UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Synthesizing interpretable strategies for real-time planning in zero-sum games

Julian Ricardo Hernandez Mariño

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Julian Ricardo Hernandez Mariño

Synthesizing interpretable strategies for real-time planning in zero-sum games

Doctoral dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Claudio Fabiano Motta Toledo

USP – São Carlos
December 2021

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

H557s Hernandez Mariño, Julian Ricardo
 Synthesizing interpretable strategies for real-
 time planning in zero-sum games / Julian Ricardo
 Hernandez Mariño; orientador Cláudio Fabiano Motta
 Toledo. -- São Carlos, 2021.
 98 p.

 Tese (Doutorado - Programa de Pós-Graduação em
 Ciências de Computação e Matemática Computacional) --
 Instituto de Ciências Matemáticas e de Computação,
 Universidade de São Paulo, 2021.

 1. Program Synthesis. 2. Search Algorithm. 3.
 Artificial Intelligence. 4. Evolutionary Algorithm.
 5. RTS Games. I. Motta Toledo, Cláudio Fabiano,
 orient. II. Título.

Julian Ricardo Hernandez Mariño

**Sintetizando estratégias interpretáveis para o planejamento
em tempo real em jogos de soma zero**

Tese apresentada ao Instituto de Ciências
Matemáticas e de Computação – ICMC-USP,
como parte dos requisitos para obtenção do título
de Doutor em Ciências – Ciências de Computação e
Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e
Matemática Computacional

Orientador: Prof. Dr. Claudio Fabiano Motta Toledo

USP – São Carlos
Dezembro de 2021

To my mother Maria and my brother Oscar.

ACKNOWLEDGEMENTS

I would like to thank God for everything. I want to thank my family for their love and support, especially to my mother Maria, brother Oscar, sister-in-law Magda, and nephew Joaquin.

I would like to thank my advisors Claudio and Levi for their guidance, support, and invaluable shared knowledge. It was an honor to have been their student.

My thanks also go to Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - grants 10121925/D, for having provided me with a scholarship, as well to Levi and the University of Alberta for the temporary scholarship they gave me during the last part of my course.

I thank the Universidade de São Paulo (USP) and the Instituto de Ciências Matemáticas e de Computação (ICMC) for having allowed me to realize my PhD studies in that great institute. Also, I would like to thank to the departamento de informática at Universidad Federal de Viçosa (UFV) for allowing me to spend some months in their labs as an external researcher.

Thanks to the Center for Mathematical Sciences Applied to Industry (CeMEAI) funded by FAPESP (grant 2013/07375-0), the cluster Jupiter from UFV, and the clusters from Compute Canada for having provided me with computational resources to perform my research.

My thanks also go to Franciany for her support and love.

Thanks to my friends in Brazil for their support, especially to Valdemar, Hugo Gualdron, and Juanito.

*“Hey! if you see a chance to be kind to someone tomorrow,
take it, I think we need it.” (B.P)*

ABSTRACT

MARIÑO, J. R. H. **Synthesizing interpretable strategies for real-time planning in zero-sum games**. 2021. 98 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

Interpretable and explainable Artificial Intelligence (AI) is projected as one of the most important topics for the community in the next years. In addition to developing effective AI approaches that can help humans solving problems, it might be necessary to understand the reasons behind the decisions of such approaches to finally trust in their behavior. Search and learning-based algorithms represent the current state-of-the-art approaches for planning in zero-sum real-time games. The problem with those approaches is that usually the behavior of their resulting agents is not interpretable. On the other hand, hard-coded programs usually are not as effective as search-based methods but have an important vantage; they can be more easily interpretable. In this thesis, we present a collection of works where we approach the problem of synthesizing effective interpretable scripts for planning in zero-sum real-time domains. First, we approach the problem of generating a set of scripts that can be used as an action abstraction to reduce search action spaces in zero-sum real-time strategy games. Namely, we present an evolutionary approach that can generate action abstractions that search-based algorithms can use for planning. Search-based systems that use action abstractions generated by our system outperformed the state-of-the-art search-based methods we use for experiments and won the 2018 μ RTS competition. We also present Gesy and LS2, two systems focused on synthesizing scripts that can plan by themselves in zero-sum real-time strategy games. Gesy is a system that uses a Genetic Programming (GP) approach to synthesize interpretable scripts. LS2 is a system that combines a novel method to reduce Domain-Specific Languages (DSLs), and a local-search algorithm that uses self play to synthesize interpretable scripts. The scripts Gesy and LS2 synthesize are competitive with complex search-based methods and scripts designed by professional programmers. We also show that the scripts synthesized by both systems can be used to discover possible optimizations that programmers could include in their implementations.

Keywords: Program Synthesis, Search Algorithm, Artificial Intelligence, Evolutionary Algorithm, Self-Play, Zero-Sum, Games, RTS.

RESUMO

MARIÑO, J. R. H. **Sintetizando estratégias interpretáveis para o planejamento em tempo real em jogos de soma zero**. 2021. 98 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

A Inteligência Artificial (IA) interpretável e explicável é projetada como um dos temas mais importantes para a comunidade nos próximos anos. Além de desenvolver abordagens eficazes de IA que possam ajudar aos humanos a resolver problemas, pode ser necessário entender as razões por trás das decisões de tais abordagens para finalmente confiar em seu comportamento. Os algoritmos baseados em busca e aprendizagem representam o estado da arte para o planejamento em jogos de soma zero em tempo real. O problema com essas abordagens é que geralmente o comportamento de seus agentes resultantes não é interpretável. Por outro lado, scripts geralmente não são tão eficazes quanto os métodos de busca, mas têm uma vantagem importante; eles podem ser mais facilmente interpretáveis. Nesta tese, apresentamos uma coleção de trabalhos onde abordamos o problema de sintetizar scripts interpretáveis e eficazes para o planejamento em domínios de tempo real de soma zero. Primeiro, abordamos o problema de gerar um conjunto de scripts que pode ser usado como uma abstração de ação para reduzir os espaços de busca de ações em jogos em tempo real de soma zero. Apresentamos uma abordagem evolutiva que pode gerar abstrações de ação que algoritmos baseados em busca podem usar para o planejamento. Sistemas baseados em busca que usam abstrações de ação geradas por nosso sistema superaram os métodos de busca do estado da arte que usamos nos experimentos, e venceram a competição μ RTS do 2018. Também apresentamos o Gesy e o LS2, dois sistemas focados em sintetizar scripts que podem ser usados por si sós para planejamento em jogos em tempo real de soma zero. Gesy é um sistema que usa uma abordagem de Programação Genética (GP) para sintetizar scripts interpretáveis. LS2 é um sistema que combina um novo método para reduzir Linguagens Específicas de Domínio (DSLs) e um algoritmo de busca local que usa a self-play para sintetizar scripts interpretáveis. Os scripts que Gesy e LS2 sintetizam são competitivos com métodos complexos baseados em busca e scripts codificados por programadores profissionais. Também mostramos que os scripts sintetizados pelos dois sistemas podem ser usados para descobrir possíveis otimizações que os programadores poderiam incluir em suas implementações.

Palavras-chave: Síntese de programas, Algoritmo de Busca, Inteligência Artificial, Algoritmo Evolutivo, Self Play, Soma Zero, Jogos, RTS.

LIST OF FIGURES

Figure 1 – Fraction of the only strictly dominant strategy WR in the population P of Algorithm 1 across 20 generations.	44
Figure 2 – Figure 2a shows an script wrote for μ RTS, and 2b shows a screenshot of a μ RTS game that is being played by two players that are following the strategy encoded by the script in figure 2a.	53
Figure 3 – Derivation tree representation of a synthesized script using G_1	54
Figure 4 – Example of a script randomly created by the initialization procedure in Gesy. In figure 4a it is shown the pseudocode of the script, and in figure 4b it is shown the derivation tree that represents the script.	57
Figure 5 – Example of crossover following grammar G_2	58
Figure 6 – Example of mutation following grammar G_2	58
Figure 7 – The experiments execute round-robin tournaments from the best individuals found at each method run. The scripts play against each other to assess their performance, and the winner plays against state-of-art related methods. . . .	61
Figure 8 – Derivation tree for “c1 if (b1) then c3”.	75
Figure 9 – Extensive form representations for a two-player zero-sum game	96
Figure 10 – A screenshot of a μ RTS match.	97

LIST OF ALGORITHMS

Algorithm 1 – Evolutionary Algorithm for Solving SSG	38
Algorithm 2 – Gesy	55
Algorithm 3 – Evaluate($\mathcal{G}, \Phi, \Psi, F$)	57
Algorithm 4 – Self play with local search	78

LIST OF TABLES

Table 1	– Average percentage of matches won by the column player against the row player for maps of different sizes.	46
Table 2	– Winning rate of the row player against the column player. The last column (‘Avg.’) shows the average winning rate of each row method considering its matches against all methods we included in our experiments.	62
Table 3	– Response time of different approaches.	64
Table 4	– 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. . .	82
Table 5	– Average winning rate across all four maps tested.	83
Table 6	– Scripts synthesized by LS2 and the best two scripts written by the programmers for maps 18×8 and 24×24	84

LIST OF ABBREVIATIONS AND ACRONYMS

Lasi	Domain-Specific Language Simplifier
LS2	Local Search and Language Simplifier
AHT	Adversarial Hierarchical Task Network
CFG	context-free grammar
DSBs	domain-specific Boolean functions
DSFs	domain-specific functions
DSL	Domain Specific Language
ESS	Evolutionary Stable Strategies
GP	Genetic Programming
HR	Heavy Rush
LR	Light Rush
PGS	Portfolio Greedy Search
PS	Puppet Search
RR	Ranged Rush
RTS	real-time strategy
SCV	Strategy Creation Through Voting
SSG	subset selection game
SSS	Stratified Strategy Selection
STT	Strategy Tactics
WR	Worker Rush

CONTENTS

1	INTRODUCTION	27
1.1	Context and Motivation	27
1.2	Problem and Research Questions	28
1.3	Objectives	29
1.4	Structure of the Thesis	29
1.4.1	<i>Chapter 2: Evolving Action Abstractions for Real-Time Planning in Extensive-Form Games</i>	30
1.4.2	<i>Chapter 3: Evolving Interpretable Strategies for Zero-Sum games</i>	30
1.4.3	<i>Chapter 4: Programmatic Strategies for Real-Time Strategy Games</i>	31
2	EVOLVING ACTION ABSTRACTIONS FOR REAL-TIME STRATEGY GAMES	33
2.1	Introduction	34
2.2	Related Works	35
2.3	Background	35
2.4	Generation of Action Abstractions as a Two-Player Zero-Sum Game	37
2.5	An Evolutionary Algorithm for SSG	38
2.6	Evolutionary Stable Strategies	39
2.7	Generation of Novel Strategies from Expert-Designed Rule-Based Strategies	40
2.8	Empirical Methodology	41
2.8.1	<i>Problem Domain</i>	41
2.8.2	<i>Set $\mathcal{L}$ of Strategies</i>	42
2.8.3	<i>Search Algorithms</i>	42
2.8.4	<i>Maps</i>	42
2.8.5	<i>Experiment Sets</i>	43
2.8.6	<i>Evolutionary Parameters</i>	44
2.9	Empirical Results	44
2.9.1	<i>First Part: Convergence to an ESS</i>	44
2.9.2	<i>Second Part: Comparison with the State of the Art</i>	45
2.10	Limitations	47
2.11	Conclusions	48

2.11.0.0.1	Acknowledgements	48
3	EVOLVING INTERPRETABLE STRATEGIES FOR ZERO-SUM GAMES	
		49
3.1	Introduction	49
3.2	Related Work	51
3.3	RTS games and μ RTS	52
3.4	Script Synthesis in Zero-sum Games	54
3.5	Genetic Programming to Synthesize Scripts	55
3.6	Empirical Methodology	59
3.6.1	<i>A DSL for μRTS</i>	59
3.6.2	<i>Experiment Setup</i>	60
3.6.3	<i>Results: Strength of Play</i>	63
3.6.4	<i>Results: Computational Time</i>	64
3.6.5	<i>Interpretability Analysis</i>	64
3.7	Conclusions and Future Works	65
3.8	Appendix	66
3.8.1	<i>DSL used in the experiments</i>	66
4	PROGRAMMATIC STRATEGIES FOR REAL-TIME STRATEGY-GAMES	
		71
4.1	Introduction	72
4.2	Related Work	73
4.3	The Script Synthesis Problem	74
4.4	Domain-Specific Languages	75
4.5	Domain-Specific Language Simplifier	75
4.5.1	<i>The DSL Simplification Problem</i>	76
4.5.2	<i>Simplification Problem as Set Cover</i>	76
4.5.2.1	<i>Removing DSFs</i>	77
4.5.2.2	<i>Removing DSBs</i>	77
4.6	Synthesizing Scripts with Self Play	77
4.7	Empirical Methodology	78
4.7.1	<i>Problem Domain: μRTS</i>	79
4.7.2	<i>Generating Trace for Simplifying the DSL</i>	79
4.7.3	<i>Competing Agents</i>	80
4.7.4	<i>μLanguage: a Domain-Specific Language for μRTS</i>	80
4.7.5	<i>Scripts Written by Programmers</i>	81
4.8	Empirical Results	81
4.8.1	<i>Evaluation of Strength of Play</i>	81
4.8.2	<i>Interpretability of Scripts</i>	84

4.9	Conclusions	85
4.10	Acknowledgements	86
5	CONCLUSION	87
	BIBLIOGRAPHY	89
	APPENDIX A BASIC DEFINITIONS	95
A.1	Zero-Sum Games	95
A.2	μ RTS	96
A.3	Domain-Specific Languages	97

INTRODUCTION

This thesis focuses on the synthesis of interpretable strategies for planning in zero-sum real-time games. In section 1.1 we describe the context and the reasons that led us to carry out this work, in section 1.2 we describe the research question is approached in this thesis, in section 1.3 are presented the main goals pursued by this research, and in section 1.4 we present the description of the structure of this document.

1.1 Context and Motivation

Search-based and machine-learning systems have shown impressive advancements in complex-planning domains such as robotics and computer games. AlphaStar (VINYALS *et al.*, 2019a), for example, is a system that combines search-based techniques with reinforcement learning to defeat human-professional players in starcraft II. This popular real-time strategy game has complex rules, partial observability, non-dominant strategies, and vast action space. AlphaStar reached the top league of players and the category of grand-master for the game. Such achievement meant a significant milestone for the progress of Artificial Intelligence (AI) in general (ARULKUMARAN; CULLY; TOGELIUS, 2019) since insights from AlphaStar might impact areas such as self-driving cars and robotics.

One disadvantage of search and learning-based methods is to approach them as 'black-boxes, i.e., their decisions are often non-interpretable (DOŠILOVIĆ; BRČIĆ; HLUPIĆ, 2018). The Artificial Intelligence community has grown interested in developing methods that can generate interpretable and transparent agent behaviors. Interpretability and transparency are key features for the acceptance of AI methods (SIAU; WANG, 2018). For example, in health-care, where human lives are involved, the reasons behind the diagnostic decisions given by an AI system are required to better trust in the clinical pathway that a patient has to follow (AHMAD; ECKERT; TEREDESAL, 2018).

Hard-coded programs, which we refer to as scripts, may not have shown to be as powerful as search-based and machine-learning methods for planning in real-time planning domains. However, scripts have a significant advantage: they are easier to interpret than search-based and machine-learning models. It's possible to build scripts using one or more interpretable code instructions. Therefore, scripts can be used in domains where trustworthy, explainable, and transparent agent decisions are essential.

Scripts can also increase the effectiveness of search-based methods. Complex planning domains can state large action spaces where search-based algorithms have to explore. The exploration over wide spaces can be problematic when the available time to make decisions is limited. A promising approach that has shown to be effective in the context of searching in large actions spaces is the use of action abstractions ([CHURCHILL; BURO, 2013](#)). An action abstraction is a set of scripts that redefine an action space. Namely, instead of searching for promising actions in the original large action space, a search algorithm uses a few scripts to search only in the space given by the actions that the scripts return. Action abstractions are advantageous in domains where search algorithms have to produce a response in milliseconds, such as real-time strategy games or robot rescue tasks. As an example of the effectiveness of action abstractions, consider the results of 2018 μ RTS competition ([ONTAÑÓN *et al.*, 2018](#)), a competition where different planning approaches compete playing μ RTS, a zero-sum real-time strategy game developed for research purposes. In that competition, the winner was a system that combines an action abstraction induced by a set of few scripts with a search algorithm that "pays attention" to different aspects of the game. The system used to generate those action abstractions is one of the resulting systems of this thesis.

1.2 Problem and Research Questions

The problem of synthesizing effective scripts can be challenging. For example, consider the game industry where scripts control AI agents that can play the game. Programmers often design the scripts manually in trial-and-error loops to find trustworthy strategies to be released safely in the game's final version. Such a need makes the script developing process time-consuming.

It's possible to use program synthesis approaches ([GULWANI; POLOZOV; SINGH, 2017](#)) to synthesize scripts for zero-sum real-time games automatically. However, in contrast with traditional program synthesis problems, it is necessary to perform complete game matches in zero-sum games to evaluate a single script's performance accurately. There are no explicit specifications previously defined, such as input-output examples to be used as guides during the search of scripts. Therefore, synthesizing effective scripts can be an intractable problem in domains where the search space of scripts is large, or game matches require long run times.

In program synthesis, a Domain Specific Language (DSL) defines the space of possible

scripts that can be synthesized for a specific domain. If the DSL is too constrained, the process of coding scripts could be accelerated. However, very constrained DSLs might not be expressive enough to allow the synthesis of effective programs. On the other hand, if the DSL defines a large space of programs, the search for effective scripts can become infeasible.

In this thesis, we raise the questions: *"Is it possible to synthesize effective scripts for planning in zero-sum real-time domains?"*, if so, *"The synthesized scripts are competitive with search-based approaches as well as with scripts developed by professional programmers?"*, *"The synthesized scripts can be easily interpreted?"*, and, *"Is it possible to synthesize effective action abstractions induced by a set of scripts?"* To have these questions answered, in section 1.3 we define the main goals of this thesis.

1.3 Objectives

The purpose of this thesis is twofold: to explore ways to synthesize interpretable scripts encoding effective strategies for real-time zero-sum games and to develop a system able to generate effective action abstractions induced by a set of scripts for real-time zero-sum games. Namely, the specific goals pursued in this thesis are:

- Build a system able to automatically synthesize scripts that encode effective planning strategies in zero-sum real-time domains. The strategies encoded in the scripts must be interpretable.
- Build a system able to synthesize action abstractions induced by a set of scripts automatically. Search-based algorithms must use the generated action abstractions to focus their search on more promising action spaces during planning in real-time domains.
- Investigate automatic ways for simplifying a DSL that defines the space of programs that a script synthesizer can synthesize. The simplified DSL must allow scripts synthesizers to focus on the search of strategies within a reduced but promising scripts space.
- Evaluate the proposed systems in a zero-sum real-time game. Compare the resulting behavior of the agents that follows the strategies synthesized by our systems with the state-of-the-art search-based systems and with scripts coded by human professional programmers.

1.4 Structure of the Thesis

This thesis is presented as follows: Chapters 2, 3 and 4 each contain a submitted or published paper that reports how one or more of the research goals proposed in this thesis was

approached¹. Chapter 5 presents the main conclusions from the research carried out in this thesis. Finally, we present some definitions related to this work in appendix A. For ease of reading, we present a summary of the papers presented in chapters 2, 3 and 4, as follows:

1.4.1 Chapter 2: Evolving Action Abstractions for Real-Time Planning in Extensive-Form Games

The chapter presents the paper "*Evolving Action Abstractions for Real-Time Planning in Extensive-Form Games*" (MARIÑO *et al.*, 2019), published in the *AAAI Conference on Artificial Intelligence (AAAI-19)*.

Previous works showed that using a set of scripts to induce action abstractions helps search-based methods to search on more promising actions spaces during planning (MORAES *et al.*, 2018; CHURCHILL; BURO, 2013). The main drawback of those approaches is that the action-abstracted space is limited to a small number of hand-crafted strategies developed by experts. No deep investigation was performed about how action abstractions induced by different scripts could impact the behavior of the search-based methods. Also, using only a limited pool of scripts to induce an action abstraction can constraint the behavior of the agents to the human knowledge encoded in that small set of scripts.

The paper presented in this chapter introduces a system to generate action abstractions for two-player extensive-form games from two steps. First, the system automatically synthesizes a large set of scripts, variations of a small hand-crafted set of scripts by changing their original parameters. Thus, an evolutionary algorithm selects a subset of that large set of scripts that induces an effective action abstraction. We model selecting the most effective subset of scripts as a two-player simultaneous-move game by calling the subset selection game (SSG). Each possible subset of scripts from the original set is a possible strategy that the players of the SSG can follow. As a result, the combination of the action abstractions induced by our system with the search algorithm A3N (MORAES *et al.*, 2018), won the 2018 IEEE μ RTS competition.

1.4.2 Chapter 3: Evolving Interpretable Strategies for Zero-Sum games

The chapter presents the paper "*Evolving Interpretable Strategies for Zero-Sum Games*", submitted in the journal *Applied Soft Computing*.

The main disadvantage of search-based algorithms, such as the A3N used in (MARIÑO *et al.*, 2019), is that the resulting behavior of the agents can be hard to understand. There are no explicit descriptions of the decision taken by search-based methods during the search. Scripts, on the other hand, can be more easily interpretable. Another drawback of search-based approaches is the computational time they need to search for effective actions. Search-based approaches

¹ Papers presented can be reproduced in this thesis according to the copyright specifications of the conference/journal where the papers were submitted or published.

often require more computational resources to decide the agent's next action than the resources required by scripts. However, scripts present an important disadvantage: they usually are not as effective as search-based systems for planning on zero-sum real-time domains.

The paper presented in this chapter explores the automatic synthesis of scripts to plan in zero-sum real-time games. We present Gesy, a Genetic Programming (GP) approach that searches in the space of scripts defined by a Domain Specific Language (DSL). The DSL used in the experiments allows for the synthesis of scripts with various code structures. The output of Gesy is a script that can play the game.

1.4.3 Chapter 4: Programmatic Strategies for Real-Time Strategy Games

This chapter presents the paper "Programmatic Strategies for Real-Time Strategy-Games" (MARIÑO *et al.*, 2021), published in the *AAAI Conference on Artificial Intelligence (AAAI-21)*.

The main drawback of the GP approach presented in chapter 3 is that in some scenarios, the synthesizer needs the inclusion of a set of handcrafted scripts in the initial population of the evolutionary procedure to help Gesy to find robust strategies. When it is not always possible to have a pre-defined set of scripts, the approach could demand a very long run time to find strong strategies. The explanation for such an issue is that the DSL used in the experiments defines a large space of scripts, including different scripts employed in different scenarios. One solution to this problem is introducing a DSL that describes a reduced programs space to accelerate the search for scripts. The problem with this solution is that synthesized programs might not be expressive enough to encode robust game strategies.

The paper in this chapter presents a system that combines a novel algorithm for simplifying DSLs with a local search algorithm that synthesizes interpretable scripts via self-play. The DSL simplifier receives as input a trace of the game played by a demonstrator, which can be a search-based system. Next, the system greedily chooses a subset of features from the DSL given as input to produce a simplified DSL. Such DSL allows the local search algorithm to synthesize a program that can reproduce the game played by the demonstrator.

EVOLVING ACTION ABSTRACTIONS FOR REAL-TIME STRATEGY GAMES

Authors:

Julian R. H. Mariño (julianmarino@usp.br)

Rubens O. Moraes (rubens.moraes@ufv.br)

Claudio Toledo (claudio@icmc.usp.br)

Levi H. S. Lelis (levi.lelis@ufv.br)

Abstract

A key challenge for planning systems in real-time multi-agent domains is to search in large action spaces to decide an agent's next action. Previous works showed that hand-crafted action abstractions allow planning systems to focus their search on a subset of promising actions. In this paper¹ we show that the problem of generating action abstractions can be cast as a problem of selecting a subset of pure strategies from a pool of options. We model the selection of a subset of pure strategies as a two-player game in which the strategy set of the players is the powerset of the pool of options—we call this game the subset selection game. We then present an evolutionary algorithm for solving such a game. Empirical results on small matches of μ RTS show that our evolutionary approach is able to converge to a Nash equilibrium for the subset selection game. Also, results on larger matches show that search algorithms using action abstractions derived by our evolutionary approach are able to substantially outperform all state-of-the-art planning systems tested.

¹ This work was published in the *AAAI Conference on Artificial Intelligence (AAAI-19)*

2.1 Introduction

Real-time planning in multi-agent scenarios such as real-time strategy (RTS) games are challenging for current search-based systems. In such domains search algorithms have to choose an action from an often large number of options, and the time allowed for planning is in the order of milliseconds. A promising approach for dealing with large action spaces in the context of real-time planning is the use of action abstractions to reduce the action space (CHURCHILL; BURO, 2013). Instead of accounting for all actions available to an agent during search, algorithms searching with action abstractions consider only a subset of the legal actions. Churchill and Buro (CHURCHILL; BURO, 2013) used domain knowledge through a set of strategies to induce action abstractions, where a strategy is a function mapping a state to an action. Churchill and Buro’s idea was to only consider during search the actions returned by a strategy from a set of expert-designed options, with all the other actions being ignored. Several algorithms were introduced for searching in action-abstracted state spaces (JUSTESEN *et al.*, 2014; WANG *et al.*, 2016; LELIS, 2017; MORAES; LELIS, 2018).

The main drawback of these approaches is that they are limited to a small number of hard-coded expert-designed strategies. That is, the action-abstracted space from which a search algorithm derives a strategy in real time is limited to a small number of handcrafted strategies, which often limit the behavior of the agent controlled by the system. In this paper we introduce an approach to automate the generation of action abstractions for two-player extensive form games while still using the knowledge encoded in expert-designed strategies. Our approach generates a large pool of strategies $\mathcal{Z}$ from a small set of options $\mathcal{I}$ designed by domain experts. This is achieved by defining the strategies in $\mathcal{I}$ as rule-based strategies, which are strategies composed by a set of rules, that are themselves defined by a Boolean expression and an action to be performed by the agent. For example, in RTS games, a Boolean expression of a rule could be “controls 3 worker units”, where the number 3 is a parameter of the expression, and the action the agent performs if the expression returns true could be “construct a base”. By changing the parameter values of rule-based strategies we are able to generate a large pool of strategies $\mathcal{Z}$, and any non-empty subset of $\mathcal{Z}$ can be used to induce an action abstraction.

We cast the problem of generating an action abstraction from $\mathcal{Z}$ as a two-player simultaneous-move game which we name the subset selection game (SSG). In the SSG each player selects a subset of $\mathcal{Z}$, A and B , which are then used to induce an action abstraction for each player. The payoff of the SSG is defined by the result of the original two-player extensive-form game played by two versions of a planning system: one using the action abstraction induced by A and another using the action abstraction induced by B .

We introduce an evolutionary algorithm for solving the SSG and evaluate it in the domain of μ RTS, an RTS game developed for research purposes. Empirical results on small μ RTS matches show that our evolutionary approach is able to converge to a Nash equilibrium profile for the SSG. Results in medium-sized and large μ RTS matches show that two search-based

planning systems using action abstractions derived by our method are able to outperform all state-of-the-art planning systems tested in our experiments.

2.2 Related Works

All previous works that used strategies to induce action abstractions (CHURCHILL; BURO, 2013; WANG *et al.*, 2016; LELIS, 2017; MORAES; LELIS, 2018) relied on a limited number of strategies, with six being the largest number of strategies considered. Instead of using only strategies designed by experts, we use a large set of strategies generated from an initial set of expert-designed strategies. Puppet Search (PS) (BARRIGA; STANESCU; BURO, 2017b) uses an approach similar to ours for generating strategies from a small set of rule-based strategies. Instead of searching in the actual space of the problem, PS searches in the parameter space of a set of strategies. Our approach differs from PS in that we generate novel strategies as a preprocessing step, while PS searches for novel strategies as it plays the game. Strategy Tactics (STT) (BARRIGA; STANESCU; BURO, 2017a) is an enhanced version of PS which combines PS’s search in the parameter space with a search in the game’s actual space.

Strategy Creation through Voting (SCV) also generates a large set of strategies from a small number of expert-designed ones (SILVA *et al.*, 2018). In contrast with our approach, SCV’s strategies are not used to induce action abstractions, they are used to play the game directly.

SPRONCK *et al.* (SPRONCK *et al.*, 2006) introduced a reinforcement learning method for exploring sequences of rules to define strategies. Others have evolved strategies (CANAAN *et al.*, 2018; PONSEN *et al.*, 2006). In contrast with these works that derive strategies to play a game directly, we derive a set of strategies to induce action abstractions.

Action abstractions have also been applied to the domain of poker (SANDHOLM, 2015). Action abstractions for poker are fundamentally different from our action abstractions. While most of the action abstractions for poker are manually crafted by human experts, a few works describe methods to automatically find a discretization of the continuous bet sizing in no-limit poker (HAWKIN; HOLTE; SZAFRON, 2011; HAWKIN; HOLTE; SZAFRON, 2012; BROWN; SANDHOLM, 2014). By contrast, our action abstraction scheme is used to select a subset of actions from a discrete set of options. Action abstractions in poker are used to reduce the size of the game so that the abstracted game can be solved in an offline procedure, while we use action abstractions to search for a strategy in real time.

2.3 Background

Our evolutionary approach is designed to automatically craft action abstractions to real-time zero-sum simultaneous-moves extensive-form games, which can be defined by a tuple $\nabla = (\mathcal{N}, \mathcal{S}, s_{init}, \mathcal{A}, \mathcal{R}, \mathcal{T})$. Here,

- $\mathcal{N} = \{i, -i\}$ is the set of **players**.
- $\mathcal{S} = \mathcal{D} \cup \mathcal{F}$ is the set of **states**, where $\mathcal{D}$ denotes the set of **non-terminal states** and $\mathcal{F}$ the set of **terminal states**.
- $s_{init} \in \mathcal{D}$ is the start state of the game.
- $\mathcal{A} = \mathcal{A}_i \times \mathcal{A}_{-i}$ is the set of **joint actions**. $\mathcal{A}_i(s)$ is the set of legal **actions** that player i can perform at state s .
- $\mathcal{R}_i : \mathcal{F} \rightarrow \mathbb{R}$ is a **utility function** with $\mathcal{R}_i(s) = -\mathcal{R}_{-i}(s)$, for any $s \in \mathcal{F}$, since the game is zero sum.
- The **transition function** $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{A}_{-i} \rightarrow \mathcal{S}$ determines the successor state for a state s and a set of join actions taken at s .

The game is represented by a **game tree** rooted at s_{init} in which each node in the tree represents a state in $\mathcal{S}$ and every edge represents a joint action in $\mathcal{A}$. For states $s_k, s_j \in \mathcal{S}$, there exists an outgoing edge from s_k to s_j if there exists $a_i \in \mathcal{A}_i$ and $a_{-i} \in \mathcal{A}_{-i}$ such that $\mathcal{T}(s_k, a_i, a_{-i}) = s_j$. Nodes representing states in $\mathcal{F}$ are leaf nodes. We call a **decision point** for player i a state s in the tree such that i is the player to act in s . Since we consider real-time settings in this paper, each system playing the game is allowed a planning time in the order of milliseconds in every decision point.

A **pure strategy** is a function $\sigma_i : \mathcal{S} \rightarrow \mathcal{A}_i$ for player i , mapping a state s to an action a . In simultaneous-moves games one might have to play a **mixed strategy** to optimize the player's payoffs (GINTIS, 2000). A mixed strategy is a function $\sigma_i : \mathcal{S} \times \mathcal{A}_i \rightarrow [0, 1]$ for player i that maps a state s and an action a to the probability of the player i taking action a at state s . We denote as Σ_i and Σ_{-i} as the set of all strategies for i and $-i$. A **player strategy profile** $\sigma = (\sigma_i, \sigma_{-i})$ defines the strategy of both players.

The optimal **value of the game** rooted at state s is denoted by $V^*(s)$. We denote an optimal profile $\sigma^* = (\sigma_i^*, \sigma_{-i}^*)$, which is known as a Nash Equilibrium profile that yields $V^*(s)$. Backward induction methods (ROSS, 1971) can be used to find optimal profiles. However, depending on the problem's size, finding optimal profiles becomes impractical due to the computational cost required to compute such a profile. An approach for handling very large extensive-form games is by employing action abstraction schemes to reduce the number of actions available at nodes in the game tree. One then derives strategies by searching in the action-abstracted game tree (CHURCHILL; BURO, 2013).

An **action abstraction** (or abstraction for short) for player i is defined as a function mapping the set of legal actions $\mathcal{A}(s)$ to a subset $\mathcal{A}'(s)$ of $\mathcal{A}(s)$. Previous works have shown that one can build abstractions through a collection $\mathcal{P}$ of pure strategies (CHURCHILL; BURO,

2013). The abstracted set of legal actions of player i at state s is defined as the set of actions returned by the strategies in $\mathcal{P}$,

$$\mathcal{A}'_{\mathcal{P}}(s) = \{\sigma_i(s) | \sigma_i \in \mathcal{P}\}.$$

Once an abstraction is defined, real-time planning algorithms derive a strategy for the player by searching in the abstracted game tree in every decision point of the game.

2.4 Generation of Action Abstractions as a Two-Player Zero-Sum Game

Previous works have employed a small pool of hand-crafted pure strategies to derive an action abstraction to reduce the size of the game tree. By contrast, in this paper we generate a large pool $\mathcal{Z}$ of pure strategies for the game by changing the parameter-values from a small set $\mathcal{S}$ of expert-designed rule-based strategies. Then, we select a subset A of $\mathcal{Z}$ that is then used to define an action abstraction for the game. The subset A is selected for player i assuming that the opponent $-i$ is also able to select a subset B of $\mathcal{Z}$ to define $-i$'s action abstraction. Let $V^*(s, A, B)$ be the optimal value of the game rooted at state s if i and $-i$ use action abstractions induced by A and B , respectively, defined by,

$$\begin{aligned} V^*(s, A, B) = \max_{\sigma_i \in \Sigma_i^A} \min_{\sigma_{-i} \in \Sigma_{-i}^B} \sum_{a_i \in \mathcal{A}'_{\sigma_i}(s)} \sum_{a_{-i} \in \mathcal{A}'_{\sigma_{-i}}(s)} \sigma_i(s, a_i) \cdot \\ \sigma_{-i}(s, a_{-i}) \cdot V(\mathcal{T}(s, a_i, a_{-i}), A, B). \end{aligned} \quad (2.1)$$

The base of the recurrence is defined by $V^*(s, A, B) = \mathcal{R}_i(s)$, if $s \in \mathcal{F}$. Here, Σ_i^A and Σ_{-i}^B are the set of all possible strategies for i and $-i$ when i is restricted by an action abstraction induced by A and $-i$ is restricted by an action abstraction induced by B . $V^*(s)$ differs from $V^*(s, A, B)$ in that the latter is the optimal value of the game while players i and $-i$ account only for actions derived from strategies in Σ_i^A and Σ_{-i}^B , respectively, for every s . The former is the optimal value computed while accounting for all legal actions $\mathcal{A}(s)$, i.e., the un-abstracted game tree.

We model the problem of selecting a subset of strategies to induce action abstractions as a two-player zero-sum simultaneous-move game in which player i chooses a subset A that maximizes the game value, while player $-i$ chooses a subset B that minimizes the game value. We call such a game the **subset selection game** (SSG). Note that if the players optimize $V^*(s_{init}, A, B)$ in the SSG, then $A = B = \mathcal{Z}$ is a trivial optimal solution, since the value of V^* can only increase for larger sets A , and only decrease for larger sets B (MORAES; LELIS, 2018). However, the solution $A = B = \mathcal{Z}$ is unlikely to be effective in practice as it might yield game trees that are too large to be practical, specially for planning in games with real-time constraints. Instead of choosing A and B that optimize $V^*(s_{init}, A, B)$, in the SSG we choose A and B that solves the following,

$$\max_{A \in 2^{\mathcal{Z}}} \min_{B \in 2^{\mathcal{Z}}} V(s_{init}, A, B). \quad (2.2)$$

Algorithm 1 – Evolutionary Algorithm for Solving SSG

Require: Set $\mathcal{Z}$ of strategies, number of generations l , fitness function Ψ , population size n , elitist parameter e , tournament parameter t , mutation parameter u .

Ensure: Individual A .

```

1:  $k \leftarrow 0$ 
2:  $P \leftarrow \text{Init}(\mathcal{Z}, n)$ 
3: while  $k < l$  do
4:   Evaluate( $P, \Psi$ )
5:    $M \leftarrow \text{Select}(P, n, e, t)$ 
6:    $M \leftarrow \text{Crossover}(M)$ 
7:    $M \leftarrow \text{Mutation}(M, \mathcal{Z}, u)$ 
8:    $P' \leftarrow \text{Elitism}(P, e)$ 
9:    $P \leftarrow P' \cup M$ 
10:   $k \leftarrow k + 1$ 
11: end while
12: Evaluate( $P, \Psi$ )
13: return  $\text{argmax}_{A \in P} \Psi(A)$ 

```

Here, $V(s_{\text{init}}, A, B)$ is an approximation of $V^*(s_{\text{init}}, A, B)$ as computed by a search algorithm using abstractions induced by A and B , for players i and $-i$, respectively. We write $V(A, B)$ instead of $V(s_{\text{init}}, A, B)$ whenever s_{init} is clear from the context. By optimizing $V(s_{\text{init}}, A, B)$ instead of $V^*(s_{\text{init}}, A, B)$, we account for real-time constraints and imperfect search schemes. For example, if $A = \mathcal{Z}$ and B is a much smaller subset, then despite player i having access to a larger set of actions in the game, $V(s_{\text{init}}, A, B)$ might still be negative. This is because B allows the search procedure deriving a strategy for $-i$ to focus on a promising set of actions, while i 's search might spend all its time available for planning evaluating unpromising actions.

Our formulation of the SSG is simplified in the sense that we assume the existence of a solution with pure strategies (i.e., subsets of $\mathcal{Z}$), although one might need to play a mixed strategy to solve simultaneous-move games. We further observe that although we use the maxmin value in Equation 2.2 as the value of the SSG, we could have used the minmax value, as the maxmin and minmax values are equal in finite, two-player, zero-sum games (NEUMANN, 1928).

2.5 An Evolutionary Algorithm for SSG

In this section we describe an evolutionary algorithm for solving the SSG. A high-level description of the approach is presented in Algorithm 1. The algorithm receives as input a set $\mathcal{Z}$ of pure strategies from which one can induce abstractions, a number of generations l , a fitness function Ψ , a population size n , an elitist parameter e , a tournament parameter t , and a mutation parameter $u \in [0, 1]$. The algorithm returns a subset of $\mathcal{Z}$. In our evolutionary approach, subsets of $\mathcal{Z}$ are called individuals and a population P is a set of individuals. In our implementation, individuals are treated as totally-ordered subsets of $\mathcal{Z}$, as the order in which the pure strategies

are placed in an individual affects the result of the genetic operators we describe below. Function $\text{Init}(\mathcal{Z}, n)$ returns n random subsets of $\mathcal{Z}$ of random size (line 2) as the initial population of the procedure.

A generation is defined by an iteration of the while loop (line 3). In every generation, a set M of individuals are selected to produce another set of individuals through crossover and mutation operators (lines 6–7). Function $\text{Select}(P, n, e, t)$ selects a set M of individuals with a tournament (TALBI, 2009). That is, t individuals from P are randomly selected, and amongst them, the one with largest Ψ -value is added to M ; this process is repeated until $|M| = n - e$, which ensures that the next population has size n , as later in the process, e individuals are added to M .

Function $\text{Crossover}(M)$ implements a recombination procedure in which two individuals p_1 and p_2 of M are randomly selected to generate novel individuals. p_1 and p_2 are randomly divided in two disjoint parts: $p_{1,1}$ and $p_{1,2}$, and $p_{2,1}$ and $p_{2,2}$. Individuals p'_1 and p'_2 are generated by appending $p_{1,1}$ to $p_{2,2}$ and $p_{2,1}$ to $p_{1,2}$, respectively. Individuals p'_1 and p'_2 are then added to an initially empty set M' . This process is repeated until $|M'| = |M|$, when M' is returned and attributed to M (line 7). Function $\text{Mutation}(M, \mathcal{Z}, u)$ iterates through each strategy σ in each individual in M and replaces σ by a random $\sigma' \in \mathcal{Z}$ with probability u .

Function $\text{Elitism}(P, e)$ selects the subset P' of P with e individuals with the largest Ψ -value (line 9). The population of the next generation is determined by the union of M and P' . Since $|M| = n - e$ and $|P'| = e$, the population is always of size n . Once the procedure has completed l generations, it returns the individual in P with the largest Ψ -value (line 16).

The Ψ -value of an individual A in P is computed as,

$$\Psi(A, P) = \sum_{B \in P \wedge B \neq A} V(s_{init}, A, B).$$

The value of $V(s_{init}, A, B)$ is computed by playing the actual game ∇ (e.g., RTS match) with a search algorithm using the abstraction induced by A against the same search algorithm using an abstraction induced by B . The V -value is the final score of the game; in our implementation we use 1, 0, and -1 if the player controlled by the search algorithm using the abstraction induced by A wins, draws, or loses the match, respectively. The Ψ -value of A is the sum of the V -values of A with each $B \neq A$ in the population.

2.6 Evolutionary Stable Strategies

In our experiments we use the concept of Evolutionary Stable Strategies (ESS) for symmetric games, introduced by Maynard Smith; PRICE (Maynard Smith; PRICE, 1973), to evaluate the solutions encountered by our approach for SSG. A two-player game is symmetric if it has the same set of strategies for both players and if the utilities of the game are symmetric, i.e., players payoffs are the same if they are “player 1” or “player 2”. Our SSGs have the same

set of strategies for both players (the powerset of $\mathcal{Z}$), but there could be asymmetries in the utility function of the SSG, i.e., the result of a match of ∇ played by the same search algorithm using different abstractions. For example, in RTS games one of the players could be in advantage depending on the location she starts the game. Instead of computing the value of V directly with a single ∇ match, we mitigate possible asymmetries by computing V as the average result of two ∇ matches in which we swap the roles of “player 1” and “player 2” in the matches. This procedure allows us to assume our SSG to be symmetric.

A pure strategy A is an ESS if a population composed of A (population in the sense of set P in Algorithm 1) cannot be “invaded” by another strategy B during an evolutionary process. That is, if B appears in the population, then it is quickly eliminated through the evolutionary process for being “less fit to survival” than A . Formally, we have the following definition adapted from Maynard Smith; PRICE (Maynard Smith; PRICE, 1973).

Definition 1 (ESS). Let ∇ be a zero-sum extensive-form game with start state s_{init} . Let A be a strategy for the SSG defined over ∇ and a set of strategies $\mathcal{Z}$ for ∇ . Strategy A is an Evolutionary Stable Strategy (ESS) if the following conditions hold, for any strategy $B \neq A$ of the SSG.

$$V(A,A) \geq V(B,A) \quad (2.3)$$

$$V(A,A) = V(B,A) \Rightarrow V(A,B) > V(B,B) \quad (2.4)$$

If Equation 2.3 is satisfied, then profile (A,A) is a Nash equilibrium profile for the SSG. Equation 2.4 is a stability condition that guarantees that newly inserted strategies (mutants) are repealed by the ESS. That is, even if a mutant B is as good as the A against A , then B is still repealed for being strictly worse than A against B . In our experiments we are particularly interested in observing if the populations in our evolutionary approach present the characteristics of an ESS, i.e., if the population is composed solely of a single strategy and if other strategies are quickly eliminated.

2.7 Generation of Novel Strategies from Expert-Designed Rule-Based Strategies

Here we define rule-based strategies and how a small number of them can be used to generate a large set of strategies.

Definition 2. A rule-based strategy σ^r for ∇ is defined by a totally-ordered set of rules $\mathcal{E} = (\mathcal{B}, \mathcal{A}^r)$, where,

- $\mathcal{B} : \mathcal{S} \rightarrow \{0, 1\}$ is a function that returns, for state s , the result of a Boolean expression that depends on a set of parameters $\{\mathcal{C}_1, \dots, \mathcal{C}_h\}$. Here, each $\mathcal{C}_j$ can assume K_j different

values $\mathcal{C}_j = \{v_j^1, \dots, v_j^{K_j}\}$, and $\mathcal{C}^T = \{(v_1, \dots, v_n) \in \mathcal{C}_1 \times \dots \times \mathcal{C}_n\}$ is the set of possible combinations of value assignments for the parameters.

- $\mathcal{A}^r : \mathcal{S} \rightarrow \mathcal{A}$ is a function that receives a state s as input and returns a legal action a to be played in s .

σ^r is used to play ∇ by iteratively invoking the $\mathcal{B}$ function of each rule $\mathcal{E}$ in σ^r , for each decision point s in the game. One then executes in s the action returned by $\mathcal{A}^r(s)$ for the first rule $\mathcal{E}$ whose $\mathcal{B}$ function returns true. The order in which the $\mathcal{B}$ functions are invoked is defined by σ^r 's total ordering.

A rule-based strategy σ^r can be used to define a possibly large set of strategies by using different parameter-values $(v_1, \dots, v_n)$ in $\mathcal{C}^T$ in each rule $\mathcal{E}$ in σ^r . For example, a rule-based strategy for an RTS game could have the following rule $\mathcal{E}$: “if controlling 2 combat units, then attack the enemy”. Here, $\mathcal{B}$ is “controlling 2 combat units”, where the number of units is a parameter $\mathcal{C}$ that can assume any value $v > 0$. The action a of $\mathcal{E}$ is “attack the enemy”. By changing the parameter value from 2 to 3, one generates another instantiation of the rule-based strategy to play the game.

In our experiments we use a small set of expert-designed rule-based strategies to automatically generate a large set of strategies by changing the parameter values of the initial strategies. The resulting strategies compose our set $\mathcal{Z}$.

2.8 Empirical Methodology

2.8.1 Problem Domain

All our experiments are run on μ RTS, an RTS game developed for research (ONTAÑÓN, 2013). μ RTS allows one to test algorithms without having to deal with engineering problems normally encountered in commercial video games. Moreover, there is an active community using μ RTS as research testbed, with competitions being organized (ONTAÑÓN *et al.*, 2018), which helps organizing all methods in a single codebase.² In μ RTS each player starts the game controlling a set of worker units on a gridded map. Workers can be used to collect resources, build structures, and battle the opponent. Players also start with a structure called base, which is used to train workers and to store resources. In addition to the base, another structure workers can build is called the barracks, which can be used to train combat units. μ RTS has the following combat units: light, ranged, and heavy. Combat units differ in how much damage they can take before being removed from the game, how much damage they can inflict to opponent's units, and how close to an opponent's unit u they must be to attack u . A player wins a μ RTS match if

² <https://github.com/santiontanon/microrts>

she is able to remove all the other player’s units from the game. Every system is allowed 100 milliseconds for planning in every decision point.

2.8.2 Set $\mathcal{Z}$ of Strategies

We use an initial set $\mathcal{S}$ of rule-based strategies for μ RTS as defined by the strategies of [SILVA *et al.* \(SILVA *et al.*, 2018\)](#). Their strategies can be described as 3 rule-based strategies with different parameter sets. We call their strategies Rush, Defense, and Military. As an example, the Boolean expression of a rule of the Rush strategy verifies if the player has a minimum number n of units of a given type before attacking the opponent. The parameters for this rule are the type of units trained and the number of units required to trigger an attack. [SILVA *et al.*](#) used 4 strategies derived from Rush, all of them with $n = 1$. Their Rush strategies differed only on the type of unit trained, with one strategy training each of the types: worker, light, ranged, and heavy—these strategies are named Worker Rush (WR), Light Rush (LR), Ranged Rush (RR), and Heavy Rush (HR). One can generate novel rush strategies by changing not only the type of unit trained, but also the value of n . Similarly, one can define novel Defense and Military strategies by changing their parameters. In this paper we vary the parameters of Rush, Defense, and Military to generate 100 strategies of each type, which results in a $\mathcal{Z}$ set of 300 strategies.³

2.8.3 Search Algorithms

We use Portfolio Greedy Search (PGS) ([CHURCHILL; BURO, 2013](#)) and Stratified Strategy Selection (SSS) ([LELIS, 2017](#)) in our experiments. We use PGS and SSS because both algorithms were developed to search in action-abstracted state spaces induced by strategies. Note, however, that one could use other algorithms such as A3N ([MORAES *et al.*, 2018](#)) to search in our action-abstracted spaces. We denote as PGS* and SSS* the search algorithms using an action abstraction derived by our evolutionary procedure. When deriving PGS*’s action abstraction, we use PGS to compute the values of V in the evolutionary procedure. Similarly, we use SSS to compute the values of V when computing SSS*’s action abstraction.

2.8.4 Maps

μ RTS matches can be played in maps of different sizes. In our experiments we use small, medium, and large map sizes. We test our action abstractions in maps of size $x \times x$ with $x \in \{8, 24, 32, 64\}$, where the map of 64×64 is a map from Blizzard’s StarCraft. All maps we use are symmetric, meaning that the players are indifferent to being assigned the role of “player 1” or “player 2”. In addition to being algorithmic dependent, all action abstractions are map

³ See our codebase for details of how the 300 strategies are generated: <https://github.com/julianmarino/evolutionary-action-abstractions.git>

dependent. That is, we run our evolutionary approach to generate an action abstraction for a given search algorithm and map. Every match played is limited by a number of game cycles and gameplay time. We use (4000, 20), (6000, 20), (7000, 25), and (12000, 30) as the limit values in our experiments, where the first number specifies the limit in game cycles and the second number the limit in minutes for maps of size x of 8, 24, 32, and 64, respectively.

2.8.5 Experiment Sets

Our empirical evaluation is divided into two parts. In the first part we apply our proposed evolutionary algorithm for solving the SSG in a μ RTS map of size 8×8 , which is small enough for us to compute a Nash equilibrium profile for the SSG. This experiment allows us to verify that our approach is able to converge to an ESS. We use SSS as the search algorithm and sets $\mathcal{Z}$ of size 5, 10, and 15 in this experiment. The sets $\mathcal{Z}$ are also derived from the Rush, Defense, and Military rule-based strategies.

In the second part we test the performance of PGS* and SSS* against state-of-the-art methods. We use in our experiments the MCTS version of Puppet Search (PS) (BARRIGA; STANESCU; BURO, 2017b), Strategy Tactics (STT) (BARRIGA; STANESCU; BURO, 2017a), which was the winner of the 2017 μ RTS competition (ONTAÑÓN *et al.*, 2018), Adversarial Hierarchical Task Network (AHT) (ONTAÑÓN; BURO, 2015), NaiveMCTS (ONTAÑÓN, 2017), Strategy Creation Through Voting (SCV) (SILVA *et al.*, 2018), the four instantiations of the Rush strategy explained above, WR, LR, HR and RR (STANESCU *et al.*, 2016), which are known to perform well in μ RTS (ONTAÑÓN *et al.*, 2018). Also, we include in our comparison versions of PGS and SSS that use an action abstraction induced by the set WR, LR, HR, and RR, denoted PGS^r and SSS^r, where “r” stands for Rush. We also use versions of PGS and SSS that use an action abstraction induced by 9 strategies, which are derived from the same Rush, Defense, and Military rule-based strategies used to generate our set $\mathcal{Z}$. We denote the variants of PGS and SSS that use the action abstraction induced by these 9 strategies as PGS^s and SSS^s, where “s” refers to SCV, as these 9 strategies are those used by SCV (SILVA *et al.*, 2018). Namely, PGS^s and SSS^s use the same 4 Rush strategies used with PGS^r and SSS^r, 4 Defense strategies, and 1 Military strategy; see SILVA *et al.* (SILVA *et al.*, 2018) for details. As explained before, the action abstractions we use with PGS* and SSS* are defined by our evolutionary approach with a $\mathcal{Z}$ set with 300 strategies, for all maps tested.

Since PGS* and SSS* have their action abstractions defined by a stochastic approach, we repeat the second experiment 5 times and report the average percentage of victories of one system against another. Matches that finish in a draw (e.g., by reaching the maximum number of game cycles or time limit) are counted as 0.5 victory for both systems. In each run of the second experiment we have each system evaluated play every other system 100 times on a given map. Although all our maps are symmetric, to guarantee fairness, in every run each system plays 50 matches against another system as “player 1” and the other 50 matches as “player 2”.

Figure 1 – Fraction of the only strictly dominant strategy WR in the population P of Algorithm 1 across 20 generations.

2.8.6 Evolutionary Parameters

In the first experiment we use a mutation probability u of 0.05, size of the population n of 100, the number of generations l of 100, and the size e of the elite population is set to 25. Since the maps are much larger in the second experiment, we set $n = 20$, $l = 30$, and $e = 5$.

2.9 Empirical Results

2.9.1 First Part: Convergence to an ESS

Due to the real-time constraints, the order in which the strategies are used to induce an abstraction matters to search algorithms. For example, let σ_1^r and σ_2^r be two rule-based strategies used to induce an abstraction. The abstraction induced with $\{\sigma_1^r, \sigma_2^r\}$ could be different from the abstraction induced with $\{\sigma_2^r, \sigma_1^r\}$ as, depending on the game state in which the search is applied, the algorithm might be able to evaluate only the action given by the first strategy, which could be the action given by strategy σ_1^r or σ_2^r , depending on the abstraction's ordering. This means that for RTS games, instead of having the powerset of $\mathcal{Z}$ as the strategy set for the SSG, one has all permutations of all subsets of $\mathcal{Z}$ as strategy set for the SSG. Our evolutionary procedure naturally deals with this combinatorial explosion as the individuals in the population are treated as totally ordered subsets of $\mathcal{Z}$. For the map of size 8×8 , we tested all permutations of subsets of the sets $\mathcal{Z}$ of size 5 and 10 against WR and proved WR to be a strictly dominant strategy for these sets $\mathcal{Z}$, and thus, the profile (WR, WR) to be a Nash equilibrium for the SSG.

Figure 1 shows the fraction of WR in the population across 20 generations of Algorithm 1 for the three SSGs considered. In all three cases the fraction of the population with the WR strategy quickly approaches one; for $|\mathcal{Z}| = 5$ the fraction of WR is near one in the third generation, while for $|\mathcal{Z}| = 10$ and $|\mathcal{Z}| = 15$ the fraction of WR is near one in the fourth

generation of the algorithm. The plot suggests that the optimal WR is stable after generation 4 in all three cases, as mutant strategies inserted in the population are quickly removed for being “less fit to survival” than WR.

During a generation, each individual is evaluated once against every other individual in the population. For a population with 100 individuals this results in 5,000 evaluations per generation. After four generations (20,000 evaluations), the population in the evolutionary procedure has stabilized with the optimal WR. As a base of comparison, there are 9,864,100 permutations for the set $\mathcal{Z}$ with 10 strategies. This shows that our evolutionary approach is able to converge to an ESS while performing a much smaller number of evaluations than the number of strategies available to the player.

2.9.2 Second Part: Comparison with the State of the Art

Table 1 shows the average percentage of wins of the column player against the row player for the three maps used in our experiment. As an example of how to read the table, PGS* wins on average 98.4 percent of its matches against SSS^s in maps 24×24 . We highlight the background of cells showing the results of both PGS* and SSS*, the two search methods using action abstractions automatically derived by our evolutionary method. In addition to PGS* and SSS*, we also present as column players the four baselines, which are the same search algorithms but using handcrafted actions abstractions: PGS^r, PGS^s, SSS^r, and SSS^s. The numbers are truncated to one decimal place. All results are significant with $p < .05$ according to pairwise comparisons performed with unpaired t-tests. Each t-test compares the average percentage of matches won by a column player against a row player.

Comparison with Baselines We start by comparing PGS* and SSS* with their baselines: PGS^r, PGS^s, SSS^r, and SSS^s. One observes that PGS* and SSS* are able to substantially outperform the baselines in direct matches. This can be observed in the table of results by looking at PGS*’s and SSS*’s highlighted rows. SSS^r is the baseline that wins the largest number of matches against either PGS* and SSS*, and that is only 9.5% of the matches played against SSS* in the 24×24 map. The percentage of matches won by a baseline against one of our methods reduces as one increases the size of the map. In particular, PGS* and SSS* win all their matches against the baselines in maps of size 64×64 .

PGS* and SSS* can also be compared with the baselines by observing the average winning percentage of the PGS and SSS variants against the other approaches. In general, PGS* and SSS* win more matches against other approaches than the baselines. A notable exception are the results against WR in the 24×24 map, where the baselines PGS^s and SSS^s win more matches than PGS* and SSS*. This is because the strategies used by PGS^s and SSS^s are quite effective against WR, but less effective in general—PGS* and SSS* are much stronger than PGS^s and SSS^s against the other methods. These results highlight that the action abstractions derived by

		PGS ^r	PGS ^s	PGS [*]	SSS ^r	SSS ^s	SSS [*]
Map 24 × 24	PS	81.8	81.6	99.8	57.1	75.2	99.2
	AHT	100.0	99.8	96.8	100.0	99.2	95.6
	STT	97.4	91.8	99.8	44.9	87.6	98.6
	NS	98.9	98.9	100.0	86.5	98.0	100.0
	RR	100.0	100.0	100.0	81.2	100.0	100.0
	HR	100.0	100.0	100.0	97.0	99.8	100.0
	WR	60.4	90.3	78.8	44.8	82.6	66.6
	LR	92.9	88.6	100.0	19.8	84.8	99.2
	SCV	99.8	99.2	98.2	95.0	98.2	97.4
	PGS ^r	-	42.4	99.6	34.7	40.7	93.5
	PGS ^s	57.6	-	97.0	43.0	45.0	91.4
	PGS [*]	0.4	3.0	-	5.4	1.6	41.7
	SSS ^r	65.3	57.0	94.6	-	40.7	90.5
	SSS ^s	59.3	55.0	98.4	59.3	-	93.2
	SSS [*]	6.5	8.6	58.3	9.5	6.8	-
		PGS ^r	PGS ^s	PGS [*]	SSS ^r	SSS ^s	SSS [*]
Map 32 × 32	PS	37.9	48.4	99.8	52.1	47.8	99.6
	AHT	100.0	100.0	100.0	100.0	100.0	100.0
	STT	84.8	22.7	100.0	88.6	23.4	100.0
	NS	99.2	99.3	100.0	99.6	98.9	100.0
	RR	100.0	100.0	100.0	100.0	100.0	100.0
	HR	98.8	99.2	100.0	99.8	98.0	100.0
	WR	100.0	100.0	94.2	100.0	100.0	100.0
	LR	1.8	8.2	96.8	16.2	7.2	99.0
	SCV	11.9	29.4	86.5	56.5	27.8	91.5
	PGS ^r	-	55.2	99.6	96.0	55.7	99.6
	PGS ^s	44.8	-	99.6	97.2	51.0	100.0
	PGS [*]	0.4	0.4	-	1.0	0.0	51.6
	SSS ^r	4.0	2.8	99.0	-	2.4	99.8
	SSS ^s	44.3	49.0	100.0	97.6	-	100.0
	SSS [*]	0.4	0.0	48.2	0.2	0.0	-
		PGS ^r	PGS ^s	PGS [*]	SSS ^r	SSS ^s	SSS [*]
Map 64 × 64	PS	0.3	0.0	83.8	0.0	0.0	77.5
	AHT	100.0	100.0	99.5	100.0	100.0	100.0
	STT	25.2	0.3	95.2	20.7	2.0	91.0
	NS	100.0	100.0	100.0	100.0	100.0	100.0
	RR	87.7	100.0	100.0	94.7	98.3	100.0
	HR	99.7	100.0	100.0	99.3	100.0	100.0
	WR	60.5	82.0	98.8	75.7	72.3	97.7
	LR	54.3	78.0	100.0	59.0	66.0	100.0
	SCV	1.5	1.5	86.0	9.3	1.8	73.8
	PGS ^r	-	50.3	100.0	68.0	50.7	100.0
	PGS ^s	49.7	-	100.0	67.0	45.3	100.0
	PGS [*]	0.0	0.0	-	0.0	0.0	36.8
	SSS ^r	32.0	33.0	100.0	-	39.5	100.0
	SSS ^s	49.3	54.7	100.0	60.5	-	100.0
	SSS [*]	0.0	0.0	62.8	0.0	0.0	-

Table 1 – Average percentage of matches won by the column player against the row player for maps of different sizes.

our evolutionary approach can be far superior to those derived from existing strategies. This is likely because existing strategies (e.g., those used by [SILVA *et al.* \(SILVA *et al.*, 2018\)](#)) are designed to play the game directly, and not to be combined with other strategies through action abstractions. Since we use a large set of strategies, our evolutionary procedure might encounter strategies that if combined to induce an action abstraction, provide a search algorithm with a set of promising and complementary actions.

Comparison with the State of the Art STT was the winner of the 2017 μ RTS competition ([ONTAÑÓN *et al.*, 2018](#)). Both PGS* and SSS* substantially outperform STT in all maps tested, with 91% being the smallest average percentage of victories obtained by one of our approaches, which occurred in matches played by SSS* in the 64×64 map. Our approaches also outperform by a large margin PS, which is a previous version of STT. Our scheme for generating our set $\mathcal{Z}$ is similar to the search performed by PS and STT as they also alter the parameter values of expert-designed strategies. One possible explanation for these results is that our approach of using a set of strategies to induce an action abstraction as a preprocessing step can be more effective than the online searching for parameter values. We believe this to be true because our evolutionary method filters out weak strategies while solving the SSG, thus allowing the planning systems to focus their search on more promising actions. By contrast, PS and STT might use the limited time available for planning to evaluate weak strategies.

SCV ([SILVA *et al.*, 2018](#)) and LR also performed quite well in the 2017 μ RTS competition ([ONTAÑÓN *et al.*, 2018](#)). PGS* and SSS* are able to outperform both of them also by a large margin. The comparison with SCV is important because the rule-based strategies we use to generate our set $\mathcal{Z}$ is derived from the strategies used in SCV. Moreover, SCV also generates a large set of strategies from a small set of options. The superior performance of our approaches against SCV suggests that our system for generating a large set of strategies and thus selecting a subset to induce action abstractions can be superior to other schemes that generate a large set of strategies but use them to play the game directly.

2.10 Limitations

Although our evolutionary method for generating action abstractions allowed existing algorithms to substantially outperform state-of-the-art methods in the problem domain of μ RTS, our approach still has limitations, which point to interesting directions of future research. Although we are able to generate a large number of strategies from a small number of existing ones, the strategies we generate to compose $\mathcal{Z}$ are still somewhat restricted to the initial set of expert-designed strategies (e.g., the strategies we generate by changing the parameters of rule-based strategies are still restricted to the actions from the initial set of strategies). We expect to generate stronger action abstractions with our evolutionary approach should the set $\mathcal{Z}$ contained a larger diversity of strategies. Another limitation of our approach is due to the fact that

our approach considers only pure strategies for the SSG (i.e., a single subset of strategies from $\mathcal{Z}$ is returned after the last generation of our method). Since the SSG is a simultaneous-move game, one might need to play a mixed strategy to optimize her payoffs. We intend to investigate the use of mixed strategies for the SSG in future works.

2.11 Conclusions

In this paper we showed that the problem of generating action abstractions for extensive-form games can be cast as a problem of selecting a subset of strategies from a pool $\mathcal{Z}$ of options. We then modeled the subset selection problem as a two-player game, which we named SSG. We presented an evolutionary procedure to solve SSGs. Another contribution of this paper was to show that a large set $\mathcal{Z}$ of strategies can be generated by a small set of what we called rule-based strategies, by changing the parameter values of the strategies. Empirical results on a small map of μ RTS showed that our method is able to converge to an optimal solution to the SSG. Results on larger maps showed that PGS and SSS using action abstractions generated by our approach was able to outperform versions of the same algorithms using existing action abstractions. Our results also showed that PGS and SSS with our action abstractions outperformed all state-of-the-art systems tested, which included the best performing methods from the 2017 μ RTS competition.

2.11.0.0.1 Acknowledgements

This research was partially supported by CNPq, Capes, and FAPEMIG. The research was carried out using the computational resources of the Center for Mathematical Sciences Applied to Industry (CeMEAI) funded by FAPESP (grant 2013/07375-0), and the cluster Jupiter from Universidade Federal de Viçosa. We thank the anonymous reviewers for great suggestions.

EVOLVING INTERPRETABLE STRATEGIES FOR ZERO-SUM GAMES

Authors:

Julian R. H. Mariño (julianmarino@usp.br)

Claudio Toledo (claudio@icmc.usp.br)

Abstract

The present paper introduces Gesy, a genetic programming approach to script synthesis for zero-sum games. We will explore the sum-zero game context in Real-Time Strategy (RTS) games, where players must look for strategies (planning of actions) to maximize their gains or minimize their losses. The goal is to solve the script synthesis problem, which demands the synthesis of a computer program from a space of programs defined by a domain-specific language (DSL). The synthesized program must encode a practical strategy for zero-sum games. Empirical results validate Gesy using the μ RTS platform, an academic testbed game that presents the main features found in RTS commercial games. The results show that our method provides interpretable strategies that are competitive with state-of-the-art search-based approaches in terms of play strength. Moreover, once synthesized, scripts require only a tiny fraction of the time needed by search-based methods to decide on the agent's next action.

3.1 Introduction

The sum-zero game holds an interaction among two players, with one player gains meaning another player loses. Chess is a classic sum-zero example, while selecting options and futures trading can be view as a sum-zero game in the stock market. In both cases, if someone

gains, another one losses. A sequence of actions must define proper strategies mainly within real-time scenarios (BOSC *et al.*, 2015). Thus, the sum-zero scenario also happens in Real-Time Strategy (RTS) games, where players must look for strategies (planning of actions) to maximize their gains or minimize their losses.

This paper focuses on the script synthesis approach for zero-sum games instead of search-based systems. The effectiveness of search-based systems relies on their ability to look ahead into the state space and choose actions that maximize the player's utility. This ability often allows search algorithms to derive more expressive strategies than those encoded in hard-coded computer programs. The search-based systems have shown to be stronger than strategies hard-coded in computer programs for zero-sum games. (CHURCHILL; BURO, 2013; LELIS, 2017; BARRIGA; STANESCU; BURO, 2017b; BARRIGA; STANESCU; BURO, 2017a; ONTAÑÓN; BURO, 2015; ONTAÑÓN, 2017; MORAES *et al.*, 2018; YANG; ONTANÓN, 2019).

However, in contrast with computer programs, search methods often require more computational resources to decide the agent's following action, as they usually evaluate many actions during the search. The response time is essential in computer game domains, where the planning algorithm is part of an extensive system with resources shared among several processes. Thus, we propose a script synthesis approach to generate interpretable scripts, which encode effective strategies for zero-sum games, removing the need for planning actions and the related response time bottleneck from search-based methods.

In addition to response time, there is another advantage of using a hard-coded computer program instead of search-based algorithms for planning: strategies encoded in hard-coded computer programs can be interpretable. Plans derived by search-based methods often lack explainability (CHAKRABORTI *et al.*, 2019). This advantage of computer programs might be exploited by domain experts who want to understand artificial agents' strategies and learn from them. Moreover, game programmers could use computer-generated code instead of developing strategies from scratch to create novel ones in a human-machine collaborative task. In this paper, we refer to computer programs encoding strategies for playing a game as scripts.

There is also the need to develop Artificial Intelligence (AI) methods for generating explainable and understandable agent behaviors. This aspect becomes more relevant for domains where trustworthy, explainable, and transparent agent decisions are essential (ADADI; BERRADA, 2018). The authors in (JACOB; DEVLIN; HOFMANN, 2020) report the challenges and gaps to improve the design of intelligent agents from the perspective of game developers in the videogame industry. The adaptation of the behavior of such agents to unforeseen situations is one among other features to move forward.

A Domain-Specific Language (DSL) (DEURSEN; KLINT; VISSER, 2000) can define the space of possible scripts able to be synthesized; however, searching for scripts with effective strategies is still challenging. DSLs often define an ample search space, and in contrast with traditional program synthesis problems, the evaluation of scripts in games usually requests to

play or simulate a match.

Thus, the main contribution of this paper is Gesy, a genetic programming approach to script synthesis for zero-sum games. Gesy evolves an initial population of scripts through genetic operators, searching for those scripts in the space of programs defined by a DSL that maximize a fitness function. The fitness of a script is the number of wins in matches played against other scripts. Since these scripts define a strategy for RTS games, the results of Gesy are validated using μ RTS (ONTAÑÓN, 2013), which is an academic testbed game that emulates the most relevant features found in RTS commercial games. The results show that Gesy synthesizes interpretable scripts from the DSL that are competitive against state-of-the-art search-based methods.

3.2 Related Work

We review related works focused on synthesizing scripts for games. The so-called Dynamic Scripting (DS) is a reinforcement-learning-based technique that synthesizes scripts for zero-sum role-playing games. DS extracts rules according to a learned policy (SPRONCK; SPRINKHUIZEN-KUYPER; POSTMA, 2004), which separates rules following the type of agents related to them. Each rule has an associated weight stating its chance to be inserted in a script. The weights are modified after a match based on their contribution to improving the agent’s behavior. The authors in (PONSEN; SPRONCK, 2004; DAHLBOM; NIKLASSON, 2006) applied a modified versions of the original DS to zero-sum Real-Time Strategy (RTS) games. In these DS works, the structure of the resulting script is limited to a fixed number of consecutive if-then clauses. Therefore, DS becomes a method for synthesizing scripts that follows a pre-defined sketch (SOLAR-LEZAMA, 2009). By contrast, our method searches in the space of scripts defined by a context-free grammar that allows for the synthesis of scripts with different code structures. Thus, we are dealing simultaneously with the problems of choosing the rules for scripts and effective program structures. We explored the synthesis of scripts for RTS games in (MARINÑO *et al.*, 2021) by proposing a novel method to reduce the DSLs for the synthesis of scripts. On the other hand, the current paper focuses on the algorithms that can search over a fixed DSL given as input.

The work in (BENBASSAT; SIPPER, 2011a) introduced genetic programming approaches for synthesizing computer programs in zero-sum board games. However, in contrast with our proposed method, the authors in (BENBASSAT; SIPPER, 2011a) focus on the synthesis of programs to work as evaluation functions and pruning functions, respectively. We are synthesizing programs to encode strategies that allow playing the game directly.

Program synthesis approaches have been applied to games that are not zero-sum. In (CANAN *et al.*, 2018), the authors use an evolutionary algorithm to generate strategies to play Hanabi, a multi-player cooperative game. Their method generates sequences of if-then

rules to play the game similar to DS instead of searching for more complex code structures such as for-loops. A genetic programming approach in (FREITAS; SOUZA; BERNARDINO, 2018) evolves controllers from a context-free grammar for a Mario AI simulator. A method in (BUTLER; TORLAK; POPOVIĆ, 2017) synthesizes strategies encoded as condition-action rules for the puzzle game nonograms. Our approach will synthesize more complex code structures than (FREITAS; SOUZA; BERNARDINO, 2018) and (BUTLER; TORLAK; POPOVIĆ, 2017), and while they deal with single-agent problems, we deal with multi-player zero-sum games.

Program synthesis has been extensively investigated in other domains, including Code Repair (JOBSTMANN; GRIESMAYER; BLOEM, 2005), Code Suggestions (PERELMAN *et al.*, 2012), Data Wrangling (GULWANI, 2016), and String Transformations (GULWANI, 2011). The techniques include search-based methods, such as brute-force search (ALUR *et al.*, 2013), hybrid systems that combine search with a learned function (LIANG; JORDAN; KLEIN, 2010; MENON *et al.*, 2013), constraint satisfaction (JHA *et al.*, 2010), and machine learning (BALOG *et al.*, 2017). The synthesized programs often must satisfy user specifications expressed as constraints such as input-output examples in these domains. By contrast, in zero-sum games, one has as input a perfect model of the game, and constraints are not readily available for evaluating synthesized scripts.

Genetic Programming (GP) is a type of evolutionary algorithm that evolves computer programs (KOZA, 1992) using principles inspired by biological evolution. GP approaches have been tested in several domains, such as job shop scheduling (NGUYEN *et al.*, 2019), feature construction and selection in classification models (TRAN; XUE; ZHANG, 2016), neural networks (SUGANUMA; SHIRAKAWA; NAGAO, 2017) and anomaly detection (CAO; NICOLAU; MCDERMOTT, 2016). Our paper is the first work applying a GP approach to synthesize scripts that encode strategies for planning on zero-sum real-time domains to the best of our knowledge.

3.3 RTS games and μ RTS

Real-time strategy (RTS) games usually aim to fight for resources and eliminate enemy units and buildings within a warfare scenario. The scenarios provided by RTS games can be seen as a testbed for real-time planning, and decision making under uncertainty (ONTAÑÓN, 2013). The domains in RTS games usually request search algorithms able to find a satisfactory action from a large number of options, with the planning of actions being built within milliseconds (MARINO *et al.*, 2019).

The moves executed in RTS games have features reported in (SUN *et al.*, 2017) as durative actions, dynamic environment, and vast state space. The players execute simultaneous actions based on short-time decisions and employing multiple units. These actions are durative since the players perceived their effects later in the game; therefore, actions can change the game

scenario and lead to a dynamic environment. The dynamic requests replanning since current plans can become infeasible. Finally, many possible states are available. The authors in (SUN *et al.*, 2017) mention that the state space of chess game is around 10^{150} , while the possible states reach 10^{1685} in StarCraft for a unit location, assuming a 128×128 map with 200 units.

We propose synthesizing scripts with genetic programming using the academic μ RTS (ONTAÑÓN, 2013) prototype as a testbed. μ RTS is a simplified RTS game designed for AI research, extensively employed to validate methods (YANG; ONTANON, 2021), which satisfies game features such as durative actions and real-time decision making. It is possible to set maps with different sizes, several bases, barracks, workers, level (low or high) of power with fast or slow melee fight units, and range attack units. All these setting possibilities allow reproducing the main points about game actions.

Figure 2b shows a screenshot of a μ RTS game with two players playing against each other. Both players are following the strategy encoded by the script in Figure 2a. The units of player one have a blue line around them, and units of player two have a red line around them. The strategy encoded by the script in Figure 2a uses the base (the white square) to train worker units, represented by small gray circles, tagged with a number in the figure. If there is no units harvesting, they will go to harvest the closest resource. If there is at least one unit harvesting, all worker units u go to attack the nearest opponent unit. Namely, units one and four are harvesting resources, and units two and three are going in the direction of their opponent units to attack them.

```

1 harvest()
2 for(each_unit_u)
3   if(Distant(u,1))
4     train(u)
5     attack(u)
6 build()

```

(a) Script for playing μ RTS

(b) Gameplay on μ RTS

Figure 2 – Figure 2a shows an script wrote for μ RTS, and 2b shows a screenshot of a μ RTS game that is being played by two players that are following the strategy encoded by the script in figure 2a.

3.4 Script Synthesis in Zero-sum Games

Let $\mathcal{G} = (\mathcal{N}, \mathcal{S}, s_{init}, \mathcal{A}, \mathcal{R}, \mathcal{T})$ be a zero-sum game, where $\mathcal{N} = \{i, -i\}$ is the set of players, $\mathcal{S} = \mathcal{D} \cup \mathcal{F}$ is the set of states, where $\mathcal{D}$ denotes the set of non-terminal states and $\mathcal{F}$ the set of terminal states, $s_{init} \in \mathcal{D}$ is the start state of the game, and $\mathcal{A}(s)$ is the set of actions a player can perform in state $s \in \mathcal{D}$.

A script is a strategy represented as a function $p(s)$ that returns a legal action $a \in \mathcal{A}_i$ for player i at state s . Player i has a utility value of the game rooted at state s denoted by $\mathcal{V}(s, p_i, p_{-i})$. This function indicates player i 's utility, if player i uses p_i and player $-i$ uses p_{-i} to play the game. $\mathcal{G}$ is a zero-sum game, therefore the utility of player $-i$ is given by $-\mathcal{V}(s, p_i, p_{-i})$.

Let $G = (V, \Sigma, R, S)$ be a context-free grammar where V is a set of non-terminals, Σ is a set of terminals, R is the relations of the grammar, and S is the initial symbol. As an example, consider the following context-free grammar, which we refer to as G_1 .

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

This grammar accepts programs with either an if-then or an if-then-else structure. The Boolean formula of the if clause can be either b_1 , b_2 , or b_3 , while the commands can be either c_1 , c_2 , or c_3 . In the example, we have that $V = \{S, B, C\}$, $\Sigma = \{b_1, b_2, b_3, c_1, c_2, c_3\}$, R are the eight relations specified above (e.g., $B \rightarrow b_1$ is a relation), and S is the start symbol. When using G_1 to define a DSL, derivation trees represent the possible synthesized scripts, where the root 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 3 illustrates the derivation tree that represents the script “if (b_1) then c_3 ” accepted by G_1 .

Figure 3 – Derivation tree representation of a synthesized script using G_1 .

In a script synthesis problem, we have a zero-sum game $\mathcal{G}$ and a DSL D , which can be specified by a context-free grammar G , to synthesize a script p in D that encodes a strategy

for playing $\mathcal{G}$. We employ the terms DSL and grammar interchangeably in this paper, once context-free grammar can define a DSL. For any non-terminal state s of the game, $p(s)$ must return a legal action $a \in \mathcal{A}$ for the player acting at state s . As an optimization problem, the script synthesis problem approached has to synthesize a script $p_i \in \llbracket D \rrbracket$ that encodes a strategy for player i and maximizes the value of the game. In contrast, player $-i$ follows the strategy encoded by script p_{-i} from $\llbracket D \rrbracket$ that minimizes the value of the game. The problem can be formulated as follows,

$$\max_{p_i \in \llbracket D \rrbracket} \min_{p_{-i} \in \llbracket D \rrbracket} \mathcal{V}(s_{init}, p_i, p_{-i}). \quad (3.1)$$

Previous works focused on the generation of scripts with code structures limited to a fixed number of consecutive if-then clauses (SPRONCK; SPRINKHUIZEN-KUYPER; POSTMA, 2004; PONSEN; SPRONCK, 2004; DAHLBOM; NIKLASSON, 2006; CANAAN *et al.*, 2018). These methods can be seen as synthesizers that search for scripts in a DSL space defined by a fixed sketch (SOLAR-LEZAMA, 2009). The problem with these approaches is that effective strategies could demand different code structures than a pre-defined sequence of if-then rules. The DSL we used in our experiments offers a more expressive search space.

3.5 Genetic Programming to Synthesize Scripts

In this section, we introduce Gesy, a genetic programming approach to script synthesis. Given a DSL D , Gesy searches for the script $p_i \in \llbracket D \rrbracket$ that approximates a solution to the script synthesis problem defined by equation 4.1. Algorithm 2 describes Gesy.

Algorithm 2 – Gesy

Require: zero-sum game $\mathcal{G}$, fitness function Ψ , grammar G , number of generations g , size of the population n , number of individuals for fitness f , number of individuals w selected for tournament, number of individuals t used in recombination, probability for mutation m , size of the elite e , number of invaders a .

Ensure: Script p_i .

```

1:  $l \leftarrow 0$ 
2:  $\Phi \leftarrow \text{Initialize}(G, n)$ 
3: while  $l < g$  do
4:    $F \leftarrow \text{Opponents\_Set}(\Phi)$ 
5:    $\text{Evaluate}(\mathcal{G}, \Phi, \Psi, F)$ 
6:    $T \leftarrow \text{Select}(\Phi, t)$ 
7:    $\Phi' \leftarrow \text{Perform\_Crossover}(T)$ 
8:    $\Phi' \leftarrow \text{Mutate}(G, \Phi')$ 
9:    $E \leftarrow \text{Select\_elite}(\Phi)$ 
10:   $E' \leftarrow \text{Mutate\_elite}(G, E)$ 
11:   $I \leftarrow \text{Generate\_invaders}(G, a)$ 
12:   $\Phi \leftarrow \Phi' \cup E \cup E' \cup I$ 
13:   $l \leftarrow l + 1$ 
14: end while
15:  $\text{Evaluate}(\mathcal{G}, \Phi, \Psi)$ 
16: return  $\text{argmax}_{p_i \in \llbracket D \rrbracket} \Psi(p_i)$ 
```

The algorithm receives as input a zero-sum game $\mathcal{G}$, a fitness function Ψ and a grammar G that defines a DSL D . The parameters g and n state the number of generations and population length. The parameter f establishes the number of individuals randomly selected for a

round-robin that will state the fitness value, as described later. The parameter w defines the tournament's size for reproduction selection, while t gives the number of individuals that go through recombination. The parameter m , e , and a are the probability for mutation, the size of the elite set of individuals, and the number of invaders (individuals randomly generated and inserted in the population).

The procedure starts with the **initialization** procedure that generates a set of n scripts belonging to Φ (line 2). The set of scripts Φ is called population, with an individual encoding a script representation. Each individual is generated randomly according to the grammar G . Let's illustrate the initialization for a μ RTS script taking the grammar G_2 as follows.

$$\begin{aligned} S &\rightarrow C \mid C \text{ for}(F) \{ \text{if}(B) \text{ then } C \} \mid CS \\ C &\rightarrow \text{attack}() \mid \text{harvest}() \mid \text{build}() \mid \text{attack}()C \mid \text{harvest}()C \mid \text{build}()C \\ B &\rightarrow \text{Distant}(1) \mid \text{Quantity}(2) \\ F &\rightarrow \text{each_unit_u} \end{aligned}$$

The terminals $\text{attack}()$, $\text{harvest}()$, $\text{build}()$ and $\text{train}()$ are domain-specific functions that return an action for a given state of the game. The terminals $\text{DistantToEnemy}(1)$ and $\text{QuantityUnits}(2)$ are Boolean expressions that return true, if there is an opponent unit in a distance of one position from any controlled unit, and if there are two units alive in the game, respectively. The terminal each_unit_u is a For-Loop function that iterates over all the units.

The initialization will start with a symbol S , which is encoded as a root node in the derivation tree structure as shown in Figure 4b. This node represents the selection of one S rules to be applied. G_2 illustrates three possible rules and, if the second rule from S is selected, the individual encoding can be seen as the tree structure in Figure 4b. This rule requests a selection of terminals from C , B , and F in G_2 ; therefore, the initialization will randomly select terminals from C , B , and F . Figure 4b shows the terminals $\text{attack}()$ and $\text{harvest}()$ taken from C , each_unit from F and $\text{DistantToEnemy}(1)$ from B . The initialization can select another rule from S , repeating the previous steps, since the number of script lines does not exceed a maximum length. Figure 4a shows the script encoded in the derivation tree.

After the initialization, the algorithm performs g iterations, which are known as generations. The set F with f opponent individuals is randomly selected from Φ (line 4). These opponents are used as a testbed to evaluate the whole population, including themselves. Each individual in the population is **evaluated** according to the set of fitness function Ψ (line 5), current game $\mathcal{G}$, and set of individuals F . The fitness value of a script p , given by Ψ_p , is the number of victories of p against each script in F following Algorithm 3. Each individual from population Φ plays against the opponents in set F . The fitness of the individual ($\Psi_p \in \Psi$) counts each victory against one opponent under the game match $\mathcal{G}$.

Figure 4 – Example of a script randomly created by the initialization procedure in Gesy. In figure 4a it is shown the pseudocode of the script, and in figure 4b it is shown the derivation tree that represents the script.

Algorithm 3 – Evaluate($\mathcal{G}, \Phi, \Psi, F$)

```

1: for  $p \in \Phi$  do
2:    $\Psi_p \leftarrow 0$ 
3:   for  $f \in F$  AND  $f \neq p$  do
4:      $result \leftarrow Match(\mathcal{G}, f)$ 
5:     if  $Result == Win$  then
6:        $\Psi_p \leftarrow \Psi_p + 1$ 
7:     end if
8:   end for
9:    $\Psi \leftarrow \Psi \cup \{\Psi_p\}$ 
10: end for

```

The fitness value of each individual belonging to Ψ impacts the selection for reproduction from Φ , once we are applying a tournament (TALBI, 2009). The individual with the largest fitness value, belonging to a subset of w randomly selected individuals from Φ , is added to T (line 6 in Algorithm 2). This tournament is repeated while $|T| < t$.

Next, two individuals are randomly selected from T to execute a **crossover** (KOZA, 1992) (recombination operator in line 7 of Algorithm 2). The **crossover** executes based on the derivation tree that encodes individuals. Each individual selected for reproduction has a random cut point that split the tree structure in two disjoint parts. Let's illustrate from individuals p_1 and p_2 in Figure 5, where dashed lines illustrate the disjoint part selected to exchange.

The exchange of sub-trees will create two new individuals p_3 and p_4 included in Φ' . The **crossover** will move code blocks to avoid infeasibility, but the procedure verifies anyway to guarantee that the new individuals satisfy the grammar. If there is a problem, a new cut point is defined. The crossover is repeated while $|\Phi'| < n - 2e - a$.

The population Φ in the next generation will have those $n - 2e - a$ individuals from Φ' plus e mutated individuals and a individuals named as invaders. A **mutation** (KOZA, 1992) scheme is applied over each individual in Φ' (line 8). The mutation procedure iterates through each terminal node of the derivation tree, replacing by another terminal node accepted by G with a probability m . Figure 6 illustrates mutation over individual p_5 , where the terminal node *build()* is replaced by *train()*, and *train()* is replaced by *build()*. The mutation rate is not satisfied for node *attack()* that remains the same.

It is necessary to highlight two points about the operators: infeasible scripts and the size

Figure 5 – Example of crossover following grammar G_2 Figure 6 – Example of mutation following grammar G_2

of individuals. There are no infeasible scripts during the initialization since the operator follows the grammar G . The crossover and mutation operators also do not generate infeasible scripts since they move or redefine an entire code block. For instance, a code block always moves to a new line during the crossover, and it is not allowed to nest it in another code block to avoid infeasibility. The mutation follows grammar rules by changing only terminal nodes. It is possible to increase the size of individuals when applying crossover or mutation, which could impact the evolutionary process. Therefore, we define a maximum length of 10 commands, which are domain-specific functions from our DSL. In this case, we execute some basic repair procedures such as removing code lines that exceed the maximum value, generating a new code block to be inserted, or selecting another cut point for recombination.

The **elitism** procedure (line 9) selects a subset E of the e individuals with the best fitness values from Φ . Next, we apply mutation over a copy p' of each individual p in E , adding p'

to a new set E' (line 10). This mutation differs from the previous one in line 8 by replacing at least one terminal node. If no mutated node happens after iterating through all internal nodes, we randomly select an internal node and replace it. The idea is to guarantee mutated versions of the best individuals from the current generation. Another way to increase diversity is to include a invaders, which are randomly created individuals, following the grammar, to be added in the set of individuals I (line 11).

Finally, the population Φ of the next generation is determined by the union among sets Φ' , E , E' and I (line 12). Moreover, at the end of each generation, Φ is composed of individuals generated by crossover and mutation (Φ'), the elite individuals of the previous population (E), a set of mutated versions of the elite individuals (E'), and randomly generated individuals (I). The algorithm returns the individual p with the best fitness value in Φ at the last generation (line 16). Our implementation of Gesy is available online.¹

3.6 Empirical Methodology

This section presents the DSL designed to define a space of programs for μ RTS and describe the experiments using it, where we compare *Gesy* and *Gesy_s* with state-of-the-art approaches in terms of strength of play and computation time. We also analyze the interpretability of the scripts synthesized by our methods.

3.6.1 A DSL for μ RTS

We use the following DSL in our experiments:

$$\begin{aligned}
 S_1 &\rightarrow C S_1 \mid S_2 S_1 \mid S_3 S_1 \mid \varepsilon \\
 S_2 &\rightarrow \text{if}(B) \text{ then } \{C\} \mid \text{if}(B) \text{ then } \{C\} \text{ else } \{C\} \\
 S_3 &\rightarrow \text{for (each unit } u) \{S_4\} \\
 S_4 &\rightarrow C S_4 \mid S_2 S_4 \mid \varepsilon \\
 B &\rightarrow b_1 \mid b_2 \mid \dots \mid b_m \\
 C &\rightarrow c_1 C \mid c_2 C \mid \dots \mid c_n C \mid c_1 \mid c_2 \mid \dots \mid c_n \mid \varepsilon
 \end{aligned}$$

Terminals $c_1, c_2, \dots, c_n$ are domain-specific functions that return an action for a given state of the game. An example of a domain-specific function is `attack(Worker, closest, u)`. Based on the Euclidean distance, this function sends worker unit u of player i to attack the closest opponent unit. Terminals $b_1, b_2, \dots, b_m$ are Boolean expressions that return a Boolean value for a given state of the game. An example of a Boolean expression is `IsDistantFromEnemy(4)` which returns true if player i has at least one unit u at a Euclidean distance of 4 or less from an enemy unit. S_1 is the initial symbol, ε is an empty string, and the DSL has $n = 101$ domain-specific

¹ <<https://github.com/julianmarino/Gesy>>

functions and $m = 356$ Boolean expressions. We added a full description of our DSL in the Appendix.

3.6.2 Experiment Setup

We evaluate Gesy and Gesy_s against the state-of-the-art methods. Gesy_s has, instead of randomly generating all individuals of the initial population Φ , four scripts written by experts in its initial population of scripts: Worker Rush (WR), Light Rush (LR), Heavy Rush (HR), and Ranged Rush (RR). These scripts are available in the μ RTS repository². We encoded those scripts in our DSL by defining derivation trees and added them to the initial population of Gesy_s. This experiment allows us to verify the influence of what could improve the initial population. However, the more basic version, Gesy, generates scripts from scratch, also reaching a relevant performance as the computational results will show next.

We use the following search-based methods as competitors: Portfolio Greedy Search (PGS) (CHURCHILL; BURO, 2013), Stratified Strategy Selection (SSS) (LELIS, 2017), the MCTS version of Puppet Search (PS) (BARRIGA; STANESCU; BURO, 2017b), Strategy Tactics (STT) (BARRIGA; STANESCU; BURO, 2017a), Adversarial Hierarchical Task Network (AHT) (ONTAÑÓN; BURO, 2015), NaïveMCTS (NS) (ONTAÑÓN, 2017), A3N (MORAES *et al.*, 2018), and Naive Guided Sample (GNS) (YANG; ONTANÓN, 2019). The parameter values of those methods were configured as described by the authors. In addition to search-based methods, we also include scripts WR, LR, HR, and RR as opponents. All these methods are benchmarks validated by μ RTS competitions in the IEEE Conference on Games (CoG)³. The mentioned search-based methods spend the available time to return the best possible action, and such response time is 100 milliseconds. On the other hand, scripts have already encoded what to do, spending just a tiny fraction of that available time to return action.

The gameplay happens within a total of four maps. The first map has a 18×8 size, with no worker units neither resources to be harvested. The players start with four ranged units placed in front of three heavy units. The second and third maps are maps of sizes 8×8 and 16×16 , respectively. In these maps, players start with a base and a worker. The fourth map is a map of size 9×8 , and the players start with a base, a worker, and the resources are placed as a wall, which divides the map into two parts, separating both players. The maps with sizes 8×8 , 16×16 , and 9×8 are standard maps used in μ RTS competitions. We added the map with 18×8 size to test our approach when synthesizing scripts for a combat scenario. In this case, we expect search algorithms to perform better due to the need for fine micromanagement of units. Thus, 8×8 map requires a specific behavior of workers, map 9×8 demands ranged units, map 16×16 allows us to test the scalability of our approach, and 18×8 evaluates scripts in a combat-only scenario. Figure 7 summarizes the overall experiments for each map.

² <<https://github.com/santiontanon/microrts>>

³ <<https://sites.google.com/site/micrortsaicompetition/competition-results/2020-cog-results>>

Figure 7 – The experiments execute round-robin tournaments from the best individuals found at each method run. The scripts play against each other to assess their performance, and the winner plays against state-of-art related methods.

Our experiment executes five independent runs of Gesys and Gesys_s over each map. We determine the winner script as shown by Figure 7 performing a round-robin tournament, among the best scripts found at each run. At this point, the best scripts play against each other ten times, five as player one and five as player two. The player one and player two roles aim to ensure a fair match within μ RTS. Next, the winner scripts from Gesys and Gesys_s will play against all the opponents from the literature previously mentioned. We repeat ten times the experiment flow described in Figure 7 to generate winner scripts, since Gesys and Gesys_s can synthesize different best final scripts from distinct executions over the same map. Also, each winner script will play ten matches against the opponents at the end of the experiment in Figure 7. To sum up, Gesys and Gesys_s run, respectively, 50 times for each map, returning ten winner scripts that will play ten matches against each scripts and search-based method from literature.

The proposed methods are set with number of generations $g = 100$, size of population $n = 60$, number of individuals for fitness $f = 9$, number of individuals for selection $w = 24$, number of individuals for tournament $t = 15$, mutation probability $m = 0.3$, size of the elite $e = 6$, and number of invaders $a = 6$, for 18×8 , 8×8 and 16×16 maps. The configuration changes for 9×8 map, where we increase the size of the population to $n = 80$, number of individuals for selection to $w = 32$, number of individuals for tournament to $t = 20$, and the number of invaders to $a = 8$. These parameter values increase since the 9×8 map is the hardest. The 9×8 map structure requires particular behaviors from the player, where an equilibrium between harvesting and the training of Ranged units has shown to be a good strategy. Each run takes approximately 3 hours for maps of sizes 18×8 and 8×8 , and 5 hours for 9×8 , and 16×16 maps. All the parameters values of Gesys and Gesys_s were chosen empirically from preliminary experiments.

Map 18×8															
	WR	LR	HR	AHT	RR	SSS	PGS	NS	PS	STT	GNS	A3N	Gesy	Gesy _s	Avg.
PGS	100.0	100.0	100.0	100.0	100.0	50.0	0.0	45.0	50.0	55.0	50.0	35.0	0.0	0.0	60.4
NS	85.0	75.0	100.0	85.0	70.0	60.0	55.0	0.0	50.0	50.0	50.0	45.0	61.0	56.0	64.8
PS	100.0	100.0	100.0	100.0	100.0	50.0	50.0	50.0	0.0	50.0	40.0	30.0	50.0	35.0	65.8
STT	100.0	100.0	100.0	100.0	100.0	45.0	45.0	50.0	50.0	0.0	60.0	60.0	56.0	68.0	71.8
GNS	100.0	100.0	100.0	95.0	100.0	55.0	50.0	50.0	60.0	40.0	0.0	30.0	75.0	83.0	72.2
A3N	100.0	100.0	100.0	95.0	90.0	75.0	65.0	55.0	70.0	40.0	70.0	0.0	77.0	75.0	77.8
Gesy	100.0	100.0	100.0	98.0	100.0	100.0	100.0	39.0	50.0	44.0	25.0	23.0	0.0	60.0	72.2
Gesy _s	100.0	100.0	100.0	92.0	100.0	100.0	100.0	44.0	65.0	32.0	17.0	25.0	40.0	0.0	70.4
Map 8×8															
	HR	RR	PS	SSS	LR	PGS	AHT	STT	NS	A3N	WR	GNS	Gesy	Gesy _s	Avg.
AHT	90.0	80.0	90.0	100.0	100.0	90.0	0.0	25.0	10.0	30.0	60.0	0.0	12.0	26.0	54.8
STT	90.0	90.0	90.0	90.0	90.0	90.0	75.0	0.0	60.0	20.0	0.0	20.0	43.0	7.0	58.8
NS	100.0	100.0	90.0	90.0	100.0	100.0	90.0	40.0	0.0	50.0	10.0	15.0	47.0	0.0	64.0
A3N	100.0	90.0	100.0	90.0	100.0	90.0	70.0	80.0	50.0	0.0	10.0	40.0	84.0	12.0	70.5
WR	100.0	100.0	100.0	100.0	100.0	90.0	40.0	100.0	90.0	90.0	0.0	20.0	100.0	0.0	79.2
GNS	100.0	100.0	100.0	100.0	100.0	90.0	100.0	80.0	85.0	60.0	80.0	0.0	72.0	64.0	87.0
Gesy	100.0	100.0	100.0	98.0	100.0	92.0	88.0	57.0	53.0	16.0	0.0	28.0	0.0	0.0	64.0
Gesy _s	100.0	100.0	100.0	98.0	100.0	97.0	74.0	93.0	100.0	88.0	100.0	36.0	100.0	0.0	91.2
Map 16×16															
	HR	RR	NS	STT	PS	SSS	PGS	LR	AHT	A3N	WR	GNS	Gesy	Gesy _s	Avg.
PGS	90.0	90.0	100.0	60.0	40.0	60.0	0.0	30.0	40.0	30.0	0.0	10.0	16.0	0.0	43.5
LR	50.0	100.0	100.0	80.0	90.0	80.0	70.0	0.0	0.0	30.0	0.0	0.0	10.0	0.0	46.9
AHT	100.0	100.0	0.0	50.0	100.0	70.0	60.0	100.0	0.0	10.0	0.0	0.0	73.0	2.0	51.2
A3N	100.0	70.0	100.0	90.0	50.0	60.0	70.0	70.0	90.0	0.0	50.0	55.0	86.0	52.0	72.5
WR	100.0	100.0	80.0	100.0	100.0	100.0	100.0	100.0	100.0	50.0	0.0	20.0	100.0	20.0	82.3
GNS	100.0	100.0	90.0	80.0	100.0	90.0	90.0	100.0	100.0	45.0	80.0	0.0	100.0	68.0	87.9
Gesy	90.0	90.0	1.0	35.0	84.0	90.0	84.0	90.0	27.0	14.0	0.0	0.0	0.0	0.0	46.5
Gesy _s	100.0	100.0	87.0	99.0	100.0	100.0	100.0	100.0	98.0	48.0	80.0	32.0	100.0	0.0	88.0
Map 9×8															
	AHT	WR	PGS	SSS	HR	STT	LR	NS	RR	PS	GNS	A3N	Gesy	Gesy _s	Avg.
LR	75.0	75.0	55.0	65.0	50.0	60.0	0.0	60.0	25.0	0.0	25.0	0.0	82.0	0.0	44.0
NS	100.0	100.0	70.0	70.0	75.0	40.0	40.0	0.0	10.0	0.0	5.0	0.0	89.0	5.0	46.5
RR	75.0	75.0	85.0	80.0	75.0	85.0	75.0	90.0	0.0	75.0	10.0	25.0	81.0	0.0	63.9
PS	100.0	100.0	65.0	75.0	95.0	80.0	100.0	100.0	25.0	0.0	10.0	5.0	100.0	27.0	67.8
GNS	65.0	65.0	90.0	90.0	80.0	90.0	75.0	95.0	90.0	90.0	0.0	60.0	89.0	73.0	80.9
A3N	95.0	100.0	95.0	95.0	100.0	95.0	100.0	100.0	75.0	95.0	40.0	0.0	99.0	38.0	86.7
Gesy	49.0	53.0	38.0	32.0	43.0	21.0	18.0	11.0	19.0	0.0	11.0	1.0	0.0	5.0	23.2
Gesy _s	100.0	100.0	90.0	91.0	100.0	96.0	100.0	95.0	100.0	73.0	27.0	62.0	95.0	0.0	86.8

Table 2 – Winning rate of the row player against the column player. The last column (‘Avg.’) shows the average winning rate of each row method considering its matches against all methods we included in our experiments.

3.6.3 Results: Strength of Play

The first results are in Table 2, where each number shows the average winning rate of the row-table method against the column-table method. The winning rate is computed by summing the total number of victories and half of the number of draws of the row method against a column method, and then dividing this sum by the total number of matches played; the result of the division is then multiplied by 100. As already mentioned, the results come from a total of ten matches for each pair of methods in Table 2. The winning rate for Gesy and Gesy_s computes the overall performance of the ten winner scripts against each opponent. For example, Gesy_s wins on average 88% of its matches against GNS on the 8 × 8 map. It means that winner scripts synthesized by Gesy_s are able to keep a high win score against GNS. The last column (‘Avg.’) shows the average winning rate of row methods, considering matches against all methods in our experiments. In addition to the Gesy, we only present the best six opponents to become the table better legible.

Gesy is competitive with several other search-based methods in the 8 × 8 and 16 × 16 maps. Gesy outperforms six search methods (PS, SSS, PGS, AHT, STT, and NS) and three expert-written scripts (LR, HR, and RR) on the 8 × 8 map, and five search methods (NS, PS, SSS, STT, and PGS) and three expert-written scripts (LR, HR, and RR) on the 16 × 16 map. In both cases, the only expert-written script with better performance than Gesy is WR. WR usually achieves high scores in the μ RTS competitions on these maps (ONTAÑÓN *et al.*, 2018). As we expected, Gesy’s results on the 9 × 8 map suggest that this is the most complex map for our approach when the initial population is not initialized with existing scripts.

Gesy_s outperforms the majority of other methods in terms of average winning rate for all maps, except by 18 × 8 map where A3N is the best method, followed by Gesy, GNS, and STT. Gesy_s reaches the same average winning rate as A3N for 9 × 8 map. Search-based methods can be very effective in 18 × 8 map because they can micromanage their units well. The 8 × 8 map is another example where search-based methods should be effective since they can provide fine control to units during combat; however, Gesy_s outperforms all the search-based methods and the expert-written scripts in this map.

The 18 × 8 map is the only map where Gesy_s does not outperform Gesy for average winning rate or direct matches. The addition of the four expert-written scripts in the initial population of Gesy_s makes no difference in this map. The strategies encoded in the four added scripts are mainly designed for economy-related matches, where the player has to collect resources and train units. On the other hand, the 18 × 8 map does not require to evolve an economy and train units. In the other maps, where Gesy_s performs better than Gesy, we can see at least one of the expert-written scripts among the top six methods.

Finally, Gesy_s outperforms all expert-written scripts for all maps tested by a large margin. For example, RR is the best performing expert-written script on map 9 × 8, and it obtained a

63.9 average winning rate, while the script synthesized by Gesy_s obtained an 86.8% of average winning rate and won all matches against RR. These results demonstrate that Gesy improves the strength of existing scripts while tailoring them to specific maps.

Average Response Time (ms)													
PGS	SSS	PS	STT	AHN	NS	A3N	GNS	WR	LR	HR	RR	Gesy	Gesy _s
102.3	106.9	100.5	113.0	90.1	105.0	110.0	107.9	1.7	1.4	2.3	1.3	1.2	1.3

Table 3 – Response time of different approaches.

3.6.4 Results: Computational Time

In Table 3, we show the average response times during gameplay of each method. The response time, given in milliseconds, reports how long a method takes on average to return some action for a state of the game. The response time is a relevant metric since the artificial intelligence in a game agent must decide a move as quickly as possible. Therefore, in Table 3 we consider the time required by the scripts synthesized by Gesy and Gesy_s to play the game. Here we do not consider the time Gesy and Gesy_s need to synthesize the scripts since it is an offline method that synthesizes the script before the game.

In this context, the scripts synthesized by Gesy and Gesy_s have the lowest average response time among all methods. In general, all the scripts return action in less than three milliseconds. Most of the search-based algorithms use all available time planning (something around 100 milliseconds). In addition to play strength, as mentioned, the response time is essential in commercial games once computational resources are shared among different tasks. Our approaches synthesize scripts that encode robust strategies in terms of play strength with a low response time. We believe these characteristics make the scripts synthesized by our methods relevant for commercial use.

3.6.5 Interpretability Analysis

As an example of synthesized scripts from our approach, we show one of the scripts synthesized by Gesy_s, for the map 8×8 .

```

1 train(Worker)
2 for(each unit u)
3   if(HaveQtdUnitsHarversting(u, 1)
4     attack(Worker, closest, u)
5   else
6     harvest(u)
7   attack(Worker, closest, u)

```

The strategy encoded by the script continuously trains worker units (line 1). Next, it iterates through all units the player controls and, if there is at least one player unit that is harvesting, all worker units u are sent to attack the closest opponent unit (lines 2–4). If no units are harvesting, u is sent to harvest the closest resource (line 6). It is possible at this point that u was not issued any action. For example, the script could assign the harvest action to unit u because no unit is doing it. However, the harvest action might not be legal for u as the resources might have been depleted, or someone destroyed the base where a collected resource has to be delivered. In this case, the script sends the idle unit to attack the closest opponent unit (line 7). This strategy is an improved version of that encoded by the WR included by us in the initial population. WR does not include the attack command that the script synthesized by Gesy_s has in line 7. Therefore, in WR, if there are not more bases or resources for harvesting, the unit stays idle. This optimization is subtle and can go unnoticed by professional programmers. Gesy_s's search procedure can detect the advantage of having an extra attack action, and it outperforms WR in terms of the average winning rate on this map as a result. It is an interesting direction for future research, where Gesy is used to help professional programmers to develop more robust scripts.

3.7 Conclusions and Future Works

In this paper, we introduced Gesy, a genetic programming approach to script synthesis for zero-sum games. Gesy approximates a solution to the script synthesis problem by evolving an initial population of scripts through genetic operators. Results on μ RTS showed that our approach synthesizes competitive scripts in terms of strength of play within a reduced response time than search-based approaches from the literature. Moreover, our approach provides scripts that can be understood and used to fix existing scripts' weaknesses. In particular, Gesy_s took available scripts from the literature and generated new ones that can be seen as improvements over the literature approaches. The script generation of our approach can be computationally expensive, especially for larger maps where the matches performed to evaluate scripts can take several minutes. The synthesis in our method is an evolutionary and offline process that aims to generate a robust script. As future work, we will investigate ways to mitigate the computational cost required for evaluating the scripts. Another issue for future work is allowing and penalizing infeasible scripts, which can help us evaluate better how short the current search over the feasible space is.

3.8 Appendix

3.8.1 DSL used in the experiments

The DSL we used in our experiments is defined by the following context-free grammar.

$$\begin{aligned}
 S_1 &\rightarrow CS_1 \mid S_2S_1 \mid S_3S_1 \mid \varepsilon \\
 S_2 &\rightarrow \text{if}(B) \text{ then } \{C\} \mid \text{if}(B) \text{ then } \{C\} \text{ else } \{C\} \\
 S_3 &\rightarrow \text{for (each unit } u) \{S_4\} \\
 S_4 &\rightarrow CS_4 \mid S_2S_4 \mid \varepsilon \\
 B &\rightarrow b_1 \mid b_2 \mid \dots \mid b_m \\
 C &\rightarrow c_1C \mid c_2C \mid \dots \mid c_nC \mid c_1 \mid c_2 \mid \dots \mid c_n \mid \varepsilon
 \end{aligned}$$

S_1 is the initial symbol and ε is an empty string. The language is simplified as it does not allow nested conditionals and nested loops. The DSL allows programs with if-clauses inside and outside for-loops and infinitely large sequences of DSFs. Next, we describe the DSFs and DSBs we include in the DSL.

- `build(<struct_type>, <quant>, <direction>)`: Sends free units (units that have not received actions) to build quant structures of type `struct_type` in the cell located on the direction next to the unit.
- `train(<unit_type>, <quant>, <direction>)`: Assigns a free structure to produce quant units of type `unit_type`. The newly produced units appear on the direction of the structure.
- `harvest(<quant>)`: Sends quant free units to collect resources.
- `attack(<unit_type>, <unit_target>)`: Sends free units of type `unit_type` to attack to the enemy unit that satisfies a specific condition given by `unit_target`. *closest_enemy* is an example of an enemy condition, which indicates that the unit to be attacked is the closest opponent's unit.
- `moveaway(<unit_type>)`: Sends free units of type `unit_type` to move in the direction of the player's base.
- `moveToUnit(<unit_type>, <player_unit> <unit_target>)`: Sends free units of type `unit_type` to move in the direction of the `unit_target` that belongs to the player `player_unit`.

Next, we describe the DSBs we include in our DSL.

- `QuantityUnits(<unit_type>, <quant>)`: Returns true if the player has at least `quant` units of type `unit_type` on the map.
- `QuantityEnemy(<unit_type>, <quant>)`: Returns true if the opponent player has at least `quant` units of type `unit_type` on the map.
- `QuantityHarvesting(<quant>)`: Returns true if the player has at least `quant` units gathering resources.
- `QuantityAttacking(<unit_type>, <quant>)`: Returns true if the player has at least `quant` units of type `unit_type` attacking opponent's units.
- `AllyStrongest(<unit_type>)`: Checks if at least one player's unit of type `unit_type` can destroy at least one opponent's unit with just one attack.
- `EnemyStrongest(<unit_type>)`: Checks if there is at least one opponent's unit that can destroy at least one player's unit of type `unit_type` with just one attack.
- `AllyInEnemyRange(<unit_type>)`: Returns true if a player's unit of type `unit_type` is inside the range of attack of an opponent's unit.
- `EnemyInAllyRange(<unit_type>)`: Returns true if a opponent's unit is inside the range of attack of a player's unit of type `unit_type`.
- `DistantEnemy(<unit_type>, <distance>)`: Returns true if a player's unit of type `unit_type` is at a distance of `distance` or less from an enemy's unit.

We now describe the domains of the functions in the DSL.

- `<unit_type>`: Defines one of the possible type of units available in μ RTS that can attack, that is *Worker*, *Light*, *Heavy* and *Ranged*. We also include the value *All*, which points to those four types of units.
- `<struct_type>`: Defines one of the possible type of units available in μ RTS that can produce other units, that is *Base* and *Barrack*.
- `<unit_target>`: Defines a unit used as reference to perform an action. For example the value *closest* for `unit_target` in the DSF attack indicates that the unit that u has to attack is the oponent's closest unit. The complete list of values that `unit_target` can receive are:
 - *closest*: The closest unit from the player's unit u .
 - *farthest*: The farthest unit from the player's unit u .

- *strongest*: The unit that can produce the greatest damage to another unit with only an attack.
 - *weakest*: The unit that can produce the lowest damage to another unit with only an attack.
 - *mostHP*: The unit with the highest points health value.
 - *lessHP*: The unit with the lowest points health value.
- `<direction>`: Defines the direction of the cell on which an action will be performed. The possible values this parameter can receive are: *Right*, *Left*, *Down*, *Up*, and *EnemyDir*, which indicates the direction where is placed the opponent's base on the map.
 - `<quant>`: An integer number that defines quantity.
 - `<player_unit>`: Defines the player to whom a specific unit belongs. *Ally* defines that the referred unit belongs to the player playing with the script, and *Enem* defines that the referred unit belongs to the opponent.

In addition to the DSBs and DSFs, μ Language includes a loop statement for (each unit u) that iterates over all units u of the player controlled by the script. A for-loop can be used with DSBs and DSFs that require a unit u as input. For example, if the DSF `attack(u, <unit_type>, <unit_target>)` is inside a for-loop, the script sends only the unit u to attack the enemy unit that satisfies a specific condition given by the value of `unit_target`. If the DSB `AllyInEnemyRange(u, <unit_type>)` is inside a for-loop, it returns true only if unit u , which has to be of type `unit_type`, is inside the range of attack of an opponent unit. These are the DSFs that can receive a unit u as input.

- `attack`
- `build`
- `harvest`
- `moveAway`
- `moveToCoordinates`
- `moveToCoordinatesOnce`
- `moveToUnit`

These are the DSBs that can receive a unit u as input.

- `AllyInEnemyRange`

-
- EnemyInAllyRange
 - EnemyStrongest
 - AllyStrongest
 - DistantEnemy

PROGRAMMATIC STRATEGIES FOR REAL-TIME STRATEGY-GAMES

Authors:

Julian R. H. Mariño (julianmarino@usp.br)

Rubens O. Moraes (rubens.moraes@ufv.br)

Tassiana Oliveira (tassiana.rios@ufv.br)

Claudio Toledo (claudio@icmc.usp.br)

Levi H. S. Lelis (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 work¹ 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 μ RTS, 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.

¹ This work was published in the *AAAI Conference on Artificial Intelligence (AAAI-21)*

4.1 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.*, 2019b). 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 μ RTS, 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 μ RTS 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 themselves.

4.2 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; 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; KLEIN, 2010; MENON *et al.*, 2013; MURALI; CHAUDHURI; 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; SOLAR-LEZAMA, 2018; VERMA *et al.*, 2018; VERMA *et al.*, 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; 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; SIPPER, 2011b) and as pruning policies (BENBASSAT; 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.* canaan2018evolving 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. FREITAS; SOUZA; BERNARDINO de2018evolving use a genetic-programming approach for evolving controllers for a Mario AI simulator. BUTLER; TORLAK; POPOVIĆ butler2017synthesizing present a method for synthesizing strategies for the puzzle game nonograms. FREITAS; SOUZA; BERNARDINO, and BUTLER; TORLAK; POPOVIĆ's approaches are different than our method because they deal with single-agent problems, while we deal with two-players zero-sum games.

4.3 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 (DEURSEN; KLINT; VISSER, 2000). Let $\llbracket D \rrbracket$ be the set of scripts that can be synthesized with a DSL D . We are interested in synthesizing a script $p_i \in \llbracket D \rrbracket$ for player i that maximizes the value of the game while player $-i$ plays the game with a script p_{-i} from $\llbracket D \rrbracket$ that minimizes the value of the game. We formulate the script synthesis problem as,

$$\max_{p_i \in \llbracket D \rrbracket} \min_{p_{-i} \in \llbracket D \rrbracket} \mathcal{V}(s_{init}, p_i, p_{-i}). \quad (4.1)$$

The strategies encoded by scripts p_i and p_{-i} that solve the Equation 4.1 define a Nash equilibrium profile if one considers only strategies encoded by scripts in $\llbracket D \rrbracket$ 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 $\llbracket D \rrbracket$. LS2 uses a self play procedure with local search for approximating a solution to Equation 4.1.

Figure 8 – Derivation tree for “c1 if (b1) then c3”.

4.4 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.

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

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 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 8 shows an example of a derivation tree.

4.5 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 4.1.

Lasi receives a game $\mathcal{G}$, a grammar $G = (V, \Sigma, R, S)$ defining a DSL, and a sequence of state-action pairs, $T = \{(s_{init}, a_1), (s_2, a_2), \dots, (s_n, a_n)\}$, 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.

4.5.1 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 3 (DSL Simplification Problem). Let $\llbracket G \rrbracket$ 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 $p \in \llbracket G' \rrbracket$ 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 $p \in \llbracket G \rrbracket$ with $p(s_m) = a_m$ for all $(s_m, a_m) \in T$, otherwise the simplification task does not have a solution.

4.5.2 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 .

4.5.2.1 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; 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 $|\{(s, a) | (s, a) \in Q \wedge o(s) = a\}|$. We remove from Q the state-action pairs in $\{(s, a) | (s, a) \in Q \wedge o(s) = a\}$ 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).

4.5.2.2 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 $\{b | b \in \Sigma \wedge b(s) = \text{false} \forall s \in T\}$ nor the DSBs in $\{b | b \in \Sigma \wedge b(s) = \text{true} \forall s \in T\}$. 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}`, `while(not b){c1}`, and `var ← b` can be replaced by `ε`, `c1`, `ε`, `while(True){c1}`, and `var ← false`, respectively, where ε is an empty string. Similarly, all programs 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 .

4.6 Synthesizing Scripts with Self Play

Once Lasi simplifies the DSL, LS2 uses the self play procedure described in Algorithm 4 to synthesize a script for game $\mathcal{G}$. Algorithm 4 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

Algorithm 4 – 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:  $p \leftarrow \text{random-script}(D)$ 
2: for  $k = 1$  to  $n$  do
3:    $p_{BR} \leftarrow \text{local-search}(\mathcal{G}, D, p)$ 
4:   if  $p_{BR}$  defeats  $p$  in  $\mathcal{G}$  then
5:      $p \leftarrow p_{BR}$ 
6:   end if
7: end for
8: return  $p$ 

```

uniformly at random to be applied to non-terminal symbols; a script is returned once it contains no non-terminal symbols.

Algorithm 4 iteratively improves the initial random solution p with a local search algorithm. The local search receives the game $\mathcal{G}$, the DSL D , and current script p , and returns 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 μ RTS, 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 4 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 assumes 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 symbol 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 8 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.

4.7 Empirical Methodology

We evaluate LS2's scripts in μ RTS (ONTAÑÓN, 2017) by comparing 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 interpretability of the synthesized scripts. Our implementation of LS2 is available online.²

4.7.1 Problem Domain: μ RTS

We chose to use μ RTS in our experiments because it has an active research community with competitions being organized (ONTAÑÓN *et al.*, 2018), with all competing algorithms available in a single codebase.³ Moreover, in 2017 a script won two tracks of the μ RTS competition, cf. Table 1 of (ONTAÑÓN *et al.*, 2018), demonstrating that scripts can outperform other approaches for this type of game. Finally, μ RTS can offer a diverse set of challenges because the agents can be easily evaluated on different maps of the game.

Most μ RTS 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 battle the opponent. In some of the maps, players also start with a structure called base, which is used to train workers 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 action, including bases and barracks.

We used four μ RTS 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 opponent 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 workers, one base placed on each side of the map. The maps 8×8 , 9×8 , and 24×24 were used in μ RTS competitions.

4.7.2 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.

² <<https://github.com/julianmarino/LS2>>

³ <<https://github.com/santiontanon/microrts>>

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; 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 .

4.7.3 Competing Agents

We use the following search algorithms: Portfolio Greedy Search (PGS) (CHURCHILL; BURO, 2013), Stratified Strategy Selection (SSS) (LELIS, 2017), the MCTS version of Puppet Search (PS) (BARRIGA; STANESCU; BURO, 2017b), Strategy Tactics (STT) (BARRIGA; STANESCU; BURO, 2017a), NaïveMCTS (NS) (ONTAÑÓN, 2017), A3N (MORAES *et al.*, 2018), and Naive Guided Sample (GNS) (YANG; 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.

4.7.4 μ Language: a Domain-Specific Language for μ RTS

We have developed a DSL for μ RTS, which we name μ Language, to allow for the development of scripts in μ RTS. Both users in our study and synthesizers use the same DSL.

The CFG below summarizes the μ Language.

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

S_1 is the initial symbol, ε is an empty string, and symbols $c_1, c_2, \dots, c_n$ are DSFs and $b_1, b_2, \dots, b_n$ are DSBs. μ Language 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.

4.7.5 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 μ RTS and showing a few examples of μ Language, and also had access to a manual describing each DSF and DSB implemented in the μ Language. 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.

4.8 Empirical Results

4.8.1 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 4, 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

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 4 – 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.

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 μ RTS. The time required to generate one trace

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 5 – Average winning rate across all four maps tested.

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 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 4, 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 P₃ 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 P₂. In the 18×8 map, LS2 was also outperformed by P₃ 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. moraes2018abstractions](#), which was used to generate the results of Table 4).

Table 5 shows the average winning rate of the methods evaluated across all four maps. The scripts synthesized by LS2 are outperformed only by programmer P₂, 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.

```

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 6 – Scripts synthesized by LS2 and the best two scripts written by the programmers for maps 18×8 and 24×24 .

4.8.2 Interpretability of Scripts

Table 6 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 μ RTS competitions. The script LS2 synthesized for map of size 8×8 (LS2-8x8) iterates through all units the player controls 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 4). We discovered that this extra instruction optimizes the pathfinding system of μ RTS. 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.

4.9 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 μ RTS 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 themselves.

4.10 Acknowledgements

This research was partially supported by CNPq, CAPES, and Canada's CIFAR AI Chairs program. The research was carried out using the computational resources of the Center for Mathematical Sciences Applied to Industry (CeMEAI) funded by FAPESP (grant 2013/07375-0), the cluster Jupiter from Universidade Federal de Viçosa, and the clusters from the Compute Canada. We thank the anonymous reviewers and David da Silva Aleixo for great suggestions on an earlier version of this paper.

CONCLUSION

In this thesis, we introduced a system that can generate action abstractions, from which search-based methods can plan on zero-sum real-time games. While previous works use action abstractions induced by a unique set of handcrafted scripts, our approach generates many novel scripts. To choose the best subset of scripts from the entire set of novel scripts, we cast the problem as selecting a subset of pure strategies from a large pool of options in a two-player game named subset selection game (SSG). The goal is to choose the subset of scripts that leads to the best action abstraction. Results on μ RTS show that search-based algorithms, using the action abstractions generated by our system, outperformed the same search-based algorithms with existing action abstractions found in the literature. Another test performed in a small μ RTS map showed that our evolutionary approach could converge to a Nash equilibrium profile for the SSG.

Before this thesis, search-based approaches had shown to be stronger than strategies encoded by scripts for planning in zero-sum real-time domains. In this research, we presented Gesy, a Genetic Programming approach that solves the problem of generating effective scripts for playing zero-sum real-time games. We showed that Gesy synthesizes competitive scripts against the state-of-the-art search-based approaches and existing scripts designed by professionals. Also, the synthesized scripts present advantages over search-based algorithms. The scripts need only a small fraction of the time than a search-based approach to play the game. Also, the scripts are more interpretable than search-based approaches. We showed that the synthesized scripts could point to subtle optimizations that a professional programmer could not easily notice.

Gesy needs the inclusion of a set of handcrafted scripts in the initial population of the evolutionary process to generate robust scripts. This constraint can be problematic in domains where there are not a readily available set of handcrafted scripts. In this thesis we introduced LS2, a script synthesizer that doesn't need an initial set of effective predefined scripts to discover effective programs. LS2 combines a novel algorithm for simplifying DSLs, we called Lasi, and a local search algorithm for synthesizing scripts with self-play. The only input of Lasi is a map

and a trace of a game played by a demonstrator for the same map. Then Lasi uses that trace to define a simplified DSL that allows the derivation of a script, reproducing the trace generated by the search-based algorithm. We show that the scripts synthesized by LS2 are interpretable and are competitive with the search-based state-of-the-art methods. We also performed a user study that showed that the scripts synthesized by LS2 are competitive against scripts developed explicitly for the same maps used to test LS2.

The solutions presented in this thesis have potential applications in other real-time domains than zero-sum real-time games. For example, agents usually have to deal with a large action space in robotics rescue tasks and have a very limited available time to return responses. Therefore, we hypothesize that action abstractions defined by the system presented in this thesis could help search-based algorithms in the process of making the most effective decisions in such domains. In domains such as self-driving cars, it's desirable to validate the transparency and reliability of the agent's behavior before using them in production. Scripts synthesized by our systems could provide interpretable strategies that can be part of entire self-driving cars systems. In medical field, Lasi could be used to reproduce the behavior of a search-based algorithm or a machine learning approach in an explicit interpretable computer program. Moreover, medical professionals could understand how and why a machine learning or search-based system gives a diagnostic to a patient.

There are several future research directions from the results of this thesis. The empirical results of LS2 show that scripts synthesized with a simplified language outperform some robust search-based methods. These scripts have low response time, and the encoded strategies can be interpretable. However, LS2 depends on a practical demonstrator to generate the trace used to delineate the space of scripts that can be synthesized. We intend to investigate how to mitigate the LS2 dependency from a demonstrator. The proposed action abstractions generator, local search algorithm used by LS2 and the evolutionary approach Gesy have the computational time to synthesize robust strategies as a drawback. The evaluation of strategy relies on game-matches which can be time-consuming for some domains. This limitation opens the opportunity to investigate faster evaluation methods, which could estimate the strategies' performance without playing full game matches.

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. Citation on page 50.

AHMAD, M. A.; ECKERT, C.; TEREDESAL, A. Interpretable machine learning in healthcare. In: **Proceedings of the 2018 ACM international conference on bioinformatics, computational biology, and health informatics**. [S.l.: s.n.], 2018. p. 559–560. Citation on page 27.

ALUR, R.; BODIK, R.; JUNIWAL, G.; MARTIN, M. M. K.; RAGHOTHAMAN, M.; SESHIA, S. A.; SINGH, R.; SOLAR-LEZAMA, A.; TORLAK, E.; UDUPA, A. 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. Citations on pages 52 and 73.

ARULKUMARAN, K.; CULLY, A.; TOGELIUS, J. Alphastar: An evolutionary computation perspective. In: **Proceedings of the Genetic and Evolutionary Computation Conference Companion**. [S.l.: s.n.], 2019. p. 314–315. Citation on page 27.

BALOG, M.; GAUNT, A. L.; BROCKSCHMIDT, M.; NOWOZIN, S.; TARLOW, D. Deep-coder: Learning to write programs. In: **Proceedings International Conference on Learning Representations**. [S.l.]: OpenReviews.net, 2017. p. 1–21. Citations on pages 52 and 73.

BARRIGA, N. A.; STANESCU, M.; BURO, M. Combining strategic learning and tactical search in real-time strategy games. **Thirteenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**, p. 9–15, 2017. Citations on pages 35, 43, 50, 60, and 80.

_____. Game tree search based on nondeterministic action scripts in real-time strategy games. **IEEE Transactions on Games**, IEEE, v. 10, n. 1, p. 69–77, 2017. Citations on pages 35, 43, 50, 60, and 80.

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. Citation on page 73.

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. Citation on page 51.

_____. Evolving board-game players with genetic programming. In: **Genetic and Evolutionary Computation Conference**. [S.l.: s.n.], 2011. p. 739–742. Citation on page 74.

_____. Evolving both search and strategy for reversi players using genetic programming. In: **IEEE Conference on Computational Intelligence and Games**. [S.l.: s.n.], 2012. p. 47–54. Citation on page 74.

BOSC, G.; TAN, P.; BOULICAUT, J.-F.; RAISSI, C.; KAYTOUE, M. A pattern mining approach to study strategy balance in rts games. **IEEE Transactions on Computational Intelligence and AI in games**, IEEE, v. 9, n. 2, p. 123–132, 2015. Citation on page 50.

BROWN, N.; SANDHOLM, T. Regret transfer and parameter optimization. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2014. p. 594–601. Citation on page 35.

BURO, M. Real-time strategy games: A new ai research challenge. In: **IJCAI**. [S.l.: s.n.], 2003. v. 2003, p. 1534–1535. Citation on page 96.

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. Citations on pages 52 and 74.

CANAAN, R.; SHEN, H.; TORRADO, R.; TOGELIUS, J.; NEALEN, A.; MENZEL, S. Evolving agents for the hanabi 2018 cig competition. In: IEEE. **2018 IEEE Conference on Computational Intelligence and Games**. [S.l.], 2018. p. 1–8. Citations on pages 35, 51, 55, and 74.

CAO, V. L.; NICOLAU, M.; MCDERMOTT, J. One-class classification for anomaly detection with kernel density estimation and genetic programming. In: SPRINGER. **European Conference on Genetic Programming**. [S.l.], 2016. p. 3–18. Citation on page 52.

CHAKRABORTI, T.; KULKARNI, A.; SREEDHARAN, S.; SMITH, D. E.; KAMBHAMPATI, S. Explicability? Legibility? Predictability? Transparency? Privacy? Security? the emerging landscape of interpretable agent behavior. In: **Proceedings of the International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2019. v. 29, n. 1, p. 86–96. Citation on page 50.

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. Citations on pages 28, 30, 34, 35, 36, 37, 42, 50, 60, and 80.

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. Citations on pages 51 and 55.

DEURSEN, A. V.; KLINT, P.; VISSER, J. Domain-specific languages: An annotated bibliography. **ACM Sigplan Notices**, ACM New York, NY, USA, v. 35, n. 6, p. 26–36, 2000. Citations on pages 50, 74, and 98.

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. Citation on page 27.

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. Citations on pages 52 and 74.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability: A Guide to the Theory of NP-Completeness**. [S.l.]: W. H. Freeman, 1979. Citation on page 77.

GINTIS, H. **Game Theory Evolving: A Problem-centered Introduction to Modeling Strategic Behavior**. [S.l.]: Princeton University Press, 2000. (Economics / Princeton University Press). ISBN 9780691009438. Citation on page 36.

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. Citation on page 52.

_____. Programming by examples: Applications, algorithms, and ambiguity resolution. In: SPRINGER. **International Joint Conference on Automated Reasoning**. [S.l.], 2016. p. 9–14. Citation on page 52.

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. Citations on pages 28 and 73.

HAWKIN, J. A.; HOLTE, R.; SZAFRON, D. Automated action abstraction of imperfect information extensive-form games. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2011. p. 681–687. Citation on page 35.

_____. Using sliding windows to generate action abstractions in extensive-form games. In: **AAAI**. [S.l.: s.n.], 2012. Citation on page 35.

HUDAK, P. Domain-specific languages. **Handbook of programming languages**, v. 3, n. 39-60, p. 21, 1997. Citation on page 98.

JACOB, M.; DEVLIN, S.; HOFMANN, K. “it’s unwieldy and it takes a lot of time” — challenges and opportunities for creating agents in commercial games. **Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**, v. 16, n. 1, p. 88–94, Oct. 2020. Available: <<https://ojs.aaai.org/index.php/AIIDE/article/view/7415>>. Citation on page 50.

JHA, S.; GULWANI, S.; SESHIA, S. A.; TIWARI, A. 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. Citations on pages 52 and 73.

JOBSTMANN, B.; GRIESMAYER, A.; BLOEM, R. Program repair as a game. In: SPRINGER. **International conference on computer aided verification**. [S.l.], 2005. p. 226–238. Citation on page 52.

JUSTESEN, N.; TILLMAN, B.; TOGELIUS, J.; RISI, S. Script- and cluster-based UCT for StarCraft. In: **IEEE Conference on Computational Intelligence and Games**. [S.l.: s.n.], 2014. p. 1–8. Citation on page 34.

KOLLER, D.; MEGIDDO, N. The complexity of two-person zero-sum games in extensive form. **Games and economic behavior**, Elsevier, v. 4, n. 4, p. 528–552, 1992. Citation on page 95.

KOZA, J. R. **Genetic Programming: On the Programming of Computers by Means of Natural Selection**. Cambridge, MA: MIT Press, 1992. Citations on pages 52, 57, and 73.

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. Citations on pages 34, 35, 42, 50, 60, and 80.

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. Citations on pages 52 and 73.

MARIÑO, J. R.; MORAES, R. O.; OLIVEIRA, T. C.; TOLEDO, C.; LELIS, L. H. 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. Citations on pages 31 and 51.

MARINO, J. R.; MORAES, R. O.; TOLEDO, C.; LELIS, L. H. Evolving action abstractions for real-time planning in extensive-form games. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2019. v. 33, n. 01, p. 2330–2337. Citation on page [52](#).

MARIÑO, J. R. H.; MORAES, R. O.; TOLEDO, C.; LELIS, L. H. S. Evolving action abstractions for real-time planning in extensive-form games. In: **The AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2019. p. 2330–2337. Citation on page [30](#).

Maynard Smith, J.; PRICE, G. R. The logic of animal conflict. **Nature**, Nature Publishing Group SN, v. 246, p. 15–18, 11 1973. Citations on pages [39](#) and [40](#).

MENON, A.; TAMUZ, O.; GULWANI, S.; LAMPSON, B.; KALAI, A. A machine learning framework for programming by example. In: **Proceedings of the International Conference on Machine Learning**. [S.l.]: PMLR, 2013. p. 187–195. Citations on pages [52](#) and [73](#).

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. Citations on pages [34](#), [35](#), and [37](#).

MORAES, R. O.; Mariño, J. R. H.; LELIS, L. H. S.; NASCIMENTO, M. A. Action abstractions for combinatorial multi-armed bandit tree search. In: AAAI. **Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**. [S.l.], 2018. p. 74–80. Citations on pages [30](#), [42](#), [50](#), [60](#), [79](#), [80](#), and [83](#).

MURALI, V.; CHAUDHURI, S.; JERMAINE, C. Bayesian sketch learning for program synthesis. In: **Proceedings International Conference on Learning Representations**. [S.l.]: OpenReviews.net, 2017. Citation on page [73](#).

NEUMANN, J. v. Zur theorie der gesellschaftsspiele. **Mathematische Annalen**, v. 100, n. 1, p. 295–320, 1928. Citation on page [38](#).

NGUYEN, S.; ZHANG, M.; JOHNSTON, M.; TAN, K. C. Genetic programming for job shop scheduling. In: **Evolutionary and Swarm Intelligence Algorithms**. [S.l.]: Springer, 2019. p. 143–167. Citation on page [52](#).

ONTAÑÓN, S. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In: **Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**. [S.l.: s.n.], 2013. p. 58–64. Citations on pages [41](#), [51](#), [52](#), [53](#), and [96](#).

ONTAÑÓN, S. Combinatorial multi-armed bandits for real-time strategy games. **Journal of Artificial Intelligence Research**, v. 58, p. 665–702, 2017. Citations on pages [43](#), [50](#), [60](#), [78](#), and [80](#).

ONTAÑÓN, S.; BARRIGA, N. A.; SILVA, C. R.; MORAES, R. O.; LELIS, L. H. The first micrororts artificial intelligence competition. **AI Magazine**, v. 39, n. 1, p. 75–83, 2018. Citations on pages [28](#), [41](#), [43](#), [47](#), [63](#), [79](#), and [97](#).

ONTAÑÓN, S.; BURO, M. Adversarial hierarchical-task network planning for complex real-time games. In: **Proceedings of the International Joint Conference on Artificial Intelligence**. [S.l.: s.n.], 2015. p. 1652–1658. Citations on pages [43](#), [50](#), and [60](#).

PERELMAN, D.; GULWANI, S.; BALL, T.; GROSSMAN, D. 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. Citation on page [52](#).

PONSEN, M.; MUNOZ-AVILA, H.; SPRONCK, P.; AHA, D. W. Automatically generating game tactics through evolutionary learning. **AI Magazine**, v. 27, n. 3, p. 75–84, 2006. Citation on page [35](#).

PONSEN, M.; SPRONCK, P. **Improving adaptive game AI with evolutionary learning**. Phd Thesis (PhD Thesis) — Citeseer, 2004. Citations on pages [51](#) and [55](#).

ROSS, S. M. Goofspiel – the game of pure strategy. **Journal of Applied Probability**, Cambridge University Press, v. 8, n. 3, p. 621–625, 1971. Citation on page [36](#).

SANDHOLM, T. Abstraction for solving large incomplete-information games. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2015. p. 4127–4131. Citation on page [35](#).

SIAU, K.; WANG, W. Building trust in artificial intelligence, machine learning, and robotics. **Cutter Business Technology Journal**, v. 31, n. 2, p. 47–53, 2018. Citation on page [27](#).

SILVA, C. R.; MORAES, R. O.; LELIS, L. H. S.; GAL, Y. Strategy generation for multi-unit real-time games via voting. **IEEE Transactions on Games**, 2018. Citations on pages [35](#), [42](#), [43](#), and [47](#).

SILVER, D.; HUBERT, T.; SCHRITTWIESER, J.; ANTONOGLOU, I.; LAI, M.; GUEZ, A.; LANCTOT, M.; SIFRE, L.; KUMARAN, D.; GRAEPEL, T.; LILICRAP, T.; SIMONYAN, K.; HASSABIS, D. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. **Science**, American Association for the Advancement of Science, v. 362, n. 6419, p. 1140–1144, 2018. ISSN 0036-8075. Citation on page [72](#).

SLAVÍK, P. A tight analysis of the greedy algorithm for set cover. In: **Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing**. [S.l.: s.n.], 1996. p. 435–441. Citation on page [77](#).

SOLAR-LEZAMA, A. The sketching approach to program synthesis. In: SPRINGER. **Asian Symposium on Programming Languages and Systems**. [S.l.], 2009. p. 4–13. Citations on pages [51](#) and [55](#).

SPRONCK, P.; PONSEN, M.; SPRINKHUIZEN-KUYPER, I.; POSTMA, E. Adaptive game ai with dynamic scripting. **Machine Learning**, Springer, v. 63, n. 3, p. 217–248, 2006. Citation on page [35](#).

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. Citations on pages [51](#), [55](#), and [73](#).

STANESCU, M.; BARRIGA, N. A.; HESS, A.; BURO, M. Evaluating real-time strategy game states using convolutional neural networks. In: IEEE. **Computational Intelligence and Games, 2016 IEEE Conference on**. [S.l.], 2016. p. 1–7. Citations on pages [43](#) and [80](#).

SUGANUMA, M.; SHIRAKAWA, S.; NAGAO, T. A genetic programming approach to designing convolutional neural network architectures. In: **Proceedings of the genetic and evolutionary computation conference**. [S.l.: s.n.], 2017. p. 497–504. Citation on page [52](#).

SUN, L.; JIAO, P.; XU, K.; YIN, Q.; ZHA, Y. Modified adversarial hierarchical task network planning in real-time strategy games. **Applied Sciences**, Multidisciplinary Digital Publishing Institute, v. 7, n. 9, p. 872, 2017. Citations on pages 52 and 53.

TALBI, E.-G. **Metaheuristics: from design to implementation**. [S.l.]: John Wiley & Sons, 2009. Citations on pages 39 and 57.

TRAN, B.; XUE, B.; ZHANG, M. Genetic programming for feature construction and selection in classification on high-dimensional data. **Memetic Computing**, Springer, v. 8, n. 1, p. 3–15, 2016. Citation on page 52.

VERMA, A.; LE, H. M.; YUE, Y.; CHAUDHURI, S. Imitation-projected programmatic reinforcement learning. **arXiv preprint arXiv:1907.05431**, 2019. Citation on page 73.

VERMA, A.; MURALI, V.; SINGH, R.; KOHLI, P.; CHAUDHURI, S. Programmatically interpretable reinforcement learning. In: **Proceedings of the International Conference on Machine Learning**. [S.l.: s.n.], 2018. p. 5052–5061. Citation on page 73.

VINYALS, O.; BABUSCHKIN, I.; CZARNECKI, W. M.; MATHIEU, M.; DUDZIK, A.; CHUNG, J.; CHOI, D. H.; POWELL, R.; EWALDS, T.; GEORGIEV, P. *et al.* Grandmaster level in starcraft ii using multi-agent reinforcement learning. **Nature**, Nature Publishing Group, v. 575, n. 7782, p. 350–354, 2019. Citation on page 27.

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.; LILICRAP, T.; KAVUKCUOGLU, K.; HASSABIS, D.; APPS, C.; SILVER, D. Grandmaster level in StarCraft II using multi-agent reinforcement learning. **Nature**, v. 575, n. 7782, p. 350–354, 2019. Citation on page 72.

WANG, C.; CHEN, P.; LI, Y.; HOLMGÅRD, C.; TOGELIUS, J. Portfolio online evolution in StarCraft. In: **Proceedings of the Conference on Artificial Intelligence and Interactive Digital Entertainment**. [S.l.: s.n.], 2016. p. 114–120. Citations on pages 34 and 35.

YANG, Z.; ONTANÓN, S. 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**. [S.l.: s.n.], 2019. v. 15, n. 1, p. 100–106. Citations on pages 50, 60, and 80.

YANG, Z.; ONTANON, S. An empirical survey on methods for integrating scripts into adversarial search for rts games. **IEEE Transactions on Games**, IEEE, p. 1–9, 2021. Citation on page 53.

BASIC DEFINITIONS

In this appendix, we aim to present general concepts that are used through this thesis.

A.1 Zero-Sum Games

In game theory, two-player games can be represented in the extensive form and the normal form. The extensive form representation is "more natural" than the normal form because a game is described as a tree in the extensive form. In the tree, the first node is the start point of the game, and each non-terminal node represents a decision point of the game where players take strategies that derives in game actions. Terminal nodes have associated a payoff vector, which has the payoff for each player in the game. An edge from one node to another represents the execution of a game action according to a strategy followed by players. Therefore, an entire path from the initial node to a terminal node represents an entire possible game-play performed by the two players. A game is zero-sum when the sum of the payoffs in the vector is zero: one player wins and the other loses ([KOLLER; MEGIDDO, 1992](#)).

Zero-sum games can be sequential or simultaneous. In a sequential zero-sum game, the game's information is perfect; that is, players know the actions performed by their opponent for a specific decision point. As an example, consider the sequential two-player game shown in figure [9a](#). This game is a simple game that finishes after each player executes an action for only one decision point. Each player has two strategies available (Strategy A and strategy B). First, player 1 (P1) chooses a strategy in the tree, represented by the two edges derived from the root node. When player 2 (P2) has to decide its strategy, it already knows the strategy followed by player P1. Since it is a zero-sum game, a payoff vector is associated with the terminal nodes where the sum of their components is always zero. Suppose the case that the game is simultaneous; in that case, a standard representation of the game is shown in figure [9b](#). In that figure, there is a dotted line joining the two possible decision points for player 2. The dotted line signifies that since the movements are simultaneous, P2 does not know which strategy is adopted

by P1.

Figure 9 – Extensive form representations for a two-player zero-sum game

In this thesis, we approach two-player zero-sum simultaneous games. We use a slightly different representation, where instead of using dotted lines, edges represent strategies adopted by both players at the same decision point. That is, For states s_1, s_2 , there exists an outgoing edge from s_1 to s_2 that represents the action performed by both players.

A.2 μ RTS

Real-time strategy (RTS) games are games where players control game units that can collect resources, build structures, train other units, and fight to eliminate the opponent's units. RTS games are one of the more dynamic sub-types of strategy games since players usually have to perform actions very quickly and make decisions under uncertainty (BURO, 2003).

For the experiments developed in this thesis, we use μ RTS as a testbed, a two-player RTS game developed for research purposes (ONTAÑÓN, 2013). Figure 10 shows a screenshot of a μ RTS match in a small map. In μ RTS, one player controls the units with a blue contour, and the opponent player controls the units with a red contour. In μ RTS, six main unit types can be controlled: Worker, Light, Ranged, Heavy, Base, Barracks units. Worker, Light, Ranged, and Heavy units can move. Bases and barracks cannot move but can train other units in exchange for resources (light-green squares represent resources). Namely, a Base can train workers, and a Barrack can train Light, Ranged, and Heavy units. Bases also allow store resources that workers harvest. Light, Ranged, and heavy units are units focused on combating; however, a worker can also attack opponent units if necessary.

All the units in μ RTS have a 'hit points' value associated, representing how much damage units can suffer before they can be removed from the map game. The game's main goal is to eliminate all the opponent's units. Light, Ranged, Heavy, and Worker units have different characteristics. Heavy units move more slowly than the other units; however, they are the units that inflict greater damage to the opponent units among all types of units. Worker and Ranged units have a 'hit points' value low, however, ranged units can attack from larger distances. All

Figure 10 – A screenshot of a μ RTS match.

the other units have to be in an adjacent cell where the target opponent unit is placed. Bases are the units with the highest 'hit-points' value in the game. Dark-green squares represent blocks that cannot be transversed by units.

μ RTS matches are played in real-time. In μ RTS each player is given 100 milliseconds for returning an action after planning for a specific state of the game. An action of a player is defined by a vector of actions, where each action in the vector is the action that one player’s unit will perform in a specific state of the game. Once players decide their action vectors, the game perform the action simultaneously in the current game state. Therefore, two units can eliminate themselves if they attack each other simultaneously. We chose to use μ RTS in our experiments because it has an active research community with competitions being organized (ONTAÑÓN *et al.*, 2018).

A.3 Domain-Specific Languages

The most common programming languages used nowadays are general-purpose languages, such as Java, Python, or C++. With those languages, it's possible to write programs for almost any purpose and application domain. Thus, one can say that the space of programs that can be written using one of those languages is almost incalculable.

A domain-specific language (DSL) is a language that defines a space of programs that are focused on being applied in a particular domain. Designing a DSL includes notations and abstractions that restrict the expressiveness of the generated programs; therefore, the number of

programs generated using a DSL is usually much lower than the number of programs that can be generated using general-purpose languages. However, with DSLs, it's possible to synthesize computer programs for specific domains more quickly and effectively than using general-purpose languages (HUDAK, 1997). Examples of DSLs are SQL, which is a language for manipulating databases, and HTML, which is a HyperText Markup Language mainly used for building websites (DEURSEN; KLINT; VISSER, 2000).

A DSL can be represented using context-free grammars. To illustrate an example of a small DSL, let $G = (V, \Sigma, R, S)$ be a context-free grammar where V is a set of non-terminals, Σ is a set of terminals, R is the relations of the grammar, and S is the initial symbol. Consider the following context-free grammar, which we refer to as G_1 .

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

This grammar accepts programs with either an if-then or an if-then-else structure. The Boolean formula of the if clause can be either b_1 , b_2 , or b_3 , while the commands can be either c_1 , c_2 , or c_3 . In the example, we have that $V = \{S, B, C\}$, $\Sigma = \{b_1, b_2, b_3, c_1, c_2, c_3\}$, R are the eight relations specified above (e.g., $B \rightarrow b_1$ is a relation), and S is the start symbol. The programs defined by this DSL can be generated selecting production rules starting from symbol S . We start with a string formed by the initial symbol S and then we choose one of the productions rules derived by S . for example, we can choose the first production rule derived by S , that is 'if (B) then C '. Then, we replace symbol B by one of their derived terminal symbols b_1, b_2 or b_3 . Similarly, we replace symbol C by one of their derived terminal symbols c_1, c_2 or c_3 . The process finishes when there are not non-terminal symbols in the final program. The program "if (b_1) then c_3 " is an example of a program defined by the DSL represented by G .