S_3\,S_1 \mid \epsilon$$
$$S_2 \to \text{if}(S_5) \text{ then } \{C\} \mid \text{if}(B) \text{ then } \{C\} \text{ else } \{C\}$$
$$S_3 \to \text{for (each unit } u)\,\{S_4\}$$
$$S_4 \to C\,S_4 \mid S_2\,S_4 \mid \epsilon$$
$$S_5 \to notB \mid B$$
$$B \to b_1 \mid b_2 \mid \cdots \mid b_m$$
$$C \to c_1\,C \mid c_2\,C \mid \cdots \mid c_n\,C \mid c_1 \mid c_2 \mid \cdots \mid c_n \mid \epsilon$$
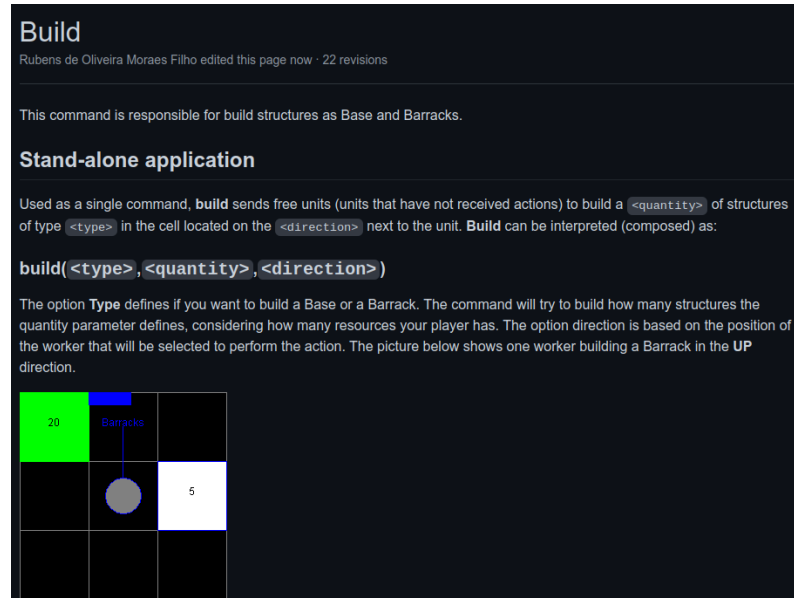
Figure 29 – Example of the command **build** and how it is explained in the wiki.

The entire DSL is described in the MicroRTS' wiki [2] and contains detailed examples of each command and how to use the DSL and his framework. Figure 29 shows a

---

[1] https://github.com/santiontanon/microrts
[2] https://github.com/santiontanon/microrts/wiki/Synthesis-of-Programmatic-Strategies-Framework—SynProS

fragment of the wiki and how a command is explained in detail. The wiki is comprised of 31 pages with images and detailed examples of each command, resource, and specifics of the DSL.

9.2   SynPros - Synthesis of Programmatic Strategies

I created the Synthesis of Programmatic Strategies (SynPros) as a track in the MicroRTS AI Competition. [3] The idea of the track is to advance research in XAI in computer games. The DSL framework is specified in Section 9.1. The DSL defines the search space, and the competitors have all the MicroRTS necessary resources to write a synthesizer for this environment. Figure 30 shows the logo of the track and a fragment of the page.

Figure 30 – The SynPros' logo and title.

The website describes all the rules and necessary information about the track. We had two competitors in the first edition, and we plan to extend it to this year.

9.3   (Paper) Programmatic Strategies for Real-Time Strategy Games

I published a paper with the first self-play algorithm to synthesize programmatic strategies for real-time strategy games using the DSL described above (see Section 9.1). [4] One contribution to program synthesis from this paper is that we introduced a Language Simplification (LS2) system able to synthesize scripts for RTS games by reducing the DSL following traces. The project was developed with the researchers Julian R. H. Mariño,Tassiana C. Oliveira,Claudio Toledo, and Levi H. S. Lelis. The work was published in the Association for the Advancement of Artificial Intelligence (AAAI), qualis A1, conference in 2021. The entire paper can be seen at Appendix A.

---

[3]   https://rubensolv.github.io/synpros-microrts/
[4]   https://ojs.aaai.org/index.php/AAAI/article/view/16114

10 Conclusion

We explored how the program synthesis can be used in three different contexts: search, interpretability, and music composition. As the main point of this project, we introduced three new heuristics for synthesizing programmatic strategies, and we evaluated them in different domains. The principal heuristic, Neighborhood Curriculum, achieved better results than any other heuristic.

For future works, we presented a project, which is already in the final stages, exploring the interpretability of program synthesis to produce interaction between humans and computers. We developed a DSL, an interface, tutorials, and the entire experiment with humans. This experiment intend to measure how the interpretability can help humans understand mistakes or weakness in their codes, being these codes programmatic strategies. We also present a full project example to demonstrate the perspective of the interpretability idea. In addition, we also suggested two new future works to extend the current research. One applies to combining programmatic strategies, and the other is a variation of Alpha-League for Programmatic Strategies.

We believe that the projects and ideas presented in this work have fundamental concepts that can work together with the field of program synthesis, interpretability in Artificial Intelligence and Explainable Artificial Intelligence.

The work with heuristics for programmatic strategies have the potential to advance the state-of-art by allowing the search algorithms to use strong heuristics during the synthesis. Essentially, the programmatic strategies synthesized are more robust with better heuristics. The synthesizer does not need to use more resources during the search to reach convergence for a pure dominant strategy, if it exists. The same lies in the project about interpretability. It has the potential to demonstrate how humans can learn and identify mistakes or weaknesses in their own code by analyzing improvements. Also, this idea can be used to identify wrong behaviors in programmatic strategies and transfer this knowledge to humans.

The Alpha-League project with local-search algorithms has the potential to be presented as a direct application of the Alpha-League for programmatic strategies, in the same way as by Mariño e Toledo (2022). Furthermore, the project to combine strategies allowing one strategy to pursue different strategies can be a options to create powerful scripts in game scenarios.

Bibliography

ADADI, A.; BERRADA, M. Peeking inside the black-box: A survey on explainable artificial intelligence (XAI). **IEEE Access**, IEEE, v. 6, p. 52138–52160, 2018.

ALUR, R. et al. Syntax-guided synthesis. In: **Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design**. [S.l.: s.n.], 2013. p. 1–17.

BALOG, M. et al. Deepcoder: Learning to write programs. In: **Proceedings International Conference on Learning Representations**. [S.l.]: OpenReviews.net, 2017.

BASTANI, O.; INALA, J. P.; SOLAR-LEZAMA, A. Interpretable, verifiable, and robust reinforcement learning via program synthesis. In: SPRINGER. **International Workshop on Extending Explainable AI Beyond Deep Models and Classifiers**. [S.l.], 2022. p. 207–228.

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.

BAXTER, I. D. et al. Clone detection using abstract syntax trees. In: IEEE. **Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)**. [S.l.], 1998. p. 368–377.

BENBASSAT, A.; SIPPER, M. Evolving board-game players with genetic programming. In: **Proceedings of the 13th annual conference companion on Genetic and evolutionary computation**. [S.l.: s.n.], 2011. p. 739–742.

BENBASSAT, A.; SIPPER, M. Evolving both search and strategy for reversi players using genetic programming. In: IEEE. **2012 IEEE Conference on Computational Intelligence and Games**. [S.l.], 2012. p. 47–54.

BERGER, U. Brown's original fictitious play. **Journal of Economic Theory**, Elsevier, v. 135, n. 1, p. 572–578, 2007.

BERNER, C. et al. Dota 2 with large scale deep reinforcement learning. **arXiv preprint arXiv:1912.06680**, 2019.

BONANNO, G. **Game Theory**. [S.l.]: CreateSpace Independent Publishing Platform, 2018.

BOSANSKÝ, B. et al. Algorithms for computing strategies in two-player simultaneous move games. **Artificial Intelligence**, v. 237, p. 1–40, 2016.

BRIGANTI, G.; MOINE, O. L. Artificial intelligence in medicine: today and tomorrow. **Frontiers in medicine**, Frontiers, v. 7, p. 27, 2020.

BROWN, G. Iterative solution of games by fictitious play, 1951. **Activity Analysis of Production and Allocation (TC Koopmans, Ed.)**, p. 374–376, 1951.

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. **Proceedings of the IEEE Conference on Computational Intelligence and Games**. [S.l.], 2018. p. 1–8.

CARMICHAEL, F. **A guide to game theory**. [S.l.]: Pearson Education, 2008.

CARVALHO, D. V.; PEREIRA, E. M.; CARDOSO, J. S. Machine learning interpretability: A survey on methods and metrics. **Electronics**, Multidisciplinary Digital Publishing Institute, v. 8, n. 8, p. 832, 2019.

CHEEMA, S. et al. A practical framework for constructing structured drawings. In: **Proceedings of the 19th international conference on Intelligent User Interfaces**. [S.l.: s.n.], 2014. p. 311–316.

CHEN, C. et al. An algorithm and user study for teaching bilateral manipulation via iterated best response demonstrations. In: IEEE. **2017 13th IEEE Conference on Automation Science and Engineering (CASE)**. [S.l.], 2017. p. 151–158.

CHURCHILL, D.; BURO, M. Portfolio greedy search and simulation for large-scale combat in StarCraft. In: **Proceedings of the Conference on Computational Intelligence in Games**. [S.l.]: IEEE, 2013. p. 1–8.

CHURCHILL, D.; SAFFIDINE, A.; BURO, M. Fast heuristic search for RTS game combat scenarios. In: **Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**. [S.l.: s.n.], 2012.

DAHLBOM, A.; NIKLASSON, L. Goal-directed hierarchical dynamic scripting for rts games. In: **Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**. [S.l.: s.n.], 2006. p. 21–28.

DAVID, C.; KROENING, D. Program synthesis: challenges and opportunities. **Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences**, The Royal Society Publishing, v. 375, n. 2104, p. 20150403, 2017.

DEEPMIND. **AlphaStar Deep Mind**. 2019. <https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/>. Accessed: 2019-02-25.

DEMAINE, E. D. et al. Combining binary search trees. In: SPRINGER. **International Colloquium on Automata, Languages, and Programming**. [S.l.], 2013. p. 388–399.

DOŠILOVIĆ, F. K.; BRČIĆ, M.; HLUPIĆ, N. Explainable artificial intelligence: A survey. In: IEEE. **2018 41st International convention on information and communication technology, electronics and microelectronics (MIPRO)**. [S.l.], 2018. p. 0210–0215.

D'ANTONI, L.; SAMANTA, R.; SINGH, R. Qlose: Program repair with quantitative objectives. In: SPRINGER. **International Conference on Computer Aided Verification**. [S.l.], 2016. p. 383–401.

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.

GENUGTEN, B. Van der. A weakened form of fictitious play in two-person zero-sum games. **International Game Theory Review**, World Scientific, v. 2, n. 04, p. 307–328, 2000.

GULWANI, S. Dimensions in program synthesis. In: **Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming**. [S.l.: s.n.], 2010. p. 13–24.

GULWANI, S. Automating string processing in spreadsheets using input-output examples. **ACM Sigplan Notices**, ACM New York, NY, USA, v. 46, n. 1, p. 317–330, 2011.

GULWANI, S. Programming by examples (and its applications in data wrangling). In: **Verification and Synthesis of Correct and Secure Systems**. [S.l.: s.n.], 2016.

GULWANI, S.; POLOZOV, O.; SINGH, R. Program synthesis. **Foundations and Trends® in Programming Languages**, v. 4, n. 1-2, p. 1–119, 2017. ISSN 2325-1107. Disponível em: <http://dx.doi.org/10.1561/2500000010>.

HAMET, P.; TREMBLAY, J. Artificial intelligence in medicine. **Metabolism**, Elsevier, v. 69, p. S36–S40, 2017.

HE, H. et al. The challenges and opportunities of artificial intelligence for trustworthy robots and autonomous systems. In: IEEE. **2020 3rd International Conference on Intelligent Robotic and Control Engineering (IRCE)**. [S.l.], 2020. p. 68–74.

HEINRICH, J.; LANCTOT, M.; SILVER, D. Fictitious self-play in extensive-form games. In: PMLR. **International conference on machine learning**. [S.l.], 2015. p. 805–813.

HO, T.-H.; CAMERER, C.; WEIGELT, K. Iterated dominance and iterated best response in experimental" p-beauty contests". **The American Economic Review**, JSTOR, v. 88, n. 4, p. 947–969, 1998.

IDC, I. D. C. **Investment in Artificial Intelligence Solutions Will Accelerate as Businesses Seek Insights, Efficiency, and Innovation, According to a New IDC Spending Guide**. 2021. <https://www.idc.com/getdoc.jsp?containerId=prUS48191221#:~:text=The%20new%20Worldwide%20Artificial%20Intelligence,2025%20period%20will%20be%2024.5%25.>

JACINDHA, S.; ABISHEK, G.; VASUKI, P. Program synthesis—a survey. In: **Computational Intelligence in Machine Learning**. [S.l.]: Springer, 2022. p. 409–421.

JADERBERG, M. et al. Human-level performance in 3d multiplayer games with population-based reinforcement learning. **Science**, American Association for the Advancement of Science, v. 364, n. 6443, p. 859–865, 2019.

JHA, S. et al. Oracle-guided component-based program synthesis. In: **Proceedings of the ACM/IEEE International Conference on Software Engineering**. [S.l.: s.n.], 2010. p. 215–224.

KITZELMANN, E. Inductive programming: A survey of program synthesis techniques. In: SPRINGER. **International workshop on approaches and applications of inductive programming**. [S.l.], 2009. p. 50–73.

KOSAR, T. et al. A preliminary study on various implementation approaches of domain-specific language. **Information and Software Technology**, v. 50, n. 5, p. 390–405, 2008. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584907000419>.

LANCTOT, M. et al. A unified game-theoretic approach to multiagent reinforcement learning. **Advances in neural information processing systems**, v. 30, 2017.

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

LELIS, L. H. S. Stratified strategy selection for unit control in real-time strategy games. In: **International Joint Conference on Artificial Intelligence**. [S.l.: s.n.], 2017. p. 3735–3741.

LENT, M. V.; FISHER, W.; MANCUSO, M. An explainable artificial intelligence system for small-unit tactical behavior. In: MENLO PARK, CA; CAMBRIDGE, MA; LONDON; AAAI PRESS; MIT PRESS; 1999. **Proceedings of the national conference on artificial intelligence**. [S.l.], 2004. p. 900–907.

LESLIE, D. S.; COLLINS, E. Generalised weakened fictitious play. **Games and Economic Behavior**, v. 56, n. 2, p. 285–298, 2006. ISSN 0899-8256. Disponível em: <https://www.sciencedirect.com/science/article/pii/S089982560500103X>.

LIANG, P.; JORDAN, M. I.; KLEIN, D. Learning programs: A hierarchical bayesian approach. In: **Proceedings of the International Conference on Machine Learning**. [S.l.]: Omnipress, 2010. p. 639–646.

MAJCHRZAK, K.; QUADFLIEG, J.; RUDOLPH, G. Advanced dynamic scripting for fighting game ai. In: SPRINGER. **International Conference on Entertainment Computing**. [S.l.], 2015. p. 86–99.

MANOHARAN, S. et al. An improved safety algorithm for artificial intelligence enabled processors in self driving cars. **Journal of artificial intelligence**, v. 1, n. 02, p. 95–104, 2019.

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

MARIÑO, J. R.; TOLEDO, C. Evolving interpretable strategies for zero-sum games. **Applied Soft Computing**, p. 108860, 2022. ISSN 1568-4946. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1568494622002496>.

MCMAHAN, H. B.; GORDON, G. J.; BLUM, A. Planning in the presence of cost functions controlled by an adversary. In: **Proceedings of the 20th International Conference on Machine Learning (ICML-03)**. [S.l.: s.n.], 2003. p. 536–543.

MEDEIROS, L. C.; ALEIXO, D. S.; LELIS, L. H. What can we learn even from the weakest? learning sketches for programmatic strategies. **arXiv preprint arXiv:2203.11912**, 2022.

MENON, A. et al. A machine learning framework for programming by example. In: **Proceedings of the International Conference on Machine Learning**. [S.l.]: PMLR, 2013. p. 187–195.

MORAES, R. O.; LELIS, L. H. S. Asymmetric action abstractions for multi-unit control in adversarial real-time scenarios. In: AAAI. **Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence**. [S.l.], 2018.

MURALI, V.; CHAUDHURI, S.; JERMAINE, C. Bayesian sketch learning for program synthesis. In: **Proceedings International Conference on Learning Representations**. [S.l.]: OpenReviews.net, 2017.

NASH, J. F. Equilibrium points in n-person games. **Proceedings of the national academy of sciences**, National Acad Sciences, v. 36, n. 1, p. 48–49, 1950.

NEUMANN, J. V.; MORGENSTERN, O. Theory of games and economic behavior, 2nd rev. Princeton university press, 1947.

NGUYEN, H. D. T. et al. Semfix: Program repair via semantic analysis. In: IEEE. **2013 35th International Conference on Software Engineering (ICSE)**. [S.l.], 2013. p. 772–781.

NORVIG, P.; RUSSELL, S. J. Inteligência artificial. **editora Campus**, 2004.

NUNEZ, C. Artificial intelligence and legal ethics: Whether ai lawyers can make ethical decisions. **Tul. J. Tech. & Intell. Prop.**, HeinOnline, v. 20, p. 189, 2017.

ONTAÑÓN, S. Combinatorial multi-armed bandits for real-time strategy games. **Journal of Artificial Intelligence Research**, v. 58, p. 665–702, 2017.

OSBORNE, M. J. Introduction to game theory. 2004.

PERELMAN, D. et al. Type-directed completion of partial expressions. In: **Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation**. [S.l.: s.n.], 2012. p. 275–286.

PETROSYAN, L. A.; ZENKEVICH, N. A. **Game theory**. [S.l.]: World Scientific, 2016. v. 3.

PNUELI, A.; ROSNER, R. On the synthesis of a reactive module. In: **Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages**. [S.l.: s.n.], 1989. p. 179–190.

POLOZOV, O. **A Framework for Mass-Market Inductive Program Synthesis**. Tese (Doutorado), 2017.

PONSEN, M.; SPRONCK, P. **Improving adaptive game AI with evolutionary learning**. Tese (Doutorado) — Citeseer, 2004.

PU, Y. et al. Selecting representative examples for program synthesis. In: DY, J.; KRAUSE, A. (Ed.). **Proceedings of the 35th International Conference on Machine Learning**. PMLR, 2018. (Proceedings of Machine Learning Research, v. 80), p. 4161–4170. Disponível em: <https://proceedings.mlr.press/v80/pu18b.html>.

ROBINSON, J. An iterative method of solving a game. **Annals of mathematics**, JSTOR, p. 296–301, 1951.

RUSSELL, S. J.; NORVIG, P. Artificial intelligence: a modern approach (international edition). {Pearson US Imports & PHIPEs}, 2002.

SILVER, D. et al. Mastering the game of go with deep neural networks and tree search. **nature**, Nature Publishing Group, v. 529, n. 7587, p. 484–489, 2016.

SILVER, D. et al. Mastering the game of go without human knowledge. **Nature**, Nature Publishing Group, v. 550, n. 7676, p. 354, 2017.

SPRONCK, P.; SPRINKHUIZEN-KUYPER, I.; POSTMA, E. Online adaptation of game opponent ai with dynamic scripting. **International Journal of Intelligent Games and Simulation**, v. 3, n. 1, p. 45–53, 2004.

SUMMERVILLE, A. et al. Understanding mario: an evaluation of design metrics for platformers. In: **Proceedings of the 12th international conference on the foundations of digital games**. [S.l.: s.n.], 2017. p. 1–10.

SZOLOVITS, P. **Artificial intelligence in medicine**. [S.l.]: Routledge, 2019.

VERMA, A. et al. Imitation-projected programmatic reinforcement learning. **Advances in Neural Information Processing Systems**, v. 32, 2019.

VERMA, A. et al. Programmatically interpretable reinforcement learning. In: DY, J.; KRAUSE, A. (Ed.). **Proceedings of the 35th International Conference on Machine Learning**. PMLR, 2018. (Proceedings of Machine Learning Research, v. 80), p. 5045–5054. Disponível em: <https://proceedings.mlr.press/v80/verma18a.html>.

VINYALS, O. et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. **Nature**, Nature Publishing Group, v. 575, n. 7782, p. 350–354, 2019.

VINYALS, O. et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. **Nature**, v. 575, n. 7782, p. 350–354, 2019.

ZHANG, T. et al. Interpretable program synthesis. In: **Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems**. [S.l.: s.n.], 2021. p. 1–16.

ZHANG, T. et al. Interactive program synthesis by augmented examples. In: **Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology**. [S.l.: s.n.], 2020. p. 627–648.

Appendix

APPENDIX A — Paper AAAI - Colaboration

Programmatic Strategies for Real-Time Strategy Games

Julian R. H. Mariño,[1] **Rubens O. Moraes,**[2] **Tassiana C. Oliveira,**[2] **Claudio Toledo,**[1] **Levi H. S. Lelis**[3]

[1] Departamento de Sistemas de Computação, ICMC, Universidade de São Paulo, Brazil
[2] Departamento de Informática, Universidade Federal de Viçosa, Brazil
[3] Department of Computing Science, Alberta Machine Intelligence Institute (Amii), University of Alberta, Canada
julianmarino@usp.br, rubens.moraes@ufv.br, tassiana.rios@ufv.br, claudio@icmc.usp.br, levi.lelis@ualberta.ca

Abstract

Search-based systems have shown to be effective for planning in zero-sum games. However, search-based approaches have important disadvantages. First, the decisions of search algorithms are mostly non-interpretable, which is problematic in domains where predictability and trust are desired such as commercial games. Second, the computational complexity of search-based algorithms might limit their applicability, especially in contexts where resources are shared among other tasks such as graphic rendering. In this work we introduce a system for synthesizing programmatic strategies for a real-time strategy (RTS) game. In contrast with search algorithms, programmatic strategies are more amenable to explanations and tend to be efficient, once the program is synthesized. Our system uses a novel algorithm for simplifying domain-specific languages (DSLs) and a local search algorithm that synthesizes programs with self play. We performed a user study where we enlisted four professional programmers to develop programmatic strategies for μRTS, a minimalist RTS game. Our results show that the programs synthesized by our approach can outperform search algorithms and be competitive with programs written by the programmers.

Introduction

Search and learning-based algorithms represent the current state-of-the-art approaches for playing zero-sum games, e.g., AlphaZero (Silver et al. 2018) and AlphaStar (Vinyals et al. 2019). One disadvantage of such approaches is that their decisions are often non-interpretable, which can be an issue if the artificial agent is deployed in scenarios where predictability, explainability, and trust are important, such as commercial games. Programmatic strategies, which we refer to as scripts, might not be as strong as strategies derived by search or reinforcement learning algorithms. However, scripts can more easily be interpreted and modified by a domain expert. The computer games industry heavily relies on scripts for controlling artificial agents because game designers and programmers are able to understand, predict, and thus trust the agent behavior in production. In the industry, scripted strategies are written by professional programmers in a trial-and-error process as they try to understand how other agents (including the player) could react to the

strategy that is encoded in a script. The process of manually writing scripts can be time consuming. Moreover, strategies written by programmers are fixed and, if a player creates new content for a game (e.g., a new map for playing a real-time strategy game), then the artificial agents might have to play a strategy encoded in a script developed for a different map.

In this paper we introduce Local Search and Language Simplifier (LS2), a system for synthesizing scripts for real-time strategy (RTS) games. The inputs of LS2 are an RTS map for which a strategy is to be synthesized and a domain-specific language (DSL). LS2 returns a script encoding a strategy that is specialized for the map provided as input.

The DSL is designed to be expressive enough to allow for the synthesis of scripts encoding effective strategies, but also restrictive enough to prune from the synthesis space programs encoding weak strategies. Since strategies can change depending on the game map, features that make a DSL effective for a given map might not be necessary for another map. LS2 uses an algorithm we call Domain-Specific Language Simplifier (Lasi) to simplify the DSL according to the map provided as input. Lasi receives as input a trace of the game, i.e., a sequence of state-action pairs starting at the game's start state and finishing at a terminal state. This trace can be generated, for example, by an algorithm playing the game against itself or by a human demonstrator. Then, Lasi greedily chooses the features from the input DSL that allows one to synthesize a program that reproduces the trace provided as input. The features selected by this greedy approach define the simplified DSL. The intuition behind Lasi's procedure is that the DSL should contain only the features deemed as important by the agent who generated the game trace, thus allowing the synthesis to search on a more promising program space. Once the DSL is simplified LS2 uses a local search algorithm with self play to search over the simplified program space.

We evaluate LS2 on four maps of μRTS, a real-time strategy (RTS) game designed for research purposes. We enlisted four professional programmers to write scripts for all four maps used in our experiment. In addition to the scripts written by programmers, we also compare the LS2's scripts with search algorithms and other scripts from the μRTS codebase in a tournament-style experiment. A script LS2 synthesized obtained the highest winning rate in two of the maps tested and a script written by one of the programmers obtained the

highest winning rate in the other two maps. In addition to this quantitative analysis, we also performed a qualitative analysis of the interpretability of the LS2 scripts, showing that the scripts synthesized in our experiments can be interpretable. Our analysis also showed that LS2 was able to synthesize scripts encoding strategies similar to those created by the programmers, but with important optimizations that would be difficult for a human to discover by themself.

Related Work

Our work is related to the fields of program synthesis, programmatic reinforcement learning, automatic generation of scripts, and planning in real-time zero-sum domains. We review relevant works in each of these areas in this section.

In program synthesis one has to synthesize a program that satisfies a given specification (Gulwani, Polozov, and Singh 2017). Approaches for program synthesis include brute-force search (Alur et al. 2013), constraint satisfaction (Jha et al. 2010), genetic programming (Koza 1992), machine learning (Balog et al. 2017), and hybrid systems that combine search with a learned function (Liang, Jordan, and Klein 2010; Menon et al. 2013; Murali, Chaudhuri, and Jermaine 2017). All these works assume some form of supervision, which could be a logical formula that needs to be satisfied or a set of input-output training pairs. Our problem is not supervised, the synthesis algorithm has access to the model of the game and it has to synthesize a program encoding a strategy for playing the game.

In programmatic reinforcement learning (PRL) one represents policies as computer programs (Bastani, Pu, and Solar-Lezama 2018; Verma et al. 2018, 2019), which might be more easily interpreted and formally verified than policies encoded in black-box representations. Instead of learning policies as was done in PRL, in our work we learn strategies for playing games. Moreover, it is possible that the idea we introduce for simplifying DSLs might also be applicable to speed up the program synthesis process in PRL.

Dynamic Scripting (DS) is a reinforcement-learning-based technique for synthesizing scripts for zero-sum role-playing games. DS allows for the generation of scripts by extracting rules from an expert-designed rule base according to a learned policy (Spronck, Sprinkhuizen-Kuyper, and Postma 2004). DS has been applied to RTS games, but the synthesized script is limited to a fixed number of consecutive if-then clauses. Our method is more expressive as it searches in the space of scripts defined by a DSL that allows for other program structures (e.g., for loops).

Program synthesis has been applied to zero-sum games in the context of synthesizing scripts to work as evaluation functions (Benbassat and Sipper 2011) and as pruning policies (Benbassat and Sipper 2012) for tree search algorithms. By contrast, in this paper we investigate the use of script synthesis for deriving complete strategies for zero-sum games.

Program synthesis has also been applied to other non-zero-sum games. Canaan et al. (2018) use an evolutionary approach for generating strategies to play a cooperative game. Similarly to DS, Canaan et al.'s method generates sequences of if-then rules to play the game.

De Freitas, de Souza, and Bernardino (2018) use a genetic-programming approach for evolving controllers for a Mario AI simulator. Butler, Torlak, and Popović (2017) present a method for synthesizing strategies for the puzzle game nonograms. De Freitas, de Souza, and Bernardino, and Butler, Torlak, and Popović's approaches are different than our method because they deal with single-agent problems, while we deal with two-players zero-sum games.

The Script Synthesis Problem

Let $\mathcal{G}$ be a zero-sum game, i and $-i$ the pair of players for $\mathcal{G}$, $\mathcal{S}$ the set of states of the game, s_{init} the start state and $\mathcal{A}_i(s)$ the set of actions player i can perform at state s. A decision point for player i is a state $s \in \mathcal{S}$ where i can act. A strategy is a function $\sigma_i : \mathcal{S} \rightarrow \mathcal{A}_i$ for player i, mapping a state s to an action a, for every decision point of player i in $\mathcal{G}$. A script is a strategy denoted as a function $p(s)$ that returns a legal action $a \in \mathcal{A}_i$ for player i at state s. The value of the game rooted at state s is denoted by $\mathcal{V}(s, p_i, p_{-i})$, which indicates player i's utility if i and $-i$ follow the strategies given by p_i and p_{-i}, respectively. Since $\mathcal{G}$ is a zero-sum game, the utility of player $-i$ is given by $-\mathcal{V}(s, p_i, p_{-i})$.

A Domain-Specific Language (DSL) is a declarative language that defines a space of programs in a particular domain (Van Deursen, Klint, and Visser 2000). Let $[\![D]\!]$ be the set of scripts that can be synthesized with a DSL D. We are interested in synthesizing a script $p_i \in [\![D]\!]$ for player i that maximizes the value of the game while player $-i$ plays the game with a script p_{-i} from $[\![D]\!]$ that minimizes the value of the game. We formulate the script synthesis problem as,

$$\max_{p_i \in [\![D]\!]} \min_{p_{-i} \in [\![D]\!]} \mathcal{V}(s_{init}, p_i, p_{-i}). \tag{1}$$

The strategies encoded by scripts p_i and p_{-i} that solve the Equation 1 define a Nash equilibrium profile if one considers only strategies encoded by scripts in $[\![D]\!]$ as valid strategies. Searching for a Nash profile can be computationally intractable, specially if D allows for the synthesis of a large set of scripts $[\![D]\!]$. LS2 uses a self play procedure with local search for approximating a solution to Equation 1.

Domain-Specific Languages

We define a DSL to synthesize scripts for zero-sum games through a context-free grammar (CFG) $G = (V, \Sigma, R, S)$, where V is a finite set of non-terminals, Σ is a finite set of terminals, R is a finite set of relations corresponding to the grammar production rules, and S is the start symbol.

As an example, the grammar G_1 below defines a DSL.

$$S \rightarrow C \mid \text{if}(B) \text{ then } C \text{ else } C \mid C \text{ if}(B) \text{ then } C$$
$$C \rightarrow c_1 \mid c_2 \mid c_3$$
$$B \rightarrow b_1 \mid b_2 \mid b_3$$

Here, $V = \{S, C, B\}$, $\Sigma = \{c_1, c_2, c_3, b_1, b_2, b_3, \text{if}, \text{then}, \text{else}\}$, R is the set of relations, (e.g., $C \rightarrow c_1$ and $B \rightarrow b_1$), and S is the start symbol. G_1 allows scripts with a single command (c_1, c_2, or c_3), scripts with an if-then-else, or scripts with a single command followed by an if-then. We represent the scripts as derivation trees, where the root
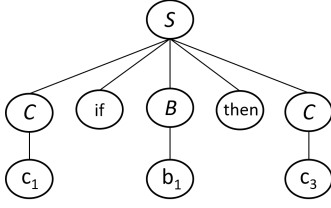
Figure 1: Derivation tree for "c1 if (b1) then c3".

node in the tree is the start symbol S, the internal nodes are non-terminals from V, and leaf nodes are terminals from Σ. Figure 1 shows an example of a derivation tree.

Domain-Specific Language Simplifier

The size of the space of possible programs to solve the script synthesis problem can grow quickly with the size of the CFG defining the DSL (i.e., the number of relations and symbols). The search for effective strategies can become infeasible for large grammars. On the other hand, if the grammar is too constrained, then synthesized programs might not be expressive enough to encode strong game strategies. In this section we introduce Lasi, a method that simplifies DSLs while balancing the language size and expressiveness for a given task. LS2 uses Lasi before running a self play procedure with local search to approximate a solution to Equation 1.

Lasi receives a game $\mathcal{G}$, a grammar $G = (V, \Sigma, R, S)$ defining a DSL, and a sequence of state-action pairs, $T = \{(s_{init}, a_1), (s_2, a_2), \cdots, (s_n, a_n)\}$, which we call a trace, from s_{init} to a terminal state $s_{n+1} \in \mathcal{F}$. The trace can be generated by an agent playing the game (e.g., tree search algorithm or human demonstrator). Lasi selects a subset of Σ to define a grammar G' that is more restrictive than G, but that still allows a program synthesized with a DSL defined by G' to reproduce the trace T received as input. Intuitively, the system designer should provide an expressive grammar G as input and Lasi automatically produces a more restrictive grammar G'. We describe how Lasi defines grammar G' in the next section.

The DSL Simplification Problem

It is common in a program synthesis task that the system designer defines a set of high-level functions to be part of the DSL. We will call these functions domain-specific functions (DSFs). In our application domain, a DSF returns an action for a given state of the game. For example, if grammar G_1 is used in a game played in a grid-world, then command c_1 could encode the knowledge to allow the agent to walk out of a room through the exit door. That is, for any state of the game, c_1 returns an action that takes the agent closer to leaving the room through the exit door. Similarly, terminal symbols representing Boolean expressions can be defined as domain-specific Boolean functions (DSBs). DSBs return a Boolean value for a given state of the game. For example, in G_1 the terminal symbol b_1 could be a DSB that returns true if the agent is waiting by the exit door and false otherwise. We formulate the DSL Simplification Problem as follows,

Definition 1 (DSL Simplification Problem) *Let $[\![G]\!]$ be the set of programs that can be synthesized with grammar $G = (V, \Sigma, R, S)$ and T be a trace with state-action pairs. In the DSL simplification problem one has to find a grammar $G' = (V, \Sigma', R', S)$ with the smallest set Σ', such that $\Sigma' \subseteq \Sigma$ and $R' \subseteq R$, for which there exists a script $p \in [\![G']\!]$ such that $p(s_m) = a_m$ for all $(s_m, a_m) \in T$.*

In a DSL Simplification Problem we assume that all terminals removed from Σ (i.e., $\Sigma \setminus \Sigma'$) are either DSFs or DSBs. That is, the grammar can include non-DSFs and non-DSBs terminals, but they are not be considered for removal in the simplification task. We also assume that G is expressive enough so that there exists a script $p \in [\![G]\!]$ with $p(s_m) = a_m$ for all $(s_m, a_m) \in T$, otherwise the simplification task does not have a solution.

Simplification Problem as Set Cover

Subset R' of R can be trivially computed once subset Σ' of Σ is defined. This is because terminal symbols appear only on the righthand side of the relations and, for that, subset R' is the set R with the relations involving symbols $\Sigma \setminus \Sigma'$ removed. Thus, the task of simplifying G into G' is equivalent to selecting a subset Σ' of Σ.

Removing DSFs We start with the selection of terminal symbols defined by DSFs. Each terminal symbol represented as a DSF in the grammar G provided as input to Lasi returns an action $a \in \mathcal{A}(s)$ for any state s in the trace T. The problem of selecting DSFs can be seen as a set cover problem, where each DSF represents a subset of the actions in the trace T and one needs to find the smallest subset of DSFs that covers all actions in T.

While the set cover problem is NP-hard (Garey and Johnson 1979), a polynomial-time greedy algorithm provides a good approximation. Let Q be the set of state-action pairs (s, a) initialized with all (s, a) in T. Let Σ' be the set of DSFs selected by the greedy algorithm, which is initially empty. One iteratively adds to Σ' a DSF o that covers the largest number of actions in Q, i.e., the DSF that maximizes $|\{(s, a)|(s, a) \in Q \land o(s) = a\}|$. We remove from Q the state-action pairs in $\{(s, a)|(s, a) \in Q \land o(s) = a\}$ for the selected o. This procedure is repeated until all actions in T are covered by a DSF in Σ'.

In the worst case all symbols in Σ are DSFs and we select all of them to ensure action coverage, i.e., $\Sigma = \Sigma'$. In this case the algorithm's time complexity is $O(|\Sigma|^2 \cdot |T|)$. This is because each iteration of the algorithm has complexity of $O(|\Sigma| \cdot |T|)$ (each DSF in Σ is tested for its coverage) and the algorithm performs $|\Sigma|$ iterations in the worst case. The performance ratio of the solution encountered by the greedy algorithm is $\ln(|\Sigma|) - \ln(\ln(|\Sigma|)) - \Theta(1)$ (Slavík 1996).

Removing DSBs Lasi adds to Σ' all DSBs from Σ but those that return either true or false for all states in T, i.e., it does not add the DSBs in $\{b|b \in \Sigma \land b(s) = \text{false } \forall s \in T\}$ nor the DSBs in $\{b|b \in \Sigma \land b(s) = \text{true } \forall s \in T\}$. Let b be a DSB that returns false for all states in T. Any program using b can be rewritten without b. That is, the commands if(b){c1}, if(not b){c1}, while(b){c1},

383

`while(not b){c1}`, and `var ← b` can be replaced by
ϵ, `c1`, ϵ, `while(True){c1}`, and `var ← false`, re-
spectively, where ϵ is an empty string. Similarly, all pro-
grams using DSBs that always return true can be replaced by
equivalent programs that do not use the DSBs. The DSL only
needs DSBs b for which $b(s)$ returns true for some states and
false for other states in the trace T to be able to synthesize
a program that reproduces T. Lasi runs in $O(|\Sigma| \cdot |T|)$ for
removing DSBs because each terminal in Σ might be a DSB
that needs to be verified against all states in T.

Synthesizing Scripts with Self Play

Once Lasi simplifies the DSL, LS2 uses the self play pro-
cedure described in Algorithm 1 to synthesize a script for
game $\mathcal{G}$. Algorithm 1 starts by generating a random script
from the DSL D (see line 1). This is achieved by starting
from the D's initial symbol and selecting rules uniformly at
random to be applied to non-terminal symbols; a script is
returned once it contains no non-terminal symbols.

Algorithm 1 iteratively improves the initial random solu-
tion p with a local search algorithm. The local search re-
ceives the game $\mathcal{G}$, the DSL D, and current script p, and re-
turns an approximated best response for p, denoted as p_{BR}.
The script p_{BR} is then attributed to p if p_{BR} defeats p. In
the context of μRTS, to ensure fairness, p and p_{BR} play two
matches, one with p as player 1 and another with p_{BR} as
player 1. We consider that p_{BR} defeats p if it either wins
both matches or if it wins one match and draws the other. In
the next iteration, the local search will approximate a best
response to the current's iteration best response.

In our implementation of Algorithm 1 we use a local
search algorithm that creates m mutated versions of the
script p provided as input and generates i other scripts at
random. All $m + i$ newly generated scripts are evaluated
with two matches against p, where we vary which script as-
sumes the role of player 1 for fairness; the best performing
script is returned as an approximated best response to p. The
best performing script is determined by a score function that
attributes the value of 1 to a victory, 0 to a loss, and 0.5 to
a draw; we break ties at random. A script p is mutated by
randomly selecting a node representing a non-terminal sym-
bol in the derivation tree representing p and replacing the
sub-tree rooted at the selected node by a sub-tree randomly
generated according to the rules of the CFG describing the
DSL. For example, if node B in the tree shown in Figure 1 is
selected for mutation, then the sub-tree b_1 would be replaced
by another sub-tree (b_1, b_2, or b_3 in our example). If the root
of the tree is selected, then the entire tree is replaced.

Empirical Methodology

We evaluate LS2's scripts in μRTS (Ontañón 2017) by com-
paring them with tree search algorithms and scripts written
by programmers. We are primarily interested in evaluating
the effect of Lasi in LS2 and in comparing the scripts written
by professional designers with those LS2 synthesizes. We
present a quantitative evaluation of all approaches in terms
of strength of play, and a qualitative evaluation of the inter-
pretability of the synthesized scripts. Our implementation of

---

Algorithm 1 Self play with local search

Require: Game $\mathcal{G}$, DSL D, number of steps n.
Ensure: Script p for playing the game $\mathcal{G}$.
 1: $p \leftarrow$ random-script(D)
 2: **for** $k = 1$ to n **do**
 3:     $p_{BR} \leftarrow$ local-search($\mathcal{G}$, D, p)
 4:     **if** p_{BR} defeats p in $\mathcal{G}$ **then**
 5:         $p \leftarrow p_{BR}$
 6: **return** p

LS2 is available online.[1]

Problem Domain: μRTS

We chose to use μRTS in our experiments because it has
an active research community with competitions being orga-
nized (Ontañón et al. 2018), with all competing algorithms
available in a single codebase.[2] Moreover, in 2017 a script
won two tracks of the μRTS competition, cf. Table 1 of
(Ontañón et al. 2018), demonstrating that scripts can outper-
form other approaches for this type of game. Finally, μRTS
can offer a diverse set of challenges because the agents can
be easily evaluated on different maps of the game.

Most μRTS matches start with each player controlling a
set of units known as workers on a gridded map. Workers
can be used to collect resources, build structures, and bat-
tle the opponent. In some of the maps, players also start
with a structure called base, which is used to train work-
ers and store resources. In addition to the base, workers can
build a barracks, which can be used to train the following
units: light, ranged, and heavy. Units differ in how much
damage they can take before being removed from the game,
how much damage they can inflict to other units, and how
close they need to be from an opponent unit to be able to
attack it. A player wins a match if they are able to remove
all the other player's units from the game. Every algorithm
is allowed 100 milliseconds for planning before deciding the
units actions. We call a unit everything that can issue an ac-
tion, including bases and barracks.

We used four μRTS maps in our evaluations, with a map
of size 18×8 being novel. This map contains no worker
units neither resources to be harvested, and has three heavy,
four ranged units, and a base. The ranged units are initially
placed in front of the heavy units and closer to the oppo-
nent units, which we hypothesized to be suboptimal as the
ranged units are able to inflict damage to enemy units from
far, but they are weaker than heavy units and can be quickly
eliminated from the game. We expected to see strategies in
which ranged and heavy units switch positions so that the
latter protect the former. In the second map, of size 8×8,
each player starts with a base and one worker. The third map
is a map of size 9×8 where players start with a base, a
worker, and the resources are placed as a wall separating the
players. The fourth map, of size 24×24, is divided in the
middle by a wall, and players start with two bases and four

---

[1]https://github.com/julianmarino/LS2
[2]https://github.com/santiontanon/microrts

workers, one base placed on each side of the map. The maps 8×8, 9×8, and 24×24 were used in μRTS competitions.

Generating Trace for Simplifying the DSL

In our experiments we use A3N (Moraes et al. 2018) to generate Lasi's required demonstration trace by having it play a match with itself. We chose A3N because Moraes et al. showed that it performs well in a variety of maps. We could have also used other search algorithms such as Naive Guided Sample (GNS) (Yang and Ontanón 2019) or human demonstrations to produce the trace.

A3N requires a set of scripts as input and, for a subset of the units controlled by the player (known as restricted units), it considers only the actions returned by the scripts; all other actions are disregarded during search. A3N accounts for all legal actions of the remaining units (known as unrestricted units). A3N uses a policy to decide which units are unrestricted in each state of the game. Since we would like to explore the action space, we use a policy that randomly chooses three unrestricted units in each state. The size of the unrestricted set was chosen empirically in preliminary experiments.

Since A3N is being used in the context of script synthesis, we do not provide expert-designed scripts as input to A3N, as is described in its original paper (Moraes et al. 2018). Instead, we generate a random script that obeys our DSL and provide it as input to A3N. This randomly synthesized script defines the action of the restricted units. Naturally, the quality of the actions returned by A3N depends on the quality of the set of scripts provided as input and the use of a randomly synthesized script reduces its strength of play. We compensate for this reduction in quality by allowing A3N more planning time. Instead of 100 ms, we allow A3N 500 ms of planning time in the self play match that generates T.

Competing Agents

We use the following search algorithms: Portfolio Greedy Search (PGS) (Churchill and Buro 2013), Stratified Strategy Selection (SSS) (Lelis 2017), the MCTS version of Puppet Search (PS) (Barriga, Stanescu, and Buro 2017b), Strategy Tactics (STT) (Barriga, Stanescu, and Buro 2017a), NaïveMCTS (NS) (Ontañón 2017), A3N (Moraes et al. 2018), and Naive Guided Sample (GNS) (Yang and Ontanón 2019). We also use the scripts Worker Rush (WR), Light Rush (LR), Heavy Rush (HR), and Ranged Rush (RR) (Stanescu et al. 2016). These scripts train one worker to collect resources and then continuously train one type of unit that is sent to battle to opponent. WR, LR, HR, and RR train workers, light, heavy, and ranged units, respectively. Although simple, these scripts were shown to perform well in a wide range of maps. We also experiment with a baseline of LS2, denoted as LS, that does not employ Lasi. That is, LS uses the self play algorithm with local search to synthesize a script while using the original DSL provided as input.

μLanguage: a Domain-Specific Language for μRTS

We have developed a DSL for μRTS, which we name μLanguage, to allow for the development of scripts in μRTS.

Both users in our study and synthesizers use the same DSL. The CFG below summarizes the μLanguage.

$$S_1 \rightarrow C\,S_1 \mid S_2\,S_1 \mid S_3\,S_1 \mid \epsilon$$
$$S_2 \rightarrow \text{if}(B) \text{ then } \{C\} \mid \text{if}(B) \text{ then } \{C\} \text{ else } \{C\}$$
$$S_3 \rightarrow \text{for (each unit } u)\,\{S_4\}$$
$$S_4 \rightarrow C\,S_4 \mid S_2\,S_4 \mid \epsilon$$
$$B \rightarrow b_1 \mid b_2 \mid \cdots \mid b_m$$
$$C \rightarrow c_1\,C \mid c_2\,C \mid \cdots \mid c_n\,C \mid c_1 \mid c_2 \mid \cdots \mid c_n \mid \epsilon$$

S_1 is the initial symbol, ϵ is an empty string, and symbols $c_1, c_2, \cdots, c_n$ are DSFs and $b_1, b_2, \cdots, b_n$ are DSBs. μLanguage does not allow nested conditionals and nested loops, but it allows programs with if-clauses inside and outside for-loops and infinitely large sequences of DSFs.

Scripts Written by Programmers

We have enlisted four professional programmers, who are not involved in this research, to write one script for each of the four maps used in our experiments. All programmers used the same DSL we provide as input to LS2, so both system and programmers had access to the same space of programs. The experiment was carried out online, advertised by email, with the participation being anonymous. Each participant watched a 10-minute video explaining the rules of μRTS and showing a few examples of μLanguage, and also had access to a manual describing each DSF and DSB implemented in the μLanguage. The participants were allowed as much time as they wanted to develop their scripts. After the participants developed the four scripts they answered a questionnaire. Our study had 3 males and 1 female participants, with an average age of 26.5 years. The average years of programming experience was 5.5 years. All participants had taken at least one course on artificial intelligence and three participants had played an RTS game at least one time, the average minutes spent to write all four scripts was 66.25 minutes. We will refer to the scripts written by the programmers as P_1, P_2, P_3, and P_4 in our table of results.

Empirical Results

Evaluation of Strength of Play

Our experiment consists in performing ten independent runs of LS and LS2 for each map. We use $n = 400$ for Algorithm 1, and $m = 70$ and $i = 20$ for our local search algorithm. Each run takes approximately 30 minutes for maps of sizes 18×8, 8×8, and 9×8, and 7 hours for the 24×24 map. We use one machine with 56 cores for our experiments.

We determine the script returned by LS and LS2 by performing a round-robin tournament among the scripts returned in each run of LS and LS2. The purpose of the tournament is to select the best solution encountered by LS and LS2 in the ten independent runs. In our round-robin tournaments, each strategy plays against all the other strategies ten times, five as player 1 and five as player 2. This is to ensure fairness in μRTS. The time required to generate one trace with A3N is negligible compared to the time required to perform the 10 independent runs of the local search algorithm and the round-robin tournament to select the best

Map 8 × 8								
	WR	GNS	P$_3$	P$_1$	P$_4$	P$_2$	LS	LS2	Avg
STT	30.0	0.0	15.0	55.0	30.0	20.0	35.0	6.0	54.1
NS	20.0	30.0	40.0	20.0	10.0	10.0	51.0	6.0	56.4
A3N	30.0	10.0	70.0	50.0	30.0	15.0	69.0	30.0	62.8
WR	-	30.0	100.0	50.0	50.0	75.0	60.0	25.0	75.6
GNS	70.0	-	65.0	60.0	30.0	90.0	67.0	69.0	81.9
P$_3$	0.0	35.0	-	25.0	25.0	50.0	45.0	11.0	55.1
P$_1$	50.0	40.0	75.0	-	50.0	0.0	5.0	0.0	62.2
P$_4$	50.0	70.0	75.0	50.0	-	50.0	65.0	20.0	75.6
P$_2$	25.0	10.0	50.0	100.0	50.0	-	70.0	64.0	76.5
LS	40.0	33.0	55.0	95.0	35.0	30.0	-	26.0	66.2
LS2	75.0	31.0	89.0	100.0	80.0	36.0	74.0	-	83.9

Map 9 × 8								
	A3N	RR	P$_4$	P$_1$	P$_3$	P$_2$	LS	LS2	Avg
LR	0.0	0.0	100.0	0.0	0.0	0.0	74.0	59.0	46.1
PS	0.0	0.0	100.0	35.0	0.0	0.0	82.0	73.0	54.4
GNS	60.0	60.0	85.0	65.0	20.0	0.0	86.0	81.0	70.4
A3N	-	20.0	100.0	80.0	40.0	10.0	100.0	92.0	79.5
RR	80.0	-	100.0	75.0	0.0	0.0	100.0	95.0	80.6
P$_4$	0.0	0.0	-	0.0	0.0	0.0	36.0	40.0	16.9
P$_1$	20.0	25.0	100.0	-	15.0	0.0	83.0	47.0	62.8
P$_3$	60.0	100.0	100.0	85.0	-	0.0	100.0	99.0	89.0
P$_2$	90.0	100.0	100.0	100.0	100.0	-	100.0	100.0	99.4
LS	0.0	0.0	64.0	17.0	0.0	0.0	-	41.0	27.9
LS2	8.0	5.0	60.0	53.0	1.0	0.0	59.0	-	39.4

Map 18 × 8								
	NS	A3N	P$_1$	P$_4$	P$_3$	P$_2$	LS	LS2	Avg
HR	40.0	45.0	50.0	50.0	0.0	0.0	20.0	10.0	37.8
PS	20.0	35.0	50.0	50.0	50.0	50.0	25.0	20.0	42.8
STT	50.0	30.0	65.0	45.0	75.0	40.0	49.0	56.0	52.5
GNS	25.0	45.0	65.0	55.0	55.0	25.0	53.0	58.0	53.8
NS	-	65.0	70.0	70.0	65.0	65.0	68.0	70.0	68.6
A3N	35.0	-	75.0	90.0	85.0	50.0	82.0	75.0	70.4
P$_1$	30.0	25.0	-	50.0	0.0	0.0	20.0	10.0	34.7
P$_4$	30.0	10.0	50.0	-	0.0	0.0	20.0	10.0	35.6
P$_3$	35.0	15.0	100.0	100.0	-	25.0	80.0	70.0	71.6
P$_2$	35.0	50.0	100.0	100.0	75.0	-	75.0	65.0	80.3
LS	32.0	18.0	80.0	80.0	20.0	25.0	-	10.0	57.4
LS2	30.0	25.0	90.0	90.0	30.0	35.0	90.0	-	68.5

Map 24 × 24								
	WR	GNS	P$_3$	P$_1$	P$_2$	P$_4$	LS	LS2	Avg
STT	50.0	0.0	95.0	90.0	35.0	20.0	47.0	6.0	50.2
A3N	15.0	0.0	80.0	50.0	30.0	35.0	40.0	17.0	52.6
PS	45.0	35.0	80.0	65.0	50.0	45.0	44.0	32.0	54.8
WR	-	90.0	0.0	100.0	100.0	0.0	45.0	20.0	69.1
GNS	10.0	-	100.0	5.0	65.0	85.0	39.0	21.0	69.4
P$_3$	100.0	0.0	-	100.0	0.0	20.0	62.0	50.0	38.9
P$_1$	0.0	95.0	0.0	-	65.0	0.0	0.0	0.0	45.9
P$_2$	0.0	35.0	100.0	35.0	-	50.0	42.0	11.0	63.0
P$_4$	100.0	15.0	80.0	100.0	50.0	-	50.0	62.0	74.8
LS	55.0	61.0	38.0	100.0	58.0	50.0	-	50.0	67.4
LS2	80.0	79.0	50.0	100.0	89.0	38.0	50.0	-	82.6

Table 1: Average winning rate of the row player against the column player. The average winning rate of row players was computed considering matches played with all methods used in our experiment, and not only those shown in the table.

synthesized script. Once scripts of LS and LS2 are determined, we perform another round robin tournament with the synthesized scripts, tree search algorithms, and the scripts written by the programmers in our study.

We execute the experiment described above ten times and calculate the average winning rate considering the ten executions. The results are presented in Table 1, where each number shows the average winning rate of the row method against the column method (e.g., LS2 wins 89% of its matches against P$_3$ on the 8×8 map). The last column shows the average winning rate of the row methods. The table presents LS2, LS and the scripts the programmers wrote. In the interest of space, we present the best five and the best two methods among the remaining ones, in terms of average winning rate, in the rows and in the columns of the table.

LS2 achieves a much higher average winning rate than its baseline LS and wins more direct matches in all maps tested. This result shows that LS2's simplification scheme given by Lasi allows for a more focused search in the scripts space. LS2 achieves the highest average winning rate in the 8×8 and 24×24 maps. In the 9×8 and 18×8 maps, the highest average value is obtained by P$_2$. In the 18×8 map, LS2 was also outperformed by P$_3$ and by the search-based methods A3N and NS. A3N and NS can be very effective in the map 18×8 because they are able to micromanage well their units.

LS2 outperforms A3N in terms of average winning rate on the 8×8 and 24×24 maps. This result suggests that although the synthesis process is being constrained to a region of the scripts space containing strategies similar to the ones played by A3N, the script synthesizer is able to encounter stronger strategies than the ones derived by A3N in its regular setting (i.e., the one described by Moraes et al. (2018), which was used to generate the results of Table 1).

Table 2 shows the average winning rate of the methods evaluated across all four maps. The scripts synthesized by LS2 are outperformed only by programmer P$_2$, and performs similarly to the tree search algorithm GNS. These results are promising as they show that it is possible to automatically synthesize programmatic strategies for playing non-trivial games that are able to defeat scripts written by programmers and tree search algorithms. The table also highlights the importance of Lasi in the synthesis process as LS obtained only a 54.7% average winning rate across all maps.

Interpretability of Scripts

Table 3 shows the best scripts synthesized by LS2 and the best scripts written by programmers in terms of average winning rate for the 8×8 and 24×24 maps. The objective of this analysis is two-fold. First, we want to verify if the programmatic strategies LS2 synthesizes can be interpretable. Second, we want to perform a qualitative comparison of the LS2 scripts with those written by the programmers.

The script P$_2$ follows an strategy similar to the WR strategy. P$_2$ sends one worker to harvest resources (line 19) and the remaining workers are sent to attack the closest enemy unit (line 21). P$_2$ continuously trains workers (line 20). This strategy has shown to be effective in previous μRTS competitions. The script LS2 synthesized for map of size 8×8 (`LS2-8x8`) iterates through all units the player controls

Average Winning Rate on All Maps																
HR	SSS	PGS	LR	RR	NS	PS	STT	WR	P_4	P_1	LS	P_3	A3N	LS2	GNS	P_2
26.7	28.6	29.6	36.2	38.5	44.1	44.7	47.6	49.9	50.8	51.4	54.7	63.6	66.3	68.6	68.9	79.8

Table 2: Average winning rate across all four maps tested.

```
1  def LS2-8x8()
2      for(each unit u)
3          if(DistanceEnemy(u, worker, 5))
4              attack(u, worker, closest)
5          else
6              harvest(u)
7          train(u, worker, Right)
8      harvest(u)
9
10 def LS2-24x24()
11     for(each unit u)
12         if(HaveQtdUnitsHarvesting(2))
13             attack(u, worker, closest)
14             train(u, worker, right)
15         else
16             harvest(u)
17
18 def P_2-8x8()
19     harvest(1)
20     train(worker, enemyDirection)
21     attack(worker, closest)
22
23 def P_4-24x24()
24     train(worker, up)
25     harvest(5)
26     attack(worker, closest)
```

Table 3: Scripts synthesized by LS2 and the best two scripts written by the programmers for maps 18×8 and 24×24.

and, if a unit u is at a distance of 5 or less from an enemy unit, u is sent to attack the closest opponent unit (lines 3–4). If no enemy unit is nearby, the script will send workers to harvest resources (line 6). The strategy continuously train worker units (line 7). LS2's script has a seemingly unnecessary instruction in line 8 ("harvest(u)"). This is because, once we reach line 8, all units will have received an action to be performed in the game. However, if we remove line 8, the winning rate of this specific LS2 script is reduced from 50% to 0% in matches against P_3 (not shown in Table 1). We discovered that this extra instruction optimizes the pathfinding system of μRTS. That is, the pathfinding algorithm might fail to find a path for a unit because the path is momentarily blocked and the unit stays idle despite the script having issued an action for the unit. In these cases it is helpful to add an extra action for the units. In LS2's script, if worker units are unable to attack the enemy, then the units will move toward their resources, as indicated by the instruction "harvest," possibly clearing the path for the next round

of actions. This is a case where LS2 is able to synthesize a script that contains a feature that would be difficult for human programmers to discover by themselves.

For the 24×24 map, script P_4 encodes a strategy that is similar to the strategy LS2 synthesized: both strategies train a large number of worker units and send them to attack. A key difference between the two scripts is the number of units used to collect resources. While P_4 uses five units to collect resources (line 25), the LS2 script uses two units (lines 12 and 13). This is also a case where LS2 is able to synthesize a script that contains a feature that might be difficult for human programmers to optimize by themselves.

The scripts synthesized by LS2 can be improved too. For example, in the 9×8 map, LS2 is able to synthesize a strategy that, similarly to P_2, trains ranged units and sends them to attack the opponent. One of the drawbacks of the LS2 strategy is that it also sends worker units to continuously build barracks, which exhaust almost all resources available to the player. The script also trains a large number of workers that clutter the region around the base, thus making it difficult for the workers themselves to move around and collect resources. The result on the map 9×8 suggests that better search algorithms could further improve the quality of the scripts LS2 synthesizes.

Conclusions

In this paper we introduced LS2, a system for synthesizing scripts for RTS games. LS2 employs Lasi, an algorithm for simplifying DSLs for RTS games. Lasi uses A3N to play the game against itself to generate a game trace. Then, Lasi greedily chooses the features from the DSL that allows one to synthesize a program that reproduces the trace generated by the search algorithm. The features selected by this greedy procedure define the simplified DSL. LS2 then uses a local search algorithm with self play to search in the space of programs defined by the simplified DSL. We evaluated the scripts synthesized by LS2 on μRTS by comparing the synthesized scripts with tree search algorithms, scripts written by four programmers, and scripts synthesized by a baseline that does not use the DSL simplification step. Our results show that the LS2 scripts were able to outperform the scripts synthesized by the baseline by a large margin. The LS2 scripts obtained the highest winning rate on two of the four tested maps and a script written by a programmer obtained the highest winning rate on the other two maps. A qualitative analysis of the scripts showed that LS2 was able to synthesize scripts encoding strategies similar to those derived by the programmers. The qualitative analysis also showed that LS2 was able to synthesize scripts with important optimizations that would be difficult for a human to discover by themself.

Acknowledgements

References

Alur, R.; Bodik, 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 *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design*, 1–17.

Balog, M.; Gaunt, A. L.; Brockschmidt, M.; Nowozin, S.; and Tarlow, D. 2017. DeepCoder: Learning to Write Programs. In *Proceedings International Conference on Learning Representations*. OpenReviews.net.

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

Barriga, N. A.; Stanescu, M.; and Buro, M. 2017b. Game Tree Search Based on Non-Deterministic Action Scripts in Real-Time Strategy Games. *IEEE Transactions on Computational Intelligence and AI in Games*.

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.

Benbassat, A.; and Sipper, M. 2012. Evolving both search and strategy for Reversi players using genetic programming. In *IEEE Conference on Computational Intelligence and Games*, 47–54.

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.

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

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.

Garey, M. R.; and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

Gulwani, S.; Polozov, O.; and Singh, R. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4(1-2): 1–119. ISSN 2325-1107. doi:10.1561/2500000010.

Jha, S.; Gulwani, S.; Seshia, S. A.; and Tiwari, A. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, 215–224.

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

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

Liang, P.; Jordan, M. I.; and Klein, D. 2010. Learning Programs: A Hierarchical Bayesian Approach. In *Proceedings of the International Conference on Machine Learning*, 639–646. Omnipress.

Menon, A.; Tamuz, O.; Gulwani, S.; Lampson, B.; and Kalai, A. 2013. A Machine Learning Framework for Programming by Example. In *Proceedings of the International Conference on Machine Learning*, 187–195. PMLR.

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.

Murali, V.; Chaudhuri, S.; and Jermaine, C. 2017. Bayesian Sketch Learning for Program Synthesis. In *Proceedings International Conference on Learning Representations*. OpenReviews.net.

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

Ontañón, S.; Barriga, N. A.; Silva, C. R.; Moraes, R. O.; and Lelis, L. H. 2018. The First MicroRTS Artificial Intelligence Competition. *AI Magazine* 39(1).

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362(6419): 1140–1144. ISSN 0036-8075. doi:10.1126/science.aar6404.

Slavík, P. 1996. A tight analysis of the greedy algorithm for set cover. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing*, 435?441.

Spronck, P.; Sprinkhuizen-Kuyper, I.; and Postma, E. 2004. Online adaptation of game opponent AI with dynamic scripting. *International Journal of Intelligent Games and Simulation* 3(1): 45–53.

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

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. M.; Yue, Y.; and Chaudhuri, S. 2019. Imitation-Projected Programmatic Reinforcement Learning. *arXiv preprint arXiv:1907.05431* .

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.

Vinyals, O.; Babuschkin, I.; Czarnecki, W. M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D. H.; Powell, R.; Ewalds, T.; Georgiev, P.; Oh, J.; Horgan, D.; Kroiss, M.; Danihelka, I.; Huang, A.; Sifre, L.; Cai, T.; Agapiou, J. P.; Jaderberg, M.; Vezhnevets, A. S.; Leblond, R.; Pohlen, T.; Dalibard, V.; Budden, D.; Sulsky, Y.; Molloy, J.; Paine, T. L.; Gulcehre, C.; Wang, Z.; Pfaff, T.; Wu, Y.; Ring, R.; Yogatama, D.; Wünsch, D.; McKinney, K.; Smith, O.; Schaul, T.; Lillicrap, T.; Kavukcuoglu, K.; Hassabis, D.; Apps, C.; and Silver, D. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 575(7782): 350–354.

Yang, Z.; and Ontanón, S. 2019. Guiding Monte Carlo Tree Search by Scripts in Real-Time Strategy Games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, 100–106.

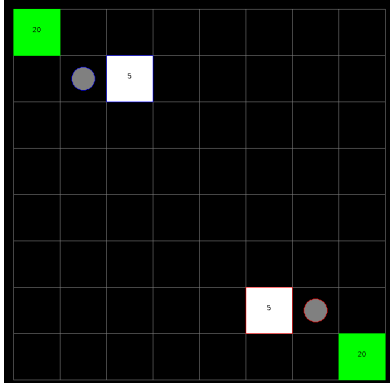# APPENDIX B – Interpretability Evaluation - Questionnaire One

Projeto Interface MicroRTS
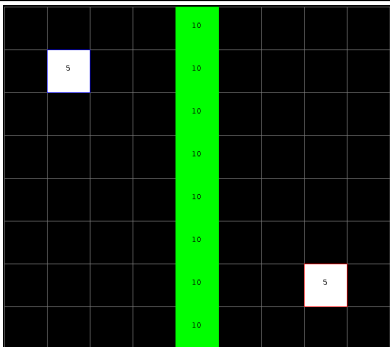
Análise dos Scripts - Questionário 1

Identificação do Participante

UUID	Idade	Identificação
984dde900033e5a0ac6423610dc850e7	26	Female
Formação superior	**Anos experiência em programação**	**Cursos na área de Inteligência Artificial**
No	2	1
Realiza(ou) pesquisa na área de IA?	**Joga jogos de computador?**	**Joga jogos de RTS?**
Yes	Never	Once or twice

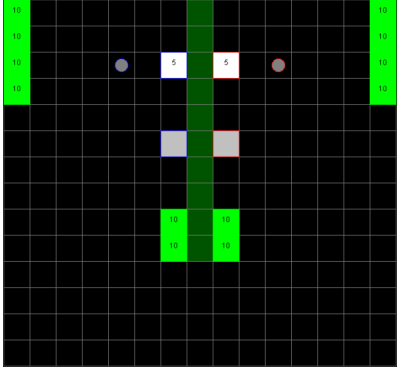## Análise Mapa 8x8 - Bases Workers A

Mapa	Script Submetido pelo Participante (Sc_8x8_1.txt)
	harvest(1) train(Worker,50,EnemyDir) attack(Worker,closest)

Solução que ganha do Script Submetido (Sc_8x8_2.txt)	Solução que ganha dos scripts anteriores (Sc_8x8_3.txt)

if not HaveQtdEnemiesbyType(Worker,3):
    harvest(2)
else:
    train(Worker,9,Up)
harvest(1)
train(Worker,9,Right)
attack(Worker,closest)
if HaveQtdUnitsAttacking(Ranged,6) :
    moveOnceToCoord(Worker,5,2,8)
harvest(1)
	if not HaveUnitsToDistantToEnemy(Worker,4):
	    harvest(3)
	    moveaway(Worker)
	    moveToUnit(Worker,Enemy,strongest)
	harvest(1)
	train(Worker,9,Right)
	attack(Worker,closest)
	for(u):
	    if not HaveQtdEnemiesbyType(Light,5):
	        moveToUnit(Worker,Enemy,mostHealthy,u)
	if not HaveQtdUnitsHarversting(6):
	    moveOnceToCoord(Heavy,9,11,11)
	for(u):
	    if not HaveUnitsToDistantToEnemy(Heavy,7,u):
	        moveToUnit(Worker,Ally,weakest,u)
	        moveOnceToCoord(Light,1,2,8,u)
	if IsPlayerInPosition(Down) :
	    moveOnceToCoord(Heavy,4,11,4)
	else:
	    moveOnceToCoord(Heavy,1,4,3)
	for(u):
	    if HaveEnemiesStrongest(Light,u) :
	        moveOnceToCoord(Worker,9,5,6,u)
	    else:
	        moveOnceToCoord(Light,3,10,12,u)
	    if HaveQtdUnitsbyType(Worker,7) :
	        moveOnceToCoord(Heavy,3,6,9,u)
	        moveOnceToCoord(Heavy,1,10,1,u)
	    else:
	        harvest(2,u)

Análise Mapa 9x8 - No Where to Run

Mapa	Script Submetido pelo Participante (Sc_9x8_1.txt)

	if IsPlayerInPosition(Left) :
	    build(Barrack,1,Down)
	if IsPlayerInPosition(Right) :
	    build(Barrack,1,Up)
	harvest(1)
	train(Ranged,50,EnemyDir)
	attack(Ranged,closest)

Solução que ganha do Script Submetido (Sc_9x8_2.txt)	Solução que ganha dos scripts anteriores (Sc_9x8_3.txt)

for(u):
    build(Base,1,Right,u)
build(Barrack,1,Up)
moveToCoord(Light,0,0)
for(u):
    attack(Light,strongest,u)
for(u):
    if not HaveUnitsinEnemyRange(Light,u):
        train(Worker,1,Up)
        harvest(2,u)
    if not HaveUnitsToDistantToEnemy(Heavy,1,u):
        train(Light,1,Down)
    else:
        moveOnceToCoord(Heavy,2,1,5,u)
	for(u):
	    if not HaveEnemiesStrongest(Ranged,u):
	        build(Barrack,1,Down,u)
	        moveToUnit(Worker,Enemy,lessHealthy,u)
	        harvest(2,u)
	    if not HaveUnitsStrongest(Light,u):
	        train(Worker,3,EnemyDir)
	    build(Barrack,1,Up,u)
	    if not HaveQtdUnitsHarversting(7):
	        train(Ranged,3,Down)
	    moveOnceToCoord(Worker,7,1,5,u)
	    if not HaveUnitsToDistantToEnemy(Ranged,6,u):
	        moveOnceToCoord(Worker,2,4,14,u)
	        moveaway(Worker,u)
	        moveToCoord(Worker,4,7,u)
	        train(Ranged,4,EnemyDir)
	for(u):
	    if not HaveQtdEnemiesbyType(Light,6):
	        attack(Ranged,closest,u)
	    moveOnceToCoord(Light,6,14,7,u)
	    harvest(1,u)
	    harvest(3,u)
	    if not HaveQtdEnemiesbyType(Light,8):
	        moveOnceToCoord(Light,2,10,4,u)
	if not HaveEnemiesStrongest(Heavy):
	    attack(Ranged,strongest)

Análise Mapa 15x14 - It's not Safe

Mapa	Script Submetido pelo Participante (Sc_15x14_1.txt)

	if IsPlayerInPosition(Left) :
	    build(Barrack,1,Down)
	if IsPlayerInPosition(Right) :
	    build(Barrack,1,Up)
	harvest(1)
	train(Ranged,50,EnemyDir)
	attack(Ranged,closest)

Solução que ganha do Script Submetido (Sc_15x14_2.txt)	Solução que ganha dos scripts anteriores (Sc_15x14_3.txt)

if not HaveQtdUnitsbyType(Light,4):
    train(Ranged,1,Down)
    attack(Ranged,closest)
if not IsPlayerInPosition(Down):
    moveOnceToCoord(Light,8,10,10)
	if not HaveQtdUnitsAttacking(Heavy,5):
	    train(Light,8,Left)
	attack(Light,closest)
	for(u):
	    if not HaveQtdEnemiesbyType(Worker,9):
	        harvest(1,u)
	        attack(Light,weakest,u)
	    train(Worker,5,Left)
	    moveOnceToCoord(Light,3,4,9,u)
	for(u):
	    if not HaveQtdUnitsAttacking(Heavy,5):
	        attack(Worker,closest,u)
	        moveOnceToCoord(Light,8,6,2,u)
	        harvest(3,u)
	    train(Worker,7,Up)

APPENDIX C – Invitation Letter for Experiments with Humans

Olá a todos!

Hoje viemos apresentar um desafio para aqueles que gostam de jogos de estratégia em tempo real. Você já se imaginou construindo estratégias inteligentes para jogar aquele mapa de Age of Empires ou Starcraft? Ou durante uma de suas partidas no seu RTS preferido você achou aquele "bot" fácil demais e pensou: "Eu construiria um bem melhor"! Hoje é o seu dia de tornar isso realidade, enquanto ajuda nossa pesquisa :) !

Nós desenvolvemos um sistema que auxilia a criação de estratégias, as quais vamos chamar de scripts, para um jogo de RTS simplificado, o MicroRTS. Utilizando esse sistema você poderá criar scripts para diferentes mapas e enfrentar diferentes oponentes.

Neste estudo, você será desafiado a criar scripts para jogar o MicroRTS em 3 mapas diferentes. Quando terminar de escrevê-los, pedimos que envie-nos os arquivos e que respondam um pequeno questionário Claro, todas as informações serão mantidas em sigilo e anônimas. O processo de envio e questionário será todo feito utilizando o nosso programa.

O que vamos fazer com seus scripts? Quando recebermos os scripts enviados por você, utilizaremos um algoritmo que tenta identificar pontos de melhoria nas estratégias criadas por você, caso elas existam. Esse algoritmo, um programa inteligente de computador, tentará derivar duas novas estratégias:

1. Uma estratégia capaz de superar a estratégia inicialmente criada por você;

2. Uma segunda estratégia capaz de, simultaneamente, superar a sua estratégia e a que será retornada no passo 1, citada acima!

Após a execução deste algoritmo, nós enviaremos um email para você, apresentando as novas estratégias descobertas. De posse deste email, você poderá analisar e estudar as estratégias enviadas e melhorar/incrementar sua estratégia inicial através da compreensão dos pontos fracos de seu script inicial. Quando você terminar seu estudo, pediremos a você que preencha um segundo questionário com algumas informações sobre o material enviado a você.

Em resumo, o desafio é:

1. Fazer o download da nossa ferramenta e verificar algumas instruções no link:

a) Pacote Interface MicroRTS

b) Se você está utilizando Windows, siga este tutorial de instalação: Tutorial Windows

c) Se você está utilizando Linux, siga este tutorial de instalação: Tutorial Linux

2. Assistir os vídeos de treinamento: Tutorial 1 e Tutorial 2

3. Criar seus scripts e salvá-los.

4. Responder o questionário 1 e anexar seus scripts.

5. Esperar o email de resposta da nossa pesquisa.

6. Dar uma conferida no email enviado por nós e responder o questionário 2.

Agora é só seguir o passo-a-passo e criar estratégias no sistema. Qualquer dúvida, sinta-se à vontade para nos chamar no chat da ferramenta ou nos envie um email.

Esse estudo é aprovado pela University of Alberta Research Ethics Board ("Assistive Tool for Writing Scripts for RTS Games" - Pro00103548).

Desde já agradecemos o interesse e participação neste estudo!

APPENDIX D – Email with the Second Part of the Experiment

Olá! Agradecemos a sua participação na primeira fase do nosso experimento.

Segue em anexo o material para a segunda fase. Nesta fase pedimos a você que leia o documento em anexo. Esse documento em anexo contém os scripts que apresentam melhorias ao seu script inicial, para cada um dos mapas.

Os arquivos estão no seguinte formato:

- Sc_NxN_1.txt = Script Submetido por você para o mapa NxN;

- Sc_NxN_2.txt = Script que ganha do script Sc_NxN_1.txt;

- Sc_NxN_3.txt = Script que ganha dos scripts Sc_NxN_1.txt e Sc_NxN_2.txt.

Estes arquivos txt, já no formato requerido para ser utilizado no nosso programa, deverão ser utilizados por você para visualizar os comportamentos das soluções que lhe sugerimos.

Estes arquivos podem ser importados em nossa ferramenta. Para isso, veja como carregá-los aqui.

Após seu estudo dos scripts, pedimos que você responda o questionário 2. Para isso, siga os seguintes passos:

1 - Abra o nosso programa. 2 - Click no botão 'Send Q2'. Caso o botão não esteja habilitado, entre em contato conosco respondendo este e-mail(rubensolv@gmail.com) ou pelo chat do programa. 3 - Preencha o questionário e clique em enviar.

Agradecemos sua colaboração e aguardamos as suas respostas desta segunda e última fase.

APPENDIX E – Questions on Questionnaire One and Two

The questionnaire one contains the following questions:

1. What is your email address? Your email will be used to send you the results of our computational experiments and will not be distributed, not even the researchers involved in this research will be able to directly access your email.

2. How old are you?

3. How do you identify yourself?

4. Do you have a degree in computer science or related field (e.g., Information Systems and Electrical Engineering)?

5. How many years of programming experience do you have?

6. How many courses in Artificial Intelligence have you taken?

7. Have you done any form of game AI research?

8. How often do you play computer games?

9. Have you ever played RTS games?

10. How much time (in minutes) have you spent working on your program?

11. The graphical user interface has helped you develop the program for playing the game.

The questionnaire two contains the following questions:

1. Please describe and comment on the reasons why you believe your program is being defeated by the program written by our system.

2. Had you anticipated that your program would have the weakness exploited by our program?

3. Please describe in English (or in Portuguese) the strategy encoded in script_2.

4. Please describe how the strategy encoded in script_2 is better than the strategy encoded in script_1 and script_3.

5. How well do you think you understand the strategies synthesized by our system?

6. **(General Question:)** Would you prefer to be assisted by our system next time you have to write a program for playing MicroRTS?