STRATEGIC REASONING IN COMPLEX ZERO-SUM COMPUTER GAMES

ANDERSON ROCHA TAVARES

STRATEGIC REASONING IN COMPLEX ZERO-SUM COMPUTER GAMES

Dissertation presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Luiz Chaimowicz

Belo Horizonte

August 2018

ANDERSON ROCHA TAVARES

RACIOCÍNIO ESTRATÉGICO EM JOGOS DIGITAIS COMPLEXOS DE SOMA ZERO

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Orientador: Luiz Chaimowicz

Belo Horizonte

Agosto de 2018

Rocha Tavares, Anderson

T231s Strategic Reasoning in Complex Zero-Sum Computer Games / Anderson Rocha Tavares. — Belo Horizonte, 2018
xxvii, 119 f. : il. ; 29cm

Tese (doutorado) — Universidade Federal de Minas Gerais
Orientador: Luiz Chaimowicz

1. Computação — Teses. 2. Inteligência Artificial. 3. Aprendizado por reforço. 4. Seleção de algoritmos. 5. Jogos digitais. I. Luiz Chaimowicz. II. Strategic Reasoning in Complex Zero-Sum Computer Games.

CDU 519.6*82(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Strategic Reasoning in Complex Zero-Sum Computer Games

ANDERSON ROCHA TAVARES

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

Prof. Luiz Chaimowicz - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]

Profa. Gisele Lobo Pappa
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. Pedro Olmo Stancioli Vaz de Melo
Departamento de Ciência da Computação - UFMG

[illegible]

Profa. Anna Helena Reali Costa
Departamento de Engenharia de Computação e Sistemas Digitais - USP

[illegible]

Prof. Bruno Castro da Silva
Departamento de Informática Aplicada - UFRGS

[illegible]

Prof. Levi Henrique Santana de Lelis
Departamento de Informática - UFV

Belo Horizonte, 10 de agosto de 2018.

To my family: those in the past, in the present and in the future.

Acknowledgments

The words in this dissertation are the result of a lot of teamwork expressed through my grateful hands. Every silent moment I recall events and people that had affected me, in subtle or perceivable ways. I'll do my best to acknowledge these and I offer my sincere apologies for potentially not recalling the ones that had such impact on my life up to the time of this writing. In this case, I hope to have the chance to give a proper acknowledgment in due time.

I'm a grateful admirer of the beautiful and complex simplicity of the Universal Intelligence, for it is the roof of us roof tiles, the ocean of us water drops. Its cosmic perfection inspires us through our ultimate and eternal endeavor: evolution.

I am extremely and continuously in gratitude with mom Dôra, dad Milton (in memorian), and sis Thaís. You all have wisely shown me the paths of evolution with your examples, patience, and love. Thanks for lifting me uncountable times and for being around, even at distance. My appreciation to the Rocha, Tavares and Demaman families as well, for always having friendly words and hugs to support me. I also thank my brother-in-law Sandro, for the many entertaining techie conversations.

I consider myself extremely fortunate for having as advisor Luiz Chaimowicz, a bona fide academic person: one with genuine openness regarding research initiatives and subjects, true dedication to various aspects of education and a warm cordiality towards fellow researchers and professors. Thank you for embracing this journey, for continuously providing kind and supportive feedback and, most importantly, for becoming a good friend.

Many thanks to the committee that had carefully read this work, spotting relevant issues and suggesting a great deal of improvements. In alphabetical order: Anna Reali, Bruno da Silva, Gisele Pappa, Levi Lelis, and Pedro Olmo: this version of the dissertation is much better thanks to you. I also thank Ana Bazzan for being part of the candidacy committee and for guiding my first academic steps during my master's degree. I'd like to show my appreciation and wish to be around such great researchers and nice people as you are. And, of course, the eventual remaining mistakes on this text are exclusively my doing.

I had the honor to work directly with many brilliant people. My thanks to the many members of the Y-Rescue team, for all the effort during our RoboCup journey, and to Héctor, Amanda, Daniel, Tiago, Leandro, Siva and Gianlucca for the great interactions during our works in co-authorship. I also thank Gabriel Ramos for the friendly advice and nice reminders on being a connected researcher, as well as of important academic deadlines.

VeRLab/J is a hub of great minds, and I'm honored to be part of this team. Some of you have directly aided in this dissertation, be it with the computational resources, or in the great moments at the bandeijão, football or peteca. In alphabetical order: Balbino, Bernardo, Bruna, Colares, David, Drews, Elerson, Michel, Omar, and Washington: thank you (and I hope bwapi user doesn't bother you for a while)!

As a brazilian with "dual citizenship" (mineiro-gaúcho), I have made many good friends in both states. I'm fortunate to have too many to enumerate, but I hope to acknowledge you in groups. In alphabetical order: Abençoadinhos, ELM-Doido, Galerinha, old-school MASLAB friends, Racionalismo Cristão, and Zudos. Thank you! Your friendship provides the energy that makes me go further.

It would be impossible to conduct this research without the financial support of the "Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001" and "Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil (CNPq)". I also thank AAAI and IEEE for travel grants that allowed me to present at conferences. I hope I can express my gratitude not only by those words but by giving back to society as well. I thank the DCC staff for the prompt help when needed, and for keeping the department up and running with grace and kindness. I thank the bandeijão for the affordable daily fuel at UFMG and the chill-, synth-, retro-wave composers and mixers of YouTube, the NAMCO Sound Team and Nobuo Uematsu for the motivating soundtracks, as well as the Palouse free online course on mindfulness-based stress reduction.

My final thanks goes to my wife, for her examples of magnificence in many aspects of life, for her sustained faith in me, for her continuous love and support. Thank you my joy, my Joyce! You make me the happiest man ever, for all things above, and above all else, for carrying within you our baby boy, our greatest treasure: Otto. You already inspire me, little man!

May the Force be with all of you!

"Perception is strong and sight weak. In strategy it is important to see distant things as if they were close and to take a distanced view of close things."
(Miyamoto Musashi)

Abstract

Complex computer games, with high-resolution state representations, a large number of actions and the need of reasoning in different temporal scales against an opponent, present many unsolved challenges to artificial intelligence. Those challenges gave rise to a variety of algorithms, specialized in different aspects of a game.

Human players succeed at such games by resorting to previously trained strategies, or courses of actions, and excel at generalizing responses by analogy between unforeseen and familiar situations. This dissertation presents a computational version of the human behavior: first, we replace the human repertoire of strategies by a portfolio of algorithms, modeling game-playing as an adversarial algorithm selection problem in a reinforcement learning framework. Second, we use known function approximation schemes to promote similar responses to similar game states. Our hierarchical decision-making framework uses existing algorithms, aiming to discover the best in each game situation, potentially resulting in a stronger performance than a single algorithm could reach.

We demonstrate the advantages of algorithm selection according to the number of actions in the domain, the portfolio size, and algorithms' strength, via experiments in a synthetic problem. Moreover, we instantiate our framework in real-time strategy games - possibly the most complex type of computer game - where a player must strategically develop its economy and quickly maneuver its units in combat.

Our framework allows the discussion of game-theoretic aspects of algorithm selection, in the sense of anticipating the choices of an algorithm-selector opponent, and leverages the performance of artificial intelligence in real-time strategy games by consistently outperforming state-of-the-art search-based approaches.

Palavras-chave: Artificial Intelligence, Reinforcement Learning, Algorithm Selection, Computer Games.

Resumo

Jogos digitais complexos, com representação de estados em alta resolução, grande número de ações e necessidade de raciocínio em diferentes escalas temporais contra um oponente, apresentam muitos desafios não resolvidos em inteligência artificial. Estes desafios deram surgimento a uma variedade de algoritmos, especializados em diferentes aspectos de um jogo.

Jogadores humanos prosperam nesses jogos ao recorrerem a um repertório de estratégias, ou linhas de ação, previamente treinadas e por conseguirem generalizar respostas por analogia entre situações imprevistas e familiares. Esta tese apresenta uma versão computacional desse comportamento: primeiramente, substituímos o repertório humano de estratégias por um portfólio de algoritmos, modelando o jogo como um problema de seleção de algoritmos com adversário em um arcabouço de aprendizado por reforço. Em seguida, usamos esquemas conhecidos de aproximação de funções para promover respostas similares a estados similares do jogo. Nosso arcabouço hierárquico para tomada de decisão usa algoritmos existentes, buscando descobrir o melhor em cada situação do jogo, potencialmente resultando em um desempenho melhor que um algoritmo sozinho poderia atingir.

Demonstramos as vantagens da seleção de algoritmos de acordo com o número de ações no domínio, o tamanho do portfólio e a competência dos algoritmos, através de experimentos em um problema sintético. Além disso, instanciamos nosso arcabouço em jogos de estratégia em tempo real - possivelmente o tipo de jogo digital mais complexo - no qual um jogador deve estrategicamente desenvolver sua economia e rapidamente manobrar unidades em combates.

Nosso arcabouço permite a discussão de aspectos de teoria dos jogos em seleção de algoritmos, no sentido de antecipar as escolhas de um oponente que também selecione algoritmos, além de alavancar o desempenho de inteligência artificial em jogos de estratégia em tempo real, ao derrotar de maneira consistente o estado da arte em abordagens baseadas em busca.

Palavras-chave: Inteligência Artificial, Aprendizado por Reforço, Seleção de Algoritmos, Jogos Digitais.

List of Figures

2.1 Illustration of a Markov Game 9
2.2 Markov Game associated with a hypothetical real-time strategy game. . . . 17

4.1 The Stochastic game over options is non-markovian 40

5.1 Meeting point of action- and algorithm-selector's cumulative rewards. . . . 49
5.2 Meeting point of $p^A_{a^*}$ and $p^\Pi_{a^*}$. 51
5.3 Screenshot of a StarCraft Terran base 53
5.4 A screenshot of μRTS. 55
5.5 Dominance graph for AIIDE 2015 Protoss bots 58
5.6 Overall win rate in the stationary scenario. 62
5.7 Overall win rate in the non-stationary scenario 63
5.8 MegaBot win rate per tournament progress 65
5.9 MegaBot tournament performance 66
5.10 New or updated entries in the competitions 67
5.11 MM, MR, QQ and QR policies against the three adversaries 73
5.12 Running average on 100 episodes of mean reward and win rate in the 5 repetitions against each training adversary. 76
5.13 Q-Learning, random and fixed policies over options vs search methods . . . 77
5.14 Algorithm selection with function approximation against state-of-the-art search approaches. 80
5.15 Consecutive selection of algorithms 81

C.1 Breakout screenshot. 115
C.2 Frostbite screenshot. 116
C.3 Montezuma's Revenge screenshot 117
C.4 Prismata screenshot. 117
C.5 Doom screenshot. 118
C.6 Dota 2 screenshot. 119

List of Tables

4.1 Algorithms' policies used in our simple example. 32

5.1 Number of units, buildings and technology tree depth of each StarCraft race. 53
5.2 Payoff matrix of AIIDE 2015 Protoss bots on Fortress map 57
5.3 Nash Equilibrium among selected bots. 58
5.4 Characteristics of the algorithm selection tournament agents. 60
5.5 Agents' parameters. 61
5.6 Win rate of specific agents against groups of opponents 62
5.7 Win rate of reply-matrix versus Skynet in both scenarios. 63

A.1 Win percent among AIIDE 2015 Protoss bots on Fortress map 109
A.2 Pairings of the algorithm selection tournament with stationary payoff matrix110
A.3 Pairings of the algorithm selection tournament with non-stationary payoff matrix 110

List of Symbols

Markov Games and reinforcement learning:

N	set of players. In two-player games, $N = \{1, 2\}$.
S	set of states. States in S are typically represented by s and s'.
A	set of joint actions. In two-player games, $A = A_1 \times A_2$, where A_1 and A_2 are the set of actions of each player. Typically, a_1 and a_2 are used to represent elements in A_1 and A_2, respectively.
$R(s)$	reward received by reaching state s. It is a vector with one component by player.
$\mathcal{R}(s, a_1, a_2)$	expected reward when joint action (a_1, a_2) is taken in state s. It is a vector with one component per player.
$T(s, a_1, a_2, s')$	transition function: the probability of reaching state s' by taking joint action (a_1, a_2) in state s.
α	step-size (or learning rate) parameter.
ω	decay rate of the step size.
ϵ	probability of exploration, i.e., taking a random action.

Game-playing:

γ	discount factor of future rewards in a Markov Game.
π	policy, a function that maps a state to a probability distribution over actions.
$V^\pi(s)$	state-value function: expected sum of discounted rewards for following policy π in state s.
$Q^\pi(s, a_1, a_2)$	action-value function: expected sum of discounted rewards for taking joint action (a_1, a_2) in state s and following policy π thereafter.

The strategic reasoning framework:

π	a game-playing algorithm (denoted by the policy it defines).
Π	portfolio, a set of game-playing algorithms.
$T_{\Pi}(s, \pi_1, \pi_2, s')$	transition function of the Markov Game over algorithms. Indicates the probability of reaching state s' when player 1 chooses algorithm π_1 and player 2 chooses π_2 in state s.
$T_{\Pi \times A}(s, \pi_1, a_2, s')$	transition function of the Markov Game of algorithms versus actions. Indicates the probability of reaching state s' when player 1 chooses algorithm π_1 and player 2 chooses action a_2 in state s.
$T'_{\Pi}(s, \pi_1, s')$	transition function of the Markov Decision Process over algorithms. Indicates the probability of reaching state s' when the agent chooses algorithm π_1 in state s.
$\overline{S}$	set of abstract states. An abstract state is typically represented as $\overline{s}$.
ϕ	state abstraction function. Maps a state $s \in S$ to an abstract state $\overline{s} \in \overline{S}$.
O	set of options (temporally-extended actions). In a two-player setting, the set of options for each player is denoted as O_1 and O_2.
$T_O(\overline{s}, o_1, o_2, \overline{s'})$	transition function of the Stochastic game over Options. Indicates the probability of reaching abstract state $\overline{s'}$ when player 1 chooses option o_1 and player 2 chooses o_2 in abstract state $\overline{s}$.
$\mathbf{f}$(s)	feature vector to describe state s.
$\mathbf{w}^{\pi}$	weight vector for algorithm π. The union of weight vectors for all algorithms is denoted by $\mathbf{w}$.
$\tilde{Q}(s, \pi, \mathbf{w})$	approximate value of selecting algorithm π in state s, given the weight vectors in $\mathbf{w}$.

Contents

Acknowledgments **xi**

Abstract **xv**

Resumo **xvii**

List of Figures **xix**

List of Tables **xxi**

List of Symbols **xxiii**

1 Introduction **1**

1.1 Objectives 2

1.2 Contributions 3

1.3 Chapter organization 4

2 Background **7**

2.1 Games 7

2.2 Formal model 8

2.3 Playing games 10

2.3.1 Game-tree search 11

2.3.2 Reinforcement learning 12

2.4 The complexity of real-time strategy games 14

3 Related work **19**

3.1 Adapted search approaches 19

3.2 Rule-based approaches 20

3.3 Reinforcement learning 21

3.4 Algorithm selection 24

3.5 Summary 27

4 The strategic reasoning framework **29**

4.1 The human approach 29

4.2 Game-playing algorithms 30

4.3 Strategic reasoning approaches 33

4.3.1 Algorithms versus algorithms 34

4.3.2 Algorithms vs actions 35

4.4 Issues with the state space 36

4.4.1 State aggregation 37

4.4.2 Linear function approximation 43

4.5 Summary 44

5 Experiments **47**

5.1 Synthetic experiments 48

5.2 Real-time strategy testbeds 52

5.2.1 StarCraft 52

5.2.2 μRTS 55

5.3 The game of algorithm selection 56

5.3.1 Methodology 56

5.3.2 Results 61

5.3.3 A functional StarCraft bot 64

5.3.4 Discussion 67

5.4 Multiple decision points 70

5.4.1 State aggregation 71

5.4.2 Linear function approximation 78

5.4.3 Discussion 82

5.5 Summary 84

6 Conclusion **87**

6.1 Overview 87

6.2 Contributions 88

6.2.1 A model-free, lightweight, full-game capable approach 88

6.2.2 A metagame analysis 89

6.2.3 An investigation of abstract versus low-level actions 90

6.2.4 Advancement in RTS game AI performance 90

6.2.5 Software contributions 90

6.3 Limitations 91

6.3.1 Rigid architecture 91
6.3.2 Perfect information 91
6.4 Directions for future research 92
6.4.1 Better insights from the experiments 92
6.4.2 More flexible hierarchical architectures 92
6.4.3 Handling imperfect and incomplete information 93
6.5 The purpose of game AI research 94

Bibliography 97

Appendix A Additional data on the game of algorithm selection 109

Appendix B MegaBot 111

Appendix C Other games mentioned in this dissertation 115
C.1 Breakout 115
C.2 Frostbite 116
C.3 Montezuma's Revenge 116
C.4 Prismata 117
C.5 Doom 117
C.6 Dota 2 118

Chapter 1

Introduction

Computer games have promoted development in a variety of disciplines. Examples include: engineering for faster computers, better graphic processors and novel interaction devices; arts for increasingly complex plots and game worlds; psychology for engaging gameplay; computer graphics for better, faster, and more realistic rendering techniques; and, within our interest, artificial intelligence (AI) for improved behavior of software-controlled players in terms of realism and/or challenges.

Computer games are especially challenging for AI techniques due to their real-time nature and large state spaces. Even so, recent advances allowed computers to play reflex-based games, such as some Atari classics, with super-human performance [Mnih et al., 2015]. However, many unsolved challenges remain in more complex games, which, in addition to large state spaces and the need of real-time interaction, have the following characteristics, which are our research challenges:

- Huge action spaces: in some games, the player controls dozens of elements and an action is the assignment of a command to each element. The number of possible actions is huge because each possible combination of commands to the elements is a distinct action;
- Competition and simultaneous moves: we are interested in games of two players with opposing goals, where both players' actions affect the game state;
- Long-term decisions: some games require actions that unfold in the future, such as grabbing a key to open a distant door, or constructing a defensive structure to resist a future attack. These require reasoning at different temporal scales.

Real-time strategy games have these characteristics [Buro and Furtak, 2004], being arguably the most complex type of computer game. In these, a player must strategically develop its economy, harvesting resources and constructing buildings; and quickly maneuver its units in combat.

Professional human players excel in such games by training a repertoire of strategies: they memorize the sequences of actions needed to achieve specific goals. When a strategy has been successfully trained, the player is able to recall it, performing the necessary movements with ease, liberating the attention to focus on abstract, strategic issues: assess the environment situation to decide which strategy to follow. Moreover, humans excel at generalizing by responding to unforeseen events in analogous ways as they do to familiar situations.

1.1 Objectives

We propose a strategic reasoning framework for complex computer games inspired by the human behavior, that is, with an abstract layer of reasoning and learning generalization capabilities. Our main goal is to create stronger agents to play complex computer games, adhering to the following guidelines:

(G1) An agent must play without resorting to the game's forward model: search algorithms require a game's forward model (which indicates the state reached after an action executed in a previous state) to generate successor states and deliberate which course of action to follow. However, the programming interfaces of commercial computer games do not provide forward models. We want our approach to play commercial games as well. Moreover, model-free approaches can be adapted to real-world applications easier;

(G2) The approach must not depend on powerful hardware: much of the recent advances in computer game AI results from executing machine learning approaches on extremely large datasets (e.g. hundreds of millions of samples as in [Mnih et al., 2015]) and/or using a hardware infrastructure many orders of magnitude superior to a desktop computer (e.g. 256 GPUs and 128 thousand CPU cores as in [OpenAI, 2018a]). Our approach should require minimal hardware infrastructure beyond what a desktop computer can offer;

(G3) The approach must handle all aspects of the complex game: some methods target specific aspects of a game (e.g. real-time strategy combats on Usunier et al. [2016]). Our approach must be able to play a typical complex game match from start to finish.

To handle the challenges imposed by complex computer games adhering to the stated guidelines, this dissertation presents a computational version of the human strategic behavior and generalization abilities. We replace the human repertoire of strategies by a portfolio of algorithms, modeling game-playing as an adversarial algorithm selection problem [Rice, 1976] in a hierarchical, model-free reinforcement learning framework [Sutton and Barto, 1998]. This satisfies **(G1)**.

We use simple generalization schemes (state aggregation and linear function approximation [Sutton and Barto, 1998, Chapter 8]) to promote similar responses to similar game states, without the need for long training sessions, satisfying **(G2)**. Our decision-making framework makes use of existing game-playing algorithms, satisfying **(G3)**, aiming to discover the best in each game situation, potentially resulting in a stronger performance than a single algorithm could reach.

1.2 Contributions

Our algorithm selection framework allows us to observe how algorithms, which encode game-playing strategies, interact in real-time strategy games. We analyze game-theoretic aspects of those games, albeit in a simplified representation.

However, the game-theoretic analysis requires both players to be algorithm selectors with known portfolios of algorithms, in analogy with two human rivals that know each others' strategies. In general, a player can train new strategies to surprise its opponent, and can, in theory, perform any low-level game action. Our experiments proceed without the known portfolio assumption, and our resulting agents are competitive against some state-of-the-art search approaches when we generalize learning with state aggregation. Furthermore, our agents consistently defeat the search opponents when we adopt linear function approximation.

We investigate the advantages of algorithm selection compared to learning directly over low-level game actions via experiments in a synthetic problem. Our findings confirm the intuition that it is better to select algorithms when their quality or the number of low-level actions grows.

We evaluate a simple version of our algorithm selection framework in actual StarCraft AI tournaments. Our fully-functional bot succeeded in its debut year, by ranking among the top 50% competitors, but its success decreased as its portfolio got increasingly outdated with the appearance of newer bots in the following years.

We can summarize our contributions as follows:

1. A model-free, lightweight framework for algorithm selection in complex computer games;
2. A better understanding of algorithm selection versus learning over low-level actions;
3. Discussion of game-theoretic aspects of algorithm selection in real-time strategy games;
4. Strong performance in real-time strategy games, trained in relatively small sessions in common hardware;
5. Software contributions, including a tournament-capable StarCraft bot.

The following publications are a result of this dissertation:

- [Tavares et al., 2016]: associated with Contributions 1 and 3 (with a initial framework version);
- [Tavares et al., 2018b]: further investigates the initial framework version and presents Contribution 5;
- [Tavares and Chaimowicz, 2018]: associated with Contribution 4 (as an initial step, because the actually strong agents came in the next publication).
- [Tavares et al., 2018a]: associated with Contributions 2 and 4.

1.3 Chapter organization

The remainder of this dissertation is organized as follows:

- Chapter 2: presents basic concepts and terminology used throughout the text;
- Chapter 3: discusses related work, their contributions and differences with this dissertation;
- Chapter 4: presents a formal definition of algorithm, our strategic reasoning framework and discusses our learning generalization approaches: state aggregation and linear function approximation;

- Chapter 5: presents and discusses experiments on the benefits of learning over algorithms rather than low-level actions and investigating game-theoretical aspects of algorithm selection or game-playing performance over real-time strategy games;
- Chapter 6: gives a summary of this dissertation, presenting concluding remarks, as well as reviewing the contributions and discussing directions for future research.

Chapter 2

Background

This chapter presents the theoretical background, terminology and definitions used in this dissertation.

2.1 Games

Formally, a game is a mathematical formulation of *strategic interactions* among independent and self-interested agents, or players. Game theory [Shoham and Leyton-Brown, 2009; Osborne and Rubinstein, 1994; Nisan et al., 2007] studies such interactions. Strategic interaction refers to the fact that an agent considers the interests and possible choices of its peers to make its own decision. Such interactions occur in various contexts, including economy, politics, biology, psychology and computer science. In special, the formal concept includes the popular understanding of a game as an "entertainment"[1] activity; examples including Chess, Go, computer games and a variety of sports.

The models of game theory usually assume rational agents, which act so as to bring the best possible situation to themselves, considering that their peers will also pursue their goals.

A game can be represented in various forms, considering the number of players, the nature of rewards, sequential or one-shot decisions, simultaneous or alternate actions, partial or full observation, among others. We focus on the particular class of two-player, zero-sum, fully-observable, sequential-interaction games. In these, players compete for the victory, thus having opposite goals. We often identify them as the *agent* and the *opponent*. Moreover, our games of interest are played on a sequence of

[1]The term is quoted because such games indeed have a playful purpose, but several are seriously studied and practiced professionally.

simultaneous moves. Additionally, we are interested in games with challenging characteristics for current state-of-the-art artificial intelligence techniques, as discussed next.

2.2 Formal model

We adopt Markov Games (also known as Stochastic games [Shapley, 1953]) as our decision model. The formalism is general enough to encompass multiple players, simultaneous moves, chance events, cooperative and competitive scenarios with full observability (players have perfect state information). Here we restrict our attention to two-player, finite, zero-sum (purely competitive) games.

Definition 1 *A finite, two-player, zero-sum Markov Game is a tuple $MG = (N,S,A,R,T)$, where each element is defined as follows:*

- *$N = \{1, 2\}$ is the set of players;*
- *S is the set of game states, with the Markov property*[2]*.*
- *$A = A_1 \times A_2$ is the set of joint actions, where A_i is the set of player i's actions. When needed, we denote the set of actions available to player i in state s as $A_i(s)$;*
- *$R : S \to \mathbb{R}^2$ is the (state-)reward function. $R(s)$ is a vector of two real numbers, $\langle R_1(s), R_2(s)\rangle$, indicating the rewards obtained by player 1 and 2 for reaching state s;*
- *$T : S \times A_1 \times A_2 \times S \to [0, 1]$ is the transition function. $T(s, a_1, a_2, s')$ denotes the probability of reaching state s' when player 1 takes action a_1 and player 2 takes action a_2 in state s.*

It is also useful to define the expected reward function $\mathcal{R} : S \times A_1 \times A_2 \to \mathbb{R}^2$. $\mathcal{R}(s, a_1, a_2)$ is a vector of two real numbers, $\langle \mathcal{R}_1(s, a_1, a_2), \mathcal{R}_2(s, a_1, a_2)\rangle$, indicating the expected rewards obtained by player 1 and 2 when they take the joint action (a_1, a_2) in state s. In a zero-sum game, $\mathcal{R}_1(s, a_1, a_2) = -\mathcal{R}_2(s, a_1, a_2)$ for all states and actions. The state- and expected reward functions differ in that the state-reward function R is perceived upon reaching a state, regardless of the previous state or actions, whereas the expected reward $\mathcal{R}$ is associated with leaving a state via a joint action, considering

[2] A state with the Markov property has all the relevant information for the agents to make decisions. More formally, if the state has the Markov property, then the environment's response to agents' actions depends only on the current state and the actions taken, rather than the whole history of states and actions [Sutton and Barto, 1998, Section 3.5].

all possible future states. The two reward functions are associated by the following relation[3]*:* $\mathcal{R}(s, a_1, a_2) = \sum_{s' \in S} T(s, a_1, a_2, s') \cdot R(s')$.

Figure 2.1 illustrates a Markov Game. Players act simultaneously and the transition is stochastic: many states can be reached from a previous state and a joint action.

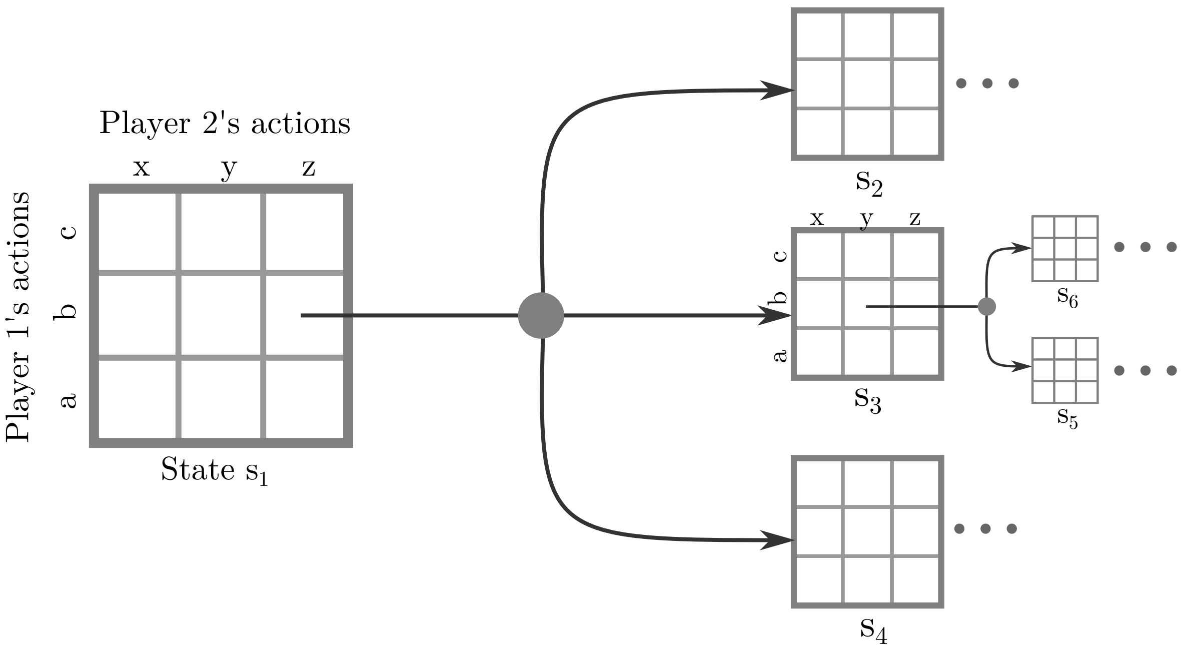


Figure 2.1: Illustration of a Markov Game. Each grid represents a state, rows represent player 1's actions and columns represent player 2's actions. Players take actions jointly and many successor states can be reached because of the stochastic transition function. In this example, joint action (b,z) in state s_1 can reach three possible successor states and joint action (b,y) in state s_3 can reach two possible successor states.

In this work we model computer games as Markov Games. Computer games have huge state spaces in general and some genres, such as real-time strategy games, have huge action spaces in addition. Nevertheless, they are still Markov Games, albeit large-scale ones. Even the asynchronous, or real-time nature of computer games is accommodated in Markov Games: each player has a passive no-op action, which is taken by default when the interval between game frames runs out. For a human player, it corresponds to not interacting with the game's graphical interface. For a computer player, it corresponds to not issuing any command.

[3]Reinforcement learning literature traditionally presents Markov Games with the expected reward function, but on this dissertation we use the state-reward function as it resembles the perception of a player in a computer game, in the sense of knowing whether she is in advantage or not in a given state. The relationship among both functions expresses that they are exchangeable and either one can be adopted.

Littman [1994] remarks that Markov Games generalize both normal-form games [Shoham and Leyton-Brown, 2009, Chapter 3], which are single-state Markov Games and Markov Decision Processes [Sutton and Barto, 1998, Chapter 3], which are single-player Markov Games. Additionally, Bošanský et al. [2016] remarks that Markov Games generalize alternating-move games such as Chess and Go: we can partition S into subsets "owned" by each player, such that a player has only the no-op action in the states owned by the opponent.

2.3 Playing games

This section briefly discusses the foundations of techniques used by computers to play games modeled after the Markov Games formalism (Section 2.2). Advanced variations are discussed in Chapter 3.

For the purposes of this dissertation, we adopt the *infinite discounted horizon* optimality model, where a discount factor $\gamma \in [0, 1]$ goads agents to prefer short-term rewards more than long-term ones, if set to less than one. If, by interacting with the environment from time-step t, the agent receives a sequence of rewards $r_t, r_{t+1}, ...$, in the infinite discounted horizon model it aims to maximize the expected sum of discounted rewards: $\mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k}\right]$.

Games are played by selecting a valid action in the current game state. Formally, a computer agent plays a Markov Game according to a policy $\pi : S \times A \to [0, 1]$, that is, a mapping from states to probability distribution over actions. Deterministic policies are special cases, denoted as $\pi : S \to A$.

Two policy-related functions indicate the specified behavior quality. Considering player 1's point of view, the state-value function $V^\pi(s)$ indicates the expected sum of discounted rewards she will receive by following policy π from state s. The action-value function $Q^\pi(s, a_1, a_2)$ indicates the expected sum of discounted rewards she will receive by performing action $a_1 \in A_1$, while the opponent selects $a_2 \in A_2$ in state s and player 1 follows π thereafter. The optimal value functions V^* and Q^* are thus associated with the optimal policy. They are calculated recursively, according to Equations 2.1 and 2.2, which are the *Bellman optimality equations* for player 1.

$$V^*(s) = \max_{\pi(s,\cdot)} \min_{a_2 \in A_2} \sum_{a_1 \in A_1} Q^*(s, a_1, a_2) \cdot \pi(s, a_1) \tag{2.1}$$

$$Q^*(s, a_1, a_2) = \mathcal{R}_1(s, a_1, a_2) + \gamma \sum_{s' \in S} T(s, a_1, a_2, s') \cdot V^*(s') \tag{2.2}$$

In zero-sum Markov Games, the optimal policy, associated with the optimal value functions, is the maximin policy. It maximizes the agent's expected reward when the opponent performs its best action (hence the $\max_{\pi(s,\cdot)} \min_{a_2 \in A_2}$ in Eq. 2.1). Moreover, each Markov Game state $s \in S$ is a normal-form game, whose payoffs are $Q^*(s, a_1, a_2)$ for all $(a_1, a_2) \in A_1(s) \times A_2(s)$. Determining $V^*(s)$ and the associated maximin policy $\pi^*(s, \cdot)$ corresponds to solving this zero-sum normal-form game, which can be done via linear programming [Nisan et al., 2007, Section 1.4.2].

The goal of game-playing computers is to discover the optimal policy, which is equivalent to solving a game. To solve the entire Markov Game, one needs to determine the optimal/maximin policy (and/or the value functions) for all states. The challenge lies on the recursion: values for a state depend on their successors, which depend on their own successors and so on. In theory, all finite games are solvable. In practice, increasingly large games were solved due to algorithmic and technology advancements, including 9 Men's Morris [Gasser, 1996], Checkers [Schaeffer et al., 2007] and *Heads-up Texas Hold'em Poker* [Bowling et al., 2015]. However, in even larger games, a computer attempts to approximate the optimal policy as close as possible within the available computational budget.

The next sections present the foundations of two major game AI approaches that aim at solving games and/or finding feasible game-playing policies: game-tree search and reinforcement learning.

2.3.1 Game-tree search

Game-tree search approaches operate by traversing or generating a tree induced by the underlying Markov Game model. In this tree, a node is associated with a game state and arcs are the actions connecting a state to its successors. The game-tree search algorithm uses the game's transition function T, also referred to as the forward model, to discover the next state, given the current state and players' actions. By iterating this process, a terminal node is eventually reached. The value of this node is back-propagated up to the tree's root. This is equivalent to expanding the right-hand side of Equations 2.1 and 2.2 successively until a terminal state (without successors) is found.

The backward induction algorithm (analised in [Bošanskỳ et al., 2016]) is a prominent game-tree search approach in this framework. The well-known α-β pruning [Knuth and Moore, 1975] algorithm is a special case of backward induction for games with alternating moves, which prunes nodes with value provably out of upper- (α) and lower- (β) bounds, calculated from previously visited nodes. In practice, where most games have deep game trees, a depth-limited version of the algorithms is employed: the traversal is interrupted at a certain depth, and a state evaluation function estimates the value of the reached state. This is equivalent to stopping the expansion of Equations 2.1 and 2.2 and replacing V^* by an estimator at the reached state.

Monte Carlo Tree Search [Browne et al., 2012] iteratively grows a game-tree by expanding previously unvisited nodes, starting from the root. Expansion is done by simulating the remainder of the match with random movements. The expanded node receives the reached terminal node's value, and its predecessors are updated retroactively up to the tree's root. MCTS does not require a state evaluation function, in contrast with the depth-limited version of α-β pruning.

Game-tree search approaches require knowledge of the game's forward model (its transition function) to generate successor states and traverse the game tree. In the real-world and in most commercial games, the transition function is not accessible, and the application of these approaches becomes limited.

2.3.2 Reinforcement learning

Reinforcement Learning (RL) is a framework for learning to act by successive interactions with the environment [Sutton and Barto, 1998]. In the presence of an adversary, interaction proceeds as follows: at each time step, agents perceive their current state and select their actions. In the next time step, agents perceive their new state and receive a reward signal. Single-agent reinforcement learning tasks are modeled as Markov Decision Processes [Sutton and Barto, 1998, Chapter 3], which Markov Games generalize to multiagent environments [Littman, 1994].

Even without knowing the transition function, an agent can learn the value functions and the optimal/maximin policy of a Markov Game via the minimax-Q algorithm of Littman [1994]. In minimax-Q, the agent updates action-value estimates Q at each interaction with the environment, according to the update rule in Equation 2.3, where $V(s)$ is given by Equation 2.1, replacing the optimal value functions V^* and Q^* by their estimates V and Q, respectively. Interaction with the environment is as follows: in state $s \in S$, the agent performs an action $a_1 \in A_1(s)$ whereas the opponent performs an action $a_2 \in A_2(s)$. The environment proceeds to state s' and the agent receives a reward $r = R_1(s')$.

$$Q(s, a_1, a_2) \leftarrow Q(s, a_1, a_2) + \alpha \left[r + \gamma V(s') - Q(s, a_1, a_2)\right] \tag{2.3}$$

In Equation 2.3, $\alpha \in [0, 1]$ is the step size, also known as learning rate, which indicates how much the estimate moves towards the target $(r + \gamma V(s'))$.

[Littman, 1996, Section 5.6.1] shows that minimax-Q asymptotically converges to Q^* with probability 1, given that some conditions are satisfied[4]:

1. Each pair (state, joint-action) has non-zero probability of being visited;
2. The received reward in each step has finite variance;
3. The step size decays with time, such that $\sum_{t=0}^{\infty} \alpha_t = \infty$ e $\sum_{t=0}^{\infty} (\alpha_t)^2 < \infty$.

Algorithm 1 presents a minimax-Q agent situated in a Markov Game $MG = (N,S,A,R,T)$ (see Section 2.2), considering itself as agent 1 and the opponent as agent 2. In the INITIALIZE procedure, the minimax-Q agent receives the initial step size α, the step size decay factor $\omega \in [0,1]$, the exploration probability $\epsilon \in [0,1]$, and the discount factor γ as parameters, which are kept as internal variables for use in the other procedures. The estimates Q, V and the corresponding policy π are also initialized. Procedure ACT receives the environment state and returns an action according to the ϵ-greedy rule. Procedure LEARN receives the previous environment state s, the actions performed by both agents a_1 and a_2, the received reward r and the reached state s'. The procedure updates the estimates Q, V, and the policy π, decaying the step size afterwards. The policy π is determined (in line 18) via the resolution of the normal-form game related to s[5]. Agent initialization, action and learning processes are sequenced in the TRAIN procedure, which receives the agent parameters and the number of training episodes e from the user.

Algorithm 1 follows the original of Littman [1994]. Q and V initialization is optimistic (i.e. with maximum reward), which encourages premature exploration. The policy π is initialized so as to select all actions with equal probability.

In some situations, it is useful to maintain a constant step-size, by making $\omega = 1$. This violates Condition 3 to convergence, but allows one to track a non-stationary problem. The discount factor γ can be set to 1 in episodic tasks (as our games). Even so, lower discount factors goads the agent to win the game sooner.

The minimax-Q algorithm belongs to a class of model-free methods. Model-free approaches are especially appealing for the broad range of problems they can tackle, because, in contrast with search approaches, a game's forward model (its transition function) is not required for an agent to successfully learn a policy.

[4][Littman, 1996, Section 5.6.1] lists more conditions as they serve to generalized Markov Decision Processes. The conditions omitted here are automatically satisfied by our Markov Game definition.

[5]The payoff matrix of the normal-form game related to s is filled with the values in $Q(s, a_1, a_2)$ for all $(a_1, a_2) \in A_1(s) \times A_2(s)$. The resolution of such normal-form game via linear programming is described in [Nisan et al., 2007, Section 1.4.2].

Algorithm 1 Minimax-Q

```
1: procedure INITIALIZE($\alpha, \omega, \epsilon, \gamma$)
2: Input: initial step size $\alpha$, decay factor of the step size $\omega$, exploration probability $\epsilon$ and discount factor $\gamma$
3:     Maintain $\alpha, \omega, \epsilon$ and $\gamma$ as internal variables to use in the other procedures.
4:     Initialize $V(s)$ optimistically for each $s \in S$
5:     Initialize $Q(s, a_1, a_2)$ optimistically for each $s \in S, a_1 \in A_1, a_2 \in A_2$
6:     $\pi(s, a_1) = \frac{1}{|A_1(s)|}$ for each $s \in S, a_1 \in A_1(s)$
7: end procedure
8:
9: procedure ACT(s)
10: Input: environment state $s$
11:     if random() $< \epsilon$ then return a random action in $A_1(s)$
12:     else return an action according to $\pi(s, \cdot)$
13: end procedure
14:
15: procedure LEARN($s, a_1, a_2, r, s'$)
16: Input: previous state $s$, players' actions $a_1$ and $a_2$, reward $r$ and reached state $s'$.
17:     $Q(s, a_1, a_2) \leftarrow Q(s, a_1, a_2) + \alpha\,(r + \gamma V(s') - Q(s, a_1, a_2))$     ▷ Equation 2.3
18:     Determine $\pi(s, \cdot) = \text{argmax}_{\pi(s,\cdot)} \min_{a'_2 \in A_2} \sum_{a'_1 \in A_1} \pi(s, a'_1) Q(s, a'_1, a'_2)$
19:     $V(s) = \min_{a_2 \in A_2} \sum_{a_1 \in A_1} \pi(s, a_1) Q(s, a_1, a_2)$
20:     $\alpha \leftarrow \alpha \cdot \omega$
21: end procedure
22:
23: procedure TRAIN($\alpha, \omega, \epsilon, \gamma, e$)
24: Input: initial step size $\alpha$, decay factor of the step size $\omega$, exploration probability $\epsilon$, discount factor $\gamma$, and number of episodes $e$
25:     INITIALIZE($\alpha, \omega, \gamma, \epsilon$)
26:     for $e$ episodes do
27:         $s \leftarrow$ initial state in $S$
28:         while $s$ is not terminal do
29:             $a_1 \leftarrow$ ACT(s)
30:             Observe opponent action $a_2$, the resulting state $s'$ and reward $r$.
31:             LEARN($s, a_1, a_2, r, s'$)
32:             $s \leftarrow s'$
33:         end while
34:     end for
35: end procedure
```

2.4 The complexity of real-time strategy games

For our purposes, the term "complex computer game" refers to games with all the following characteristics: huge state and action spaces (we make the term "huge" precise next), real-time interaction, two players, simultaneous actions and long-term decisions.

Real-time strategy is our genre of choice because these games present all aforementioned challenges. We sometimes refer to "complex games" in general to remind that our approach is suitable to any scenario with the foregoing characteristics.

Real-time strategy games are a class of computer games where players deal with multiple tasks: resource collection, construction of buildings, technological improvements and battles against enemy armies [Cunha and Chaimowicz, 2010]. Usually, players start with a base near resource fields and some workers, which are resource-harvesting and building-construction units. From collected resources, players can train more workers, strengthening their economy, and/or construct new buildings that allow military unit production and technological advancements, which unlock abilities and increase units' offensive and defensive capabilities. A player wins the game by defeating enemy armies and/or destroying its buildings. Decisions involving resource management, balancing military and economic aspects are referred to as *macro-management.* Fast-paced decisions involving unit maneuvers in battle are referred to as *micro-management.*

In this dissertation we deal with the practical complexity of real-time strategy games. To the interested reader, the theoretical complexity of *attrition games on graphs*, a simplified version of real-time strategy combats, is discussed in [Furtak and Buro, 2010].

We start by associating an instance of a real-time strategy game with our Markov Game model (see Definition 1) in Example 1, and proceed by showing the complexity of the resulting model.

Example 1 *A real-time strategy game modeled as Markov Game. The states, actions, rewards and transitions are as follows:*

- *States: the state of a real-time strategy game with the Markov property is a snapshot of all current game data (i.e. the game map and all information of resources, buildings and units, including those under production, plus the elapsed time or number of frames since the game has started). Terminal states are reached when the game time runs out or a player is defeated by having all its buildings razed and/or units killed;*

- *Actions: each player i controls a set of units U_i. Each unit $u_i \in U_i$ has a set of available actions $A_{u_i}(s)$ in a given state. The actions A_i of player i are the possible combinations of all its unit's actions, plus the passive no-op action: $A_i = \{no\text{-}op\} \cup A_{u_i} \times \cdots \times A_{u_{|U_i|}}$;*

- *Rewards: the reward signal can be* $+1, 0$ *or* -1 *respectively for the victory, draw or defeat, perceived when the game reaches a terminal state. Alternatively, in-game score measuring material advantage can be used as a reward not only in terminal states, but on non-terminal states as well.*

- *Transition: advances the game to the state corresponding to the next frame, by executing all actions issued in the current state. That is, it accounts all damage received by attacked units and buildings, removing dead units and razed buildings, counts the cooldown between successive unit attacks, starts or progresses the construction of buildings and the production of units, performs physical movement calculations and effects of special abilities such as spells. Stochastic effects might arise, which include the chance of a unit being hit and the amount of hit points (HP) reduced when a unit is damaged.*

Figure 2.2 illustrates the association of a hypothetical real-time strategy (RTS) game with the Markov Game model. Green squares are resource fields. White squares are bases, gray circles are workers. Player 1 owns the blue and player 2 the red units. The states have all game information: all unit attributes and the game time. Figure 2.2 shows some of the attributes of the units: hit points (HP), resources (Res), time to finish (TTF), and their position in the grid map. Besides, there is a one-to-one correspondence between Markov Game states and physical RTS game states. From a given state, the RTS game engine receives players commands (the joint actions) and computes the next state.

The resulting Markov Game associated with a real-time strategy game has huge state and action spaces. The difference between two successive states is the smallest possible (one game frame), such that we have a state for each possible combination of all game elements and their attributes. A player action in real-time strategy games is the assignment of a command (or unit action) to each game unit. Some unit actions are complex as they involve target parameters, which usually are map positions or other units. In StarCraft, Ontañón et al. [2013] conservatively estimated $|S| \approx 10^{1000}$ and, for each player i, $|A_i(s)| \geq 10^{50}$, on average for each state. As a reference, Chess has 10^{50} states and Go has approximately 10^{170}. In each state, Chess has $|A_i(s)| \approx 35$ actions and Go has $|A_i(s)| < 81$.

Additionally, real-time strategy games have imperfect (partial) information, so that a player can observe objects within its units' visual range. However, the Markov Game model assumes perfect information. Hence, in this dissertation we enable perfect information when necessary.

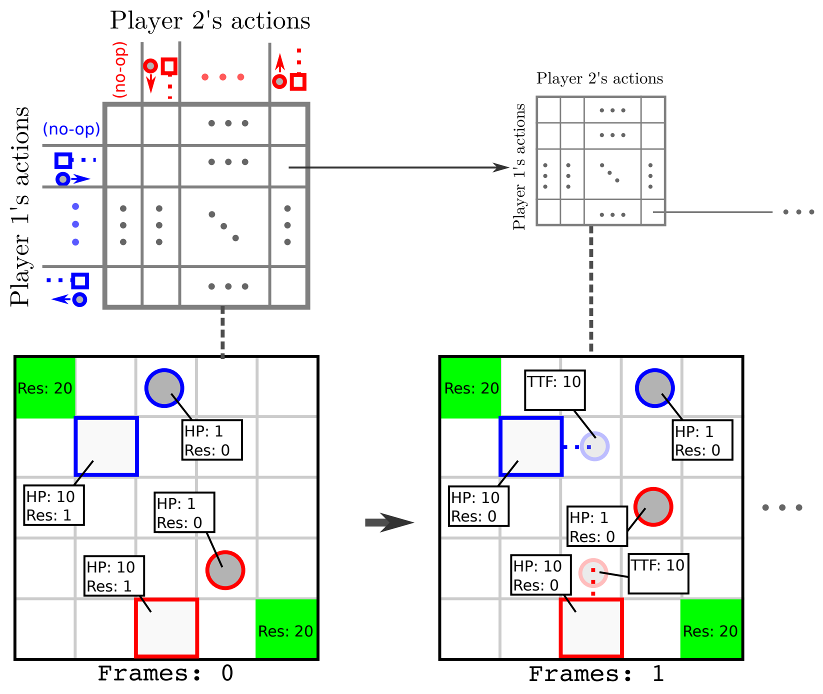


Figure 2.2: Markov Game associated with a hypothetical real-time strategy game. The top part represents the Markov Game states with the available actions of each player (blue in the rows, red in the columns). The bottom part shows the associated physical game states. In the example, in frame 0, the action of the blue player was to move her worker to the right and order her base to train a new worker at its right, whereas the action of the red player was to move his worker up and order his base to train a new worker upwards. This joint action resulted in the state shown at frame 1, where the alive workers moved in the intended directions and the new workers started being trained.

Chapter 3

Related work

Many landmarks in computer game artificial intelligence (AI) stem from improved versions of the basic α-β pruning [Knuth and Moore, 1975] or Monte Carlo Tree Search (MCTS) [Browne et al., 2012] algorithms (see Section 2.3.1 for a brief description).

Approaches based on MCTS and α-β pruning succeed in board games, but cannot satisfactorily handle the challenges imposed by complex computer games. In special, the enormous state and action spaces in those games make these algorithms evaluate insufficient positions to make any reasonable decision in a timely fashion.

This chapter reviews some techniques geared towards computer games, divided in four groups: adapted search approaches (Section 3.1), rule-based approaches (Section 3.2), reinforcement learning (Section 3.3) and algorithm-selection approaches (Section 3.4). We finish with a summary of approaches, showing how this work differs from current state-of-the-art (Section 3.5).

3.1 Adapted search approaches

Part of the research effort in large-scale games focuses on using game-tree search algorithms, such as α-β pruning [Knuth and Moore, 1975] or MCTS [Browne et al., 2012] in abstract representations of game states. Those aim to reduce the game-tree search branching factor and depth. The branching factor indicates the number of successors of a state and the tree's depth indicates the number of needed moves for a game to finish.

Uriarte and Ontañón [2014] propose an abstract representation to StarCraft, by grouping map pixels into regions, similar units in squadrons, ignoring economic aspects and considering only moves and attacks to other regions as actions. Similar ideas are adopted in [Stanescu et al., 2014; Uriarte and Ontañón, 2016].

Abstract representations allow the use of traditional game-tree search algorithms, although the abstraction quality is influenced by the designer's domain knowledge. The resulting simulations contain imprecisions and might hurt the algorithms' decisions, which happens in [Uriarte and Ontañón, 2016], when the algorithm is configured to look too much ahead.

Recent advances in search approaches to real-time strategy games aggressively prune the actions to consider. For example, Adversarial Hierarchical Task Network planning, or AHTN for short [Ontañón and Buro, 2015] extends hierarchical-task network (HTN) planning, which encode domain knowledge via the definition of useful tasks in the domain, with a minimax-like game-tree search. By encoding domain tasks with domain knowledge, one narrows the possible choices to consider.

PuppetSearch is a framework that augment the capabilities of game-playing scripts by means of move choices they expose to so-called tactical search algorithms. As computational budget allows, more choices can be exposed so the search algorithms can investigate with a broader perspective. Puppet-$\alpha\beta$ [Barriga et al., 2015] uses a version of the α-β considering durations (ABCD) as the tactical search algorithm [Churchill et al., 2012], whereas PuppetUCT [Barriga et al., 2017] uses a version of Upper-Confidence bound for Trees Considering Durations (UCTCD) [Churchill and Buro, 2013]. StrategyTactics [Barriga et al., 2017] uses a convolutional neural network to predict the output of PuppetSearch, saving time for the tactical search algorithm.

NaiveMCTS [Ontanón, 2013] builds on Monte Carlo Tree Search, using a sampling strategy based on combinatorial multi-armed bandits (and thus does not use scripts).

AHTN, Puppet-$\alpha\beta$, PuppetUCT, StrategyTactics and NaiveMCTS are implemented in μRTS and we use them as opponents in our experiments (see Sections 5.4.1 and 5.4.2).

A drawback of adapted search approaches with handcrafted abstract representations is that they require expert knowledge. Moreover, all search approaches require the forward model to operate.

3.2 Rule-based approaches

In commercial computer games, non-player characters are usually controlled by sets of "if [condition] then [action]" rules. The set of rules that dictates an agent's behavior is often referred to as a script. Scripts' advantages are their legibility, easing development and understanding of behaviors; and speed, which suits computer games where processing time is divided between the game's logic, graphics and agents' decision making.

On the other hand, scripts lack flexibility: once a player detects patterns on game agents' behavior, the challenge goes away and the game becomes less fun. Nevertheless, scripts' rule sets might be useful to compose more sophisticated decision making methods, which this section reviews.

In General Game Playing (GGP)[1], Świechowski and Mańdziuk [2014] use an heuristic portfolio to guide the simulation stage of a MCTS variant. Authors test various methods to choose the search-guiding heuristic, concluding that UCB (Upper-Confidence Bounds [Auer et al., 2002]) is the best. A similar approach is used by Churchill and Buro [2015] in Prismata, a turn-based strategy game (see Section C.4). A script portfolio generates valid actions to be investigated by a MCTS variant.

Dynamic scripting [Spronck et al., 2006] aims to dynamically construct scripts from a rule base. In dynamic scripting, each agent owns a rule base and a number of slots to receive rules. A mechanism similar to reinforcement learning [Sutton and Barto, 1998] updates rules' weights and those with highest weights are favored to fill the slots. The set of rules filling the slots form the agent-controlling script.

Dynamic scripting applications include Role-Playing Games (RPG) battles[2] [Spronck et al., 2006], real-time strategy games [Dahlbom and Niklasson, 2006] and air combat simulation [Toubman et al., 2016]. Szita and Lõrincz [2007] uses a dynamic scripting-like technique in Ms. Pac Man, adapted to allow the activation of multiple rules and the cross-entropy method of Rubinstein [1999] to update the weights, rather then the reinforcement-learning-like original mechanism.

A drawback of dynamic scripting is the need of prior rule design, which requires domain knowledge from the designer.

3.3 Reinforcement learning

In complex games, reinforcement learning is usually employed with function approximation [Sutton and Barto, 1998, Chapter 8], whose basic idea is to generalize a learned value to similar states.

In board games, remarkable examples include Samuel's checkers player [Samuel, 1959, 1967], *TD-Gammon* for backgammon [Tesauro, 1995] and Alpha Go [Silver et al., 2016] (including follow-ups, discussed next). Those are remarkable because Samuel's program was the first with significant use of any kind of learning [Russell and Norvig,

[1]In General Game Playing (GGP), a single algorithm must handle a multitude of games, which fosters the development of domain-independent approaches. The games have characteristics of board games: they are deterministic, multi-player and synchronous [Genesereth and Thielscher, 2014].

[2]In such turn-based battles, each player controls a small team of agents and each agent has attacks, spells and skills to harm the opponents or strengthen the allies.

2003, p. 850]; TD-Gammon matched the best human backgammon players of that time and Alpha Go defeated one of the currently best human Go players by combining expert game analysis and self-play. Alpha Go was further surpassed by AlphaGo Zero [Silver et al., 2017b], which relies solely on self-play, and Alpha Zero [Silver et al., 2017a], which plays not only Go, but Chess and Shogi (Japanese Chess) as well.

Reinforcement learning reached impressive performance in computer games as well, with the appearance of the *arcade learning environment* (ALE) [Bellemare et al., 2012], which provides a programming platform for classical Atari 2600's games. In the ALE, programs receive the screen's 160×210 pixels and a score information as input and must return one of the 18 possible actions allowed by Atari's joystick.

Mnih et al. [2015] present a deep reinforcement learning architecture, named *deep Q-network* (DQN), which uses a convolutional neural network [Le Cun et al., 1990] to approximate the action-value function (Q). Those are composed by hierarchical layers of convolutional filters, capable of detecting spatial correlations in images, producing increasingly abstract representations from input data. In [Mnih et al., 2015], the most recent game frames are pre-processed to form the network's input "image", allowing the detection of object trajectories. Their approach outperforms other reinforcement learning methods in 43 out of 49 tested games and reaches at least 75% of a professional human game tester in 29 games. In some games, DQN's performance is several times better than the human's. However, DQN performs poorly in games requiring longer planning horizon, such as *Frostbite* (see Section C.2) and Ms. Pac-Man.

Mnih et al. [2015]'s DQN is a significant contribution to the task of control from high dimensionality sensory inputs. Liang et al. [2016] analyses DQN's success factors, such as the ability to detect space-invariant features, like relative object positions, and short-term temporal features, such as trajectories. Liang et al. [2016] provide simple linear approximators to detect those features to replace DQN's deep learning. In fact, the performance of the "shallow" architecture of Liang et al. [2016] is comparable to Mnih et al. [2015], although methodological issues prevented a completely fair comparison. For example, Mnih et al. [2015] reports performance on repeated tests of the best obtained network for each game, rather than the average of multiple train and tests. Liang et al. [2016] notes that subsequent research may have some obstacles to reproduce Mnih et al. [2015]'s results.

Hausknecht and Stone [2015] extended DQN with recurrence to consider temporal aspects directly on the network's architecture. In contrast, Mnih et al. [2015]'s original DQN "incorporates" temporal aspects through the pre-processed input of the most recent game frames. Hausknecht and Stone [2015]'s *deep recurrent Q-network* (DRQN), even receiving a single game frame, can integrate information along time, outperforming the original DQN in games such as *Frostbite*, which require longer-term planning horizons. Nevertheless, DRQN's performance is around 50% of the human tester, reported in [Mnih et al., 2015].

Lample and Chaplot [2016] use DRQN to handle a three-dimensional domain: the *Visual Doom* platform [Kempka et al., 2016], which provides screen pixels of the first-person shooter *Doom* (see Section C.5) for a program. Authors modify DRQN to account for extracted game features, such as enemies and items, on training. Besides, they divide the game in two tasks, navigation and combat, training a separate network to each one. Kempka et al. [2016] reports that Lample and Chaplot [2016]'s approach outperforms native game bots and human players, although there is no indication of their proficiency.

So far, reinforcement learning approaches succeed in reactive games, such as *Breakout* (see Section C.1), for example. In such games, an elaborate sequence of actions is not required to reach a goal: the player just needs to promptly react to the current elements on screen. Games such as *Frostbite* (see Section C.2) are more challenging than reactive ones, as they require longer action sequences to reach a goal. Reinforcement learning is still behind humans in those games. *Montezuma's Revenge* (see Section C.3) represents the greatest challenge for computers in the ALE. Besides requiring a long-term planning horizon, the rewards are very sparse, in contrast with Frostbite, where the agent constantly scores by jumping back and forth on ice blocks.

Long-term planning horizons and sparse rewards are also issues in real-time strategy games: actions might affect the game far away from where they were taken, and a player is rewarded when the match ends[3]. For example, shallow [Wender and Watson, 2012] and deep [Usunier et al., 2016; Peng et al., 2017] reinforcement learning approaches have been successfully applied to combats in real-time strategy games. Combats are reflex-based tasks, where the player must promptly react to the current elements on screen as much as ALE's reflex-based games, successfully played by Mnih et al. [2015]'s DQN. None of these approaches tackled or succeeded in games requiring reasoning in different temporal scales, which seems to be the next challenge for reinforcement learning techniques in computer games. Reasoning in different temporal scales can be achieved by decomposing the task in a hierarchical structure such as in [Dietterich, 2000], which requires prior knowledge in the form of subgoals and policies to achieve them, and [Kulkarni et al., 2016], where subgoals are provided, but the policies to achieve them are learned. In [Kulkarni et al., 2016], the upper layer in a two-level

[3]Reward shaping, that is, giving agent constant rewards based on its material advantage can remedy this issue, although in our experiments, the agent adopted a myopic behavior (see Section 5.4.1.2).

hierarchy observes the environment state and reward, and generates intrinsic reward signals to the lower layer, which learns the policy to achieve the subgoal. The proposed approach remedies the deficiency of Mnih et al. [2015] in Montezuma's Revenge, using handcrafted goals. Machado et al. [2017] further proposes an approach to discover goals based on intrinsic reward functions that direct the agent towards traversing the state space in directions specified by a learned representation. The proposed approach was able to discover similar policies to the ones achieved via the handcrafted goals of Kulkarni et al. [2016].

Our approach is similar to Kulkarni et al. [2016]'s in the sense of a two-level hierarchical decision-making. However, Kulkarni et al. [2016]'s lower-layer is a reinforcement learning approach with access to the underlying domain's low-level actions. As our underlying domain of interest has huge action spaces, such approach is unfeasible.

Recently, a professional Dota 2 (see Section C.6) player has been defeated on a 1-vs-1 match by a reinforcement learning agent trained in self-play [OpenAI, 2017], although little technical information on the approach was released. Besides, usual Dota 2 matches are 5-vs-5, which require a team of coordinating agents.

OpenAI [2018a] presents further technical details on the approach of OpenAI [2017], as well as results of tests with a team of bots against amateur human players on 5-vs-5 matches. Each agent receives a high-dimensional input for the state (20 thousand values) and must output an action among a thousand available ones at each game frame. Reward accounts for in-game performance metrics (kills, assists, etc.) and a teamwork component. Using a version of Proximal Policy Optimization [Schulman et al., 2017], the team of agents, named Open AI Five, trained in self-play, is able to outperform a team of amateur human players. Open AI Five has lost to professional players in a subsequent test, although it exhibited strong gameplay [OpenAI, 2018b].

The drawback of the approach of OpenAI [2018a] is the necessity of powerful hardware: the team of agents trains for an equivalent of 180 years a day, on a cloud infrastructure with 128 thousand CPU cores and 256 GPUs.

3.4 Algorithm selection

The algorithm selection problem consists in defining a mapping from problem instances (or their features) to algorithms designed for that problem [Rice, 1976]. Algorithm selection techniques have been applied to complex problems either in theory (NP-complete) [Xu et al., 2008] or in practice, such as sorting [Lagoudakis and Littman, 2000] and real-time path planning [Sigurdson and Bulitko, 2017]. This section discusses some algorithm selection techniques applied to computer games.

Most works reviewed in this section are not framed as algorithm selection methods by their authors, but they fit the framework. That is, they are concerned with mapping game states to algorithms (mostly called scripts or strategies), to maximize game-playing performance.

An early application of algorithm selection in games appears with *machine games* [Abreu and Rubinstein, 1988], where players select finite automata to play the iterated Prisoner's Dilemma on their behalf. Players attempt to maximize their payoffs while reducing the complexity of the selected automata, in terms of the number of the automata's internal states.

In *General Video Game Playing* (GVGP)[4], Bontrager et al. [2016] analyze the behavior of algorithms in various games, extracting game features for classification and choosing a suitable algorithm. In this dissertation, we consider a similar approach, tackling an adversarial domain, whereas in GVGP, the agent does not face other game-playing programs.

Li and Kendall [2015] apply hyper-heuristics, which resembles algorithm selection, in the repeated prisoner's dillema, Goofspiel[5] and competitive traveling salesman problem[6]. In [Li and Kendall, 2015], however, algorithm selection techniques are tested against non-adaptive opponents. In this case, the opponent uses a single algorithm or selects randomly.

Aha et al. [2005] studies an algorithm selection approach using case-based reasoning [Xu, 1994] in Wargus, a clone of real-time strategy game Warcraft II. They assume the opponent chooses a strategy at random, which is unrealistic and potentially harmful. In our experiments, a random policy over algorithms performs poorly (see Section 5.4.1.2). In StarCraft, Preuss et al. [2013] adopt *fuzzy* rules to select a game-playing strategy, which require expert knowledge to create and adjust the rules.

Still in StarCraft, successful bots usually rely on a portfolio of strategies, commonly encoded as build-orders, and choose according to their previous performance against their opponents [Ontañón et al., 2013]. However, selection mechanisms ignores opponent's adaptation and strategic reasoning.

[4]General Video Game Playing (GVGP) brings the idea of General Game Playing (GGP) to computer games: a single algorithm must handle a multitude of computer games, fostering the development of generalist approaches. In contrast with GGP, games are real-time and possibly stochastic [Levine et al., 2013].

[5]In Goofspiel, at each round a card is revealed and two (or more) players dispute it by playing a card from their hands. Whoever throws the highest card wins the round and scores the face value of the table card.

[6]In the competitive traveling salesman problem, each salesman receives a reward for being the first to visit a city, but each travel has a cost [Fekete et al., 2004]

A game-theoretic approach for algorithm selection is studied in [Sailer et al., 2007] in a synthetic real-time strategy game. Authors use Monte Carlo simulations from the current state to fill a payoff matrix with the performance among pairs of algorithms. Nash Equilibrium is calculated and an algorithm is selected according to the resulting policy. We test reinforcement learning algorithm selection agents against this approach, which we call Monte Carlo Algorithm Selection (MCAS). MCAS outperforms our agents trained in self-play, in Section 5.4.1.1. However, our agent outperforms MCAS when training specifically against it, in Section 5.4.1.2. Moreover, our approach does not require a game's forward model to perform simulations, as we employ reinforcement learning methods.

The creation of *SparCraft*, a combat simulator for StarCraft, allowed the advancement of script-assignment search approaches. In those, each game unit controlled by the player receives a script determining a simple behavior, such as attack the closest enemy, move away from combat, or hit-and-run (also known as kiting). This is a finer-grained algorithm selection framework, where an algorithm (or script) is assigned to individual units instead of selecting a player algorithm to control all units at once. From the joint assignment of scripts, a coordinated behavior may emerge.

Portfolio Greedy Search (PGS) [Churchill and Buro, 2013] uses a hill-climbing approach to assign the scripts. This is further extended by Stratified Strategy Selection (SSS) [Lelis, 2017], which groups units and assigns scripts to the groups. Portfolio Online Evolution [Wang et al., 2016] uses a online evolutionary algorithm to try out different assignment combinations. Greedy Alpha-Beta (GAB) and Stratified Alpha-Beta (SAB) [Moraes and Lelis, 2018] build on PGS and SSS, respectively, using the notion of asymmetric action abstractions[7]. These algorithms operate in two steps. The first step fixes moves of the so-called restricted units using the underlying script-assignment algorithm (PGS or SSS). All enemy units are fixed with a specific script. The second step uses Alpha-Beta Considering Durations [Churchill et al., 2012] to determine the moves of remaining allied units. Both GAB and SAB substantially outperform the previous approaches.

The success of script-assignment approaches demonstrate the usefulness of abstracting from low-level actions, even when individual units are considered. However, the methods developed so far require the game's forward model to simulate the assignments' outcome. Our algorithm selection approaches, although being coarser by selecting an algorithm for the player rather than individual units, dismiss forward models.

[7]In the framework of action abstractions [Hawkin et al., 2011], some procedure (e.g. a heuristic or script) reduces the number of actions considered by the decision-maker.

3.5 Summary

In this chapter, we divided game-playing approaches in four groups, each one with some drawbacks:

- Adapted search approaches (Section 3.1): the ones based on hand-crafted abstract representations require expert knowledge to design such representations. Moreover, all search methods require a game's forward model;

- Rule-bases approaches (Section 3.2): those methods also require domain knowledge to construct useful rules;

- Reinforcement learning (Section 3.3): such approaches currently succeed in reflex-based scenarios, facing difficulties when temporal abstraction is required. Approaches dealing with temporal abstraction were not evaluated in scenarios with a large number of actions;

- Algorithm selection (Section 3.4): most techniques, when modeling the opponent as an algorithm selector, present it as a "dummy": either as a random selector or as a non-adaptive agent. Strategic reasoning, in the game-theoretic sense, is not considered. Sailer et al. [2007] and the unit-script approaches are exceptions, commented below.

Sailer et al. [2007] investigates game-theoretic properties of algorithm selection. One of our models also focuses on these game-theoretic properties, with the following differences: (i) we model algorithm selection as a sequential decision process, whereas Sailer et al. [2007] simulates the remainder of a match without considering possible algorithm switches; (ii) we go further by removing the assumption that the opponent is an algorithm selector.

Although unit-script approaches [Churchill and Buro, 2013; Wang et al., 2016; Lelis, 2017; Moraes and Lelis, 2018] perform algorithm selection in a finer grain (at unit level) than we do (at player level), they require forward models to evaluate the assignment's quality, whereas our approach is model-free.

Chapter 4

The strategic reasoning framework

This chapter presents our strategic reasoning framework for complex computer games. Our approach is inspired by the human game-playing behavior, which involves recalling previously trained courses of actions, or strategies, and remarkable generalization skills. We define a computational version of the term strategy as an algorithm: a method that specifies a game-playing policy - a mapping from states to actions. We also discuss the relation of algorithms and *options*, or temporally-extended actions in reinforcement learning [Sutton et al., 1999a].

We present different models of interaction, either focusing on game-theoretic aspects, assuming that the opponent is an algorithm selector, or on game-playing performance, by removing this assumption, and attempting to learn strong algorithm selection policies, although recognizing they can be exploitable in theory.

To actually implement the strategic reasoning approach in complex games, we need to handle their enormous state spaces by generalizing learned values across states. We discuss state aggregation and linear function approximation as possible ways, and their implications in our strategic reasoning framework.

We finish the chapter with a summary and a discussion of how our approach adheres to our guidelines: to be model-free, demand usual hardware and play whole matches of complex games.

4.1 The human approach

Good performance on a computer game requires specific reasoning methods to handle the enormous game complexity in terms of possible states and actions. Human players usually resort to specific "courses of actions", often referred to as strategies. In real-time strategy games, they are usually implemented as build orders and their follow-ups. A build order is a predefined recipe or script for a player to follow, specifying the sequence of buildings and units to produce on early game stages. A follow-up is the next sequence of actions to continue developing the game as it progresses.

A game is actually effected by low-level actions, but these are too many for a human player to take into account. When playing competitively, a player trains a repertoire of strategies by memorizing the sequences of actions needed to achieve each goal. This is associated with motor learning[1] and the psychology concept of *chunking*, where one groups individual pieces of information together into a meaningful whole [Verwey and Abrahamse, 2012]. When a strategy has been successfully trained, the player is able to recall it, so that the limbs perform the necessary movements from muscular memory, liberating the player's attention.

Thus, by recalling memorized strategies, human attention is free to reason at an abstract level with less choices to consider, as if the person is playing the game with a k-button controller. Each button activates a previously trained strategy, and the player assesses the environment situation and presses the most appropriate button, enabling the corresponding strategy, when necessary[2].

Humans excel at detecting patterns and associating unfamiliar situations with familiar ones [Mattson, 2014]. In game-playing, this ability allows one to handle unforeseen situations, by comparing them to previously experienced ones and implementing similar reactions. This way, the enormous state spaces of complex games are mapped to a simplified mental representation, enabling one to generalize the reaction from one situation to similar ones.

4.2 Game-playing algorithms

In general, computers play games either by game-tree search or by reinforcement learning[3] (see Section 2.3). Game-tree search requires the game's transition function, also regarded as the forward model, to simulate the action effects. However, the transition function is not available in general. In such cases, which include commercial real-time strategy games (RTS), the agent must resort to reinforcement learning.

[1]Motor learning involves improving the accuracy of movements with repetition and feedback. It is necessary for complicated movements such as speaking, playing the piano and/or computer games.

[2]This analogy is based on Ben Weber's 8-button StarCraft video: `https://youtu.be/_XLEdsEFQWE`

[3]Although model-based dynamic programming approaches [Sutton and Barto, 1998, Chapter 4] are regarded as reinforcement learning, here we use the term in the strict sense of learning without the environment model.

A reinforcement learning agent repeatedly interacts with the environment, observing the actual results of its actions, so as to maximize the obtained rewards. The agent must try each action at least once to estimate its value. Hence, domains with large action sets, such as real-time strategy games, are troublesome for reinforcement learning agents.

Humans excel in RTS games by training a repertoire of courses of actions, or game-playing strategies, and then reasoning at a strategic level, assessing which course of action better suits the current game situation, as discussed in Section 4.1. For our computational version of this behavior, we adopt a portfolio of algorithms instead of the human repertoire of strategies.

Algorithms are understood as computational procedures that receive input data and generate corresponding output data according to a well-defined sequence of steps [Cormen et al., 2001, Section 1.1]. A game-playing algorithm thus receives an environment state and outputs a valid action.

From a reinforcement learning point of view, an algorithm specifies a *policy*[4]. A (stochastic) policy, denoted by $\pi : S \times A \rightarrow [0, 1]$ maps an environment state to a probability distribution over actions. Deterministic policies are special cases that assign probability 1 to an action and 0 to the others, and can be denoted as $\pi : S \rightarrow A$. An algorithm thus mimics the human concept of strategy. The algorithm's resulting policy is a specific game-playing behavior, or course of action in human terms. We refer to the set of algorithms as the portfolio, denoted Π.

We are not concerned with internal details of algorithms, as long as they are real-time: they can implement a lookup table indexed by the state, a simple game-playing script that outputs an action based on a set of rules or a fully-featured game-playing software-controlled player (bot) employing sophisticated heuristics or machine learning approaches. Technically, algorithms with learning mechanisms define non-stationary policies, which change over time with accumulated experience. Our discussion in this Chapter assumes stationary algorithms. Non-stationary ones are discussed in the experiments of Section 5.3.

There are infinite possible stochastic policies for any given game. However, we consider only a limited amount: the ones determined by a subset of known algorithms that can operate on the game. The choices of our algorithm selectors are thus algorithms rather than low-level actions. By limiting the choices to consider, we simplify the resulting reinforcement learning problem: instead of reasoning over the enormous amount of actions, we only consider a limited set of algorithms to choose.

[4]From a game-theoretic point of view, an algorithm specifies a behavioral strategy [Nisan et al., 2007, Section 3.7].

When reasoning over algorithms in a reinforcement learning problem, the agent perceives its state, selects an algorithm, the algorithm selects an action on the agent's behalf and the environment transitions to the next state, generating a reward signal. The agent perceives the next state and the reward signal, and the cycle repeats.

Intuitively, it seems easier to discover the best choice in a limited set of algorithms than in a large set of actions. However, the algorithm selector's performance will be limited by that of the best algorithm in its portfolio. Nevertheless, an action-selector will always match or outperform an algorithm-selector in the long-run, as reinforcement learning methods have asymptotic guarantees of convergence to the optimal, reward-maximizing policy over actions [Sutton and Barto, 1998, Chapter 6]. Example 2 below illustrates this issue in a multi-armed bandit[5].

Example 2 *Learning over algorithms versus over actions.*

Consider a multi-armed bandit with four arms (actions) $A = \{a_1, a_2, a_3, a_4\}$, and two available algorithms, $\Pi = \{\pi_1, \pi_2\}$. We assume that a_1 is the optimal action, but that is not known in advance, and the algorithms' policies are the distributions shown in Table 4.1, which results from their internal reasoning procedures.

Algorithm \ Action	a_1	a_2	a_3	a_4
π_1	0.8	0.2	0	0
π_2	0.2	0	0.7	0.1

Table 4.1: Algorithms' policies (probability distribution functions) used in our simple example.

Consider two reinforcement learning agents: an action-selector P_A, and an algorithm-selector P_Π, which act via an ϵ-greedy rule with ties broken randomly over the greedy choices. In the first iteration, P_A picks a_1 with probability 0.25. P_Π picks π_1 with probability 0.5, which selects a_1 with probability 0.8. Hence, P_Π selects a_1 indirectly with probability $0.5 \cdot 0.8 = 0.4 > 0.25$. Thus P_Π's expected performance is better than P_A's in the first iteration. In the next few iterations, P_Π is even more likely to pick π_1, while P_A still needs to explore further, until finally playing enough training iterations for the action-value of a_1 be higher than the other actions.

When the number of training iterations becomes sufficiently large, P_A learns to always select a_1, and P_Π to always select π_1. However, since π_1 selects a_1 with probability 0.8, P_Π selects a_1 with probability $0.8 < 1$, and hence is outperformed by P_A in the long run. Therefore, until a certain number of training iterations, a reinforcement learning

[5]Multi-armed bandits are reviewed on [Sutton and Barto, 1998, Chapter 2].

agent may perform better by learning over algorithms, if the algorithms' policies are strong enough. In the long run, however, learning over actions will always perform at least as good as learning than over algorithms.

In Example 2, the algorithm selector fares better in the short term, but is eventually outperformed in the long run by the action selector. We further illustrate this phenomenon with experiments in synthetic multi-armed bandits (see Section 5.1).

Our notion of algorithms is closely related to the concept of *options* in Markov Decision Processes [Sutton et al., 1999a]. Options are temporally-extended actions that a reinforcement learning agent may execute to pursue specific goals, as formalized in Definition 2.

Definition 2 *An option is a tuple $\langle I, \pi, \beta \rangle$, where:*

- *$I \subseteq S$, where S is the set of environment states, is the initiation set: an option is available in state s if and only if $s \in I$;*
- *π is the option's policy. If the option is active in state s, actions are selected according to the probability distribution dictated by $\pi(s, \cdot)$;*
- *$\beta : S \to [0,1]$ is the termination condition. It indicates the probability of an option terminating in a given state.*

Our algorithms are one-step options: they may initiate in any state ($I = S$), act according to their internal policies and terminate after one transition ($\beta(s) = 1 \; \forall s \in S$). We use the terms interchangeably throughout this text, although we prefer to mention options on contexts where the temporally-extended execution is clear.

4.3 Strategic reasoning approaches

This section presents formal models for computers to learn over algorithms to play games with huge action spaces. The first model we present allows two algorithm-selection agents to play against each other. The second model pitches an algorithm-selection vs an action-selection agent. Both models consider the opponent's presence, by accounting for its possible choices. We discuss the situation where the agent attempts to find a strong algorithm selection policy by ignoring the possible opponent's actions. Each model has different degrees of theoretical guarantees and practical performance issues for the agent.

Although the presented models tackle the problem of the action state size by reasoning over algorithms, the state space remains an issue. Thus, we also discuss possible approaches to generalize learning across states and their implications with our models.

4.3.1 Algorithms versus algorithms

This model assumes that both players are algorithm selectors and their portfolios are of common knowledge. Under these assumptions, solving the resulting game results in theoretically-guaranteed performance.

The formal model is a two-player zero-sum Markov Game over algorithms, denoted MG_Π. It replaces A by Π with proper adaptations, as shown in Definition 3.

Definition 3 *A finite, two-player, zero-sum Markov Game over algorithms is a tuple* MG_Π= (MG,Π,T_Π), *where:*

- $MG = (N, S, A, R, T)$ *is the underlying Markov Game (see Definition 1 in Section 2.2);*

- $\Pi = \Pi_1 \times \Pi_2$ *is the collection of both player's portfolios, where* Π_i *is player i's portfolio, or set of algorithms;*

- $T_\Pi : S \times \Pi_1 \times \Pi_2 \times S \rightarrow [0,1]$, *is an indirect transition function, where* $T_\Pi(s, \pi_1, \pi_2, s')$ *denotes the probability of reaching state* s' *when player 1 chooses algorithm* π_1 *and player 2 chooses* π_2 *in state* s.

The indirect transition function embeds the algorithms' policies and the direct transition function of the game engine $T : S \times A_1 \times A_2 \times S \rightarrow [0,1]$. During gameplay, T_Π obtains the actions a_1 and a_2 selected by both players' algorithms and transitions to the next state according to $T(s, a_1, a_2, \cdot)$, implemented by the game engine.

The minimax-Q method (see Section 2.3.2) can be used to learn the maximin algorithm selection policy. With much fewer choices to consider than if the agent were reasoning over actions, the method can achieve a better performance faster. Game-tree search approaches (see Section 2.3.1) could also be used to derive algorithm selection policies, but they would require the underlying game's transition function to simulate the effects of actions selected by the algorithms.

4.3.2 Algorithms vs actions

The model of Markov Game over algorithms, discussed in Section 4.3.1 assumes that players' portfolios are of common knowledge. In general, however, the assumption does not hold neither in human nor computer gameplay, as humans may come up with unforeseen courses of actions and a programmer can incorporate a novel algorithm in its program's portfolio to surprise the opponent. In this situation, the algorithm selector agent can consider that the opponent can select any action in the underlying game. The model of two-player, zero-sum Markov Game of algorithms versus actions, denoted $MG_{\Pi\times A}$ is then shown on Definition 4.

Definition 4 *Without loss of generality, let's assume that player 1 is the algorithm selector and player 2 is the action selector. A two-player, zero-sum Markov Game of algorithms versus actions is a tuple* $MG_{\Pi\times A}$*=* $(MG, \Pi_1, T_{\Pi\times A})$*, where:*

- MG *=* (N,S,A,R,T) *is the underlying Markov game (see Definition 1 in Section 2.2);*
- Π_1 *is player* 1*'s algorithm portfolio;*
- $T_{\Pi\times A} : S \times \Pi_1 \times A_2 \times S \rightarrow [0, 1]$*, is an indirect transition function, where* $T_{\Pi\times A}(s, \pi_1, a_2, s')$ *denotes the probability of reaching state* s' *when player 1 chooses algorithm* $\pi_1 \in \Pi_1$ *and player 2 chooses action* $a_2 \in A_2$ *in state* s*.*

The indirect transition function embeds the policies of player 1's algorithms and the direct transition function of the game engine $T : S \times A_1 \times A_2 \times S \rightarrow [0, 1]$. During gameplay, $T_{\Pi\times A}$ obtains the action a_1 of player 1's selected algorithm, and transitions to the next state according to $T(s, a_1, a_2, \cdot)$, implemented by the game engine.

Game-tree search or reinforcement learning methods such as the minimax-Q can also be used to solve the $MG_{\Pi\times A}$ to determine a safe algorithm selection policy for the player. However, in practice, $|A_2|$ (the number of low-level actions that player 2 can perform) is large such that computing such solution is unfeasible.

An alternative for the algorithm-selection player is to ignore the opponent's actions. From the point of view of this opponent-oblivious agent, the $MG_{\Pi\times A}$ is reduced to a Markov Decision Process over algorithms with an embedded opponent, denoted MDP_Π, formalized in Definition 5.

Definition 5 *A Markov Decision Process over algorithms, is a tuple* MDP_Π*= (*MG*,* Π*,* T'_Π*), where:*

- $MG = (N, S, A, R, T)$ *is the underlying Markov Game (see Definition 1 in Section 2.2);*

- Π_1 *is player 1's portfolio, or set of algorithms.*

- $T'_\Pi : S \times \Pi_1 \times S \to [0, 1]$*, is an indirect transition function affected by the opponent.* $T'_\Pi(s, \pi_1, s')$ *denotes the probability of reaching state* s' *when the agent chooses algorithm* π_1 *in state* s*. This probability depends on the hidden opponent action.*

The transition function T'_Π is actually the $MG_{\Pi\times A}$'s transition function $T_{\Pi\times A}$ with the opponent action hidden. The opponent action depends on its policy over actions. A stationary opponent's policy determines a stationary T'_Π so that the guarantees of convergence for single-agent reinforcement learning methods, such as Q-learning, hold.

In general, however, opponents can adapt and change their policies, making the resulting transition function non-stationary for the opponent-oblivious agent. Thus, such agents are vulnerable to a devious trick, where a training opponent induces the player to learn a weak policy, to exploit it afterwards [Littman, 1994]. Nevertheless, many opponent-oblivious approaches have succeeded in the past [Tesauro, 1995] and recent [Silver et al., 2016] times. The devious trick is avoided either by training via self-play, and/or by expert game analysis. The resulting policies can be strong, being able to match or outperform the best human players in Backgammon [Tesauro, 1995], or Go [Silver et al., 2016], respectively. These approaches succeed by effective learning generalization. This involves finding efficient representations for the state space, an issue we neglected so far, but tackle in the next section.

4.4 Issues with the state space

Our strategic reasoning framework successfully reduces the number of choices to consider in a given state of a complex game, inspired by the human approach of relying on previously trained strategies. However, large state spaces are also a problem and the different approaches to tackle this issue have implications with our framework. This section discusses the approaches we adopt and their implications.

Intuitively, those approaches are also inspired by human behavior: a person tends to relate the current, possibly novel, game situation to previously experienced ones, potentially implementing similar reactions.

4.4.1 State aggregation

State aggregation is a form of function approximation in which the set S of primitive states is partitioned into a set $\overline{S}$ of clusters, or abstract states. A state abstraction function $\phi : S \to \overline{S}$ maps a state in the primitive state space to an abstract state[6]. Action-values from states in the same cluster are shared and learning is thus generalized: tabular reinforcement learning methods store an entry for each cluster-action pair rather than each state-action pair. Usually, the state abstraction function is such that $|\overline{S}| \ll |S|$ and the resulting problem becomes manageable by tabular reinforcement learning approaches.

Our algorithms can be seen as one-step options, as discussed in Section 4.2. However, combined with state aggregation, we have multi-step options, as we discuss next. We remark that an option is a tuple $\langle I, \pi, \beta \rangle$, where $I \subseteq S$ is the initiation set, indicating where the option can be activated; π is the option's policy, mapping states to probability distributions over actions; and $\beta : S \to [0, 1]$ is the termination condition, indicating the probability of the option terminating in a state (see Definition 2).

Definition 6 *Multi-step options via abstract states and algorithms.*

Given the set of abstract states $\overline{S}$, the corresponding state abstraction function ϕ and a portfolio of algorithms Π, for each abstract state $\overline{s} \in \overline{S}$ and algorithm's policy $\pi \in \Pi$, there is an option $o = \langle I, \pi, \beta \rangle$, with elements defined as follows:

- $I = \{s \in S : \phi(s) = \overline{s}\}$;
- π : *the option's policy is exactly the algorithm's policy;*
- $\beta(s) = 0$ *if* $s \in I$ *and* $\beta(s) = 1$ *otherwise.*

For each abstract state $\overline{s}$, there are $|\Pi|$ available options. The agent then selects an option whose initiation set is $\overline{s}$, the option performs actions according to the corresponding policy, passing through various primitive states inside $\overline{s}$, and terminates by reaching a primitive state s' that maps to a new abstract state: $\phi(s') = \overline{s'} \neq \overline{s}$.

The original definition of Sutton et al. [1999a] allows the inclusion of primitive actions $a \in A$ in the set of options. However, we do not include those as the action sets are very large.

We can extend the models discussed in Section 4.4.1 to include our options scheme. Let O be the set of options, as per Definition 6. The Markov Game over

[6]We follow the notation of Li et al. [2006] to represent the set of abstract states and the abstraction function.

algorithms (Definition 3), then becomes a Stochastic game over options, denoted SG_O, shown in Definition 7. We do not refer to this model as a Markov Game to remark that it does not have the Markov property, as we discuss next.

Definition 7 *A two-player zero-sum Stochastic game over options is a tuple $SG_O = (MG, \overline{S}, \phi, O)$, where:*

- *$MG = (N, S, A, R, T)$ is the underlying Markov game (see Definition 1 in Section 2.2);*
- *$\overline{S}$ is the set of abstract states, which partitions the primitive set of states S. That is, for all $\overline{s_i} \in \overline{S}$, the following three conditions hold: (i) $\overline{s_i} \subseteq S$; (ii) $\overline{s_i} \neq \overline{s_j} \implies \overline{s_i} \cap \overline{s_j} = \emptyset$; (iii) $\bigcup_{i=1}^{|\overline{S}|} \overline{s_i} = S$.*
- *$\phi : S \to \overline{S}$ is the state abstraction function;*
- *$O = O_1 \times O_2$ is the collection of both player's options, where O_i is player i's options, as per Definition 6;*
- *$T_O : \overline{S} \times O_1 \times O_1 \times \overline{S} \to [0, 1]$, is an indirect transition function, where $T_O(\overline{s}, o_1, o_2, \overline{s'})$ denotes the probability of reaching abstract state $\overline{s'}$ when player 1 chooses option o_1 and player 2 chooses o_2 in abstract state $\overline{s}$.*

The SG_O is formalized in Algorithm 2, which shows an episode, or match, in this model.

Lines 7–11 are the execution of $T_O(\overline{s}, o_1, o_2, \cdot)$. It consists of executing both players' options until a different abstract state is found. The condition $s \in I$ is equivalent to the option termination β of an option $\langle I, \pi, \beta \rangle$, since an option terminates when it reaches a different abstract state. To ensure the real-time nature of the game, an additional no-op option is executed by default when a player fails to return a valid option in line 6.

In line 13 we present the experience tuple to the agents. It contains only the reward obtained in the last transition, whereas in Sutton et al. [1999a]'s original definition, agents accumulate all rewards received in the trajectory determined by the selected options. This modification suits the idea that the information received upon reaching a different abstract state is more important, because primitive states in the same cluster are similar. Nevertheless, the algorithm can be easily changed to accommodate the original definition, by accumulating the rewards received during the options' execution (lines 7–11).

Algorithm 2 Stochastic game over Options

```
1: Input: the underlying Markov game ($N$,$S$,$A$,$R$,$T$), the state abstraction function $\phi$, and both players' portfolio of algorithms $\Pi_1$ and $\Pi_2$
2: Construct the set of multi-step options $O_1$ and $O_2$ according to Definition 6.
3: $s \leftarrow$ initial state in $S$.
4: $\overline{s} \leftarrow \phi(s)$
5: while s is not terminal do
6:     Present $\overline{s}$ to the players, which respond with their options $o_1 = \langle I, \pi_1, \beta \rangle \in O_1$ and $o_2 = \langle I, \pi_2, \beta \rangle \in O_2$, respectively
7:     while $s \in I$ do                    ▷ Execution of $T_O(\overline{s}, o_1, o_2, \cdot)$
8:         Obtain actions $a_1$ and $a_2$ according to $\pi_1(s, \cdot)$ and $\pi_2(s, \cdot)$
9:         Obtain $s'$ from $T(s, a_1, a_2, \cdot)$;
10:        $s \leftarrow s'$
11:    end while
12:    $\overline{s'} \leftarrow \phi(s')$
13:    Present the experience tuple $\langle \overline{s}, o_1, o_2, R_i(\overline{s'}), \overline{s'} \rangle$ to each player $i$
14:    $\overline{s} \leftarrow \overline{s'}$
15: end while
```

Algorithm 2 highlights that players' options might differ in their algorithms policies, but have the same initiation sets and termination conditions. This means that both players adopt the same state abstraction function ϕ. This aspect can be generalized to different state abstraction schemes by presenting the experience tuple to the player whose option terminated and allowing it to select a new option, while the other player's option remains active.

For most state abstraction functions, the SG_0 model does not have the Markov property. This is because $T_O(\overline{s}, \pi_1, \pi_2, \cdot)$ works differently depending on which primitive state s was mapped to $\overline{s}$, and this information is hidden from the agents. Figure 4.1 shows an example: two different abstract states, $\overline{s_j}$ and $\overline{s_k}$ can be reached from $\overline{s_i}$ when agents jointly select options o_1 and o_k. The reached state depends on which primitive state the game is actually in, which depends on the previous choices made by the players.

Tackling the lack of Markov property can be pursued with ideas from Whitehead and Lin [1995], but this is out of this dissertation' scope. Our reinforcement learning agents act as if the SG_O is markovian, although we're aware that optimal policies over options might not be attainable.

Nevertheless, an optimal policy over options does not necessarily mean optimal behavior in the underlying game. Intuitively, this happens because agents are limited by the options' policies, as they perform the primitive actions. We investigate this issue further in a single-state (multi-armed bandit) scenario in Section 5.1. Furthermore,

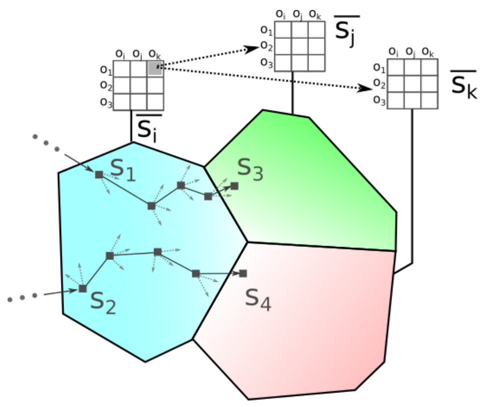


Figure 4.1: The Stochastic game over options is non-markovian. The selected options determine trajectories in the underlying game's state space. Selecting the same options in abstract state $\overline{s_i}$ ends in $\overline{s_j}$ or $\overline{s_k}$ depending on which primitive state the agents are. Dashed arrows indicate other possible, albeit non-followed, trajectories due to options' stochastic policies.

Theorem 1 of Sutton et al. [1999a] proves this claim for options in single-agent Markov Decision Processes. However, in complex domains such as real-time strategy games, sacrificing optimality in exchange for feasibility is a reasonable choice.

Agents playing the Stochastic game over options can resort to traditional tabular reinforcement learning approaches, given that the set of abstract states and the number of choices are now manageable.

A stochastic game over options SG_O can be played with minimax-Q [Littman, 1994], given that both agents are option-selectors. In our framework, a minimax-Q agent stores an joint-option-value function $Q : \overline{S} \times O_1 \times O_2 \to \mathbb{R}$, where each $Q(\overline{s}, o_1, o_2)$ stores the value, that is, the expected sum of discounted rewards, of the agent selecting option $o_1 \in O_1$ whereas the opponent selects option $o_2 \in O_2$ in state s, and following the optimal policy over options thereafter. Player i's policy over options $\mu_i : \overline{S} \times O \to [0, 1]$ is a mapping from an abstract state to a probability distribution over options. Each game match is an instance of Algorithm 2, where a player i can select its options (line 6) via an ϵ-greedy rule: select a random option with probability ϵ, or according to the safe policy, given by Eq. 4.1, otherwise.

$$\mu_i(\overline{s}, \cdot) = \underset{\mu(s,\cdot)}{\text{argmax}} \min_{o_2 \in O_2} \sum_{o_1 \in O_1} \mu(\overline{s}, o_1) Q(\overline{s}, o_1, o_2) \tag{4.1}$$

When receiving an experience tuple $\langle \overline{s}, o_1, o_2, r, \overline{s'} \rangle$, player i updates its option-value Q via Eq. 4.2.

$$\begin{array}{lcl} Q(\overline{s}, o_1, o_2) & \leftarrow & Q(\overline{s}, o_1, o_2) + \alpha \left[r + \gamma V(\overline{s'}) - Q(\overline{s}, o_1, o_2)\right] \\ V(\overline{s'}) & = & \min_{o'_2 \in O_2} \sum_{o'_1 \in O_1} \mu_i(\overline{s}, o'_1) Q(\overline{s'}, o'_1, o'_2) \end{array} \tag{4.2}$$

In summary, a minimax-Q agent reasoning over options is the same worst-case reward maximizer of Section 2.3.2, with actions replaced by options, and states by abstract states.

In general, however, the opponent will not be an option-selector, or will not select options from a known portfolio. That is, the opponent can virtually select any low-level game action. However, as remarked in Example 2, it is impractical to consider all opponent actions and a common practice is to use an opponent-oblivious Q-learning agent, which, in our case, would store an option-value function $Q : \overline{S} \times O_1 \to \mathbb{R}$. Each $Q(\overline{s}, o_1)$ stores the value, that is, the expected sum of discounted rewards, of the agent selecting option $o_1 \in O_1$ in state $\overline{s}$, and following the optimal policy over options thereafter. Player i's policy over options $\mu_i : \overline{S} \times O \to [0, 1]$ is a mapping from an abstract state to a probability distribution over options. Each game match is an instance of Algorithm 2. Let's assume that player 1 is our opponent-oblivious Q-learning agent. This agent can select an option (line 6) via an ϵ-greedy rule: select a random option with probability ϵ, or according to the greedy policy, given by Eq. 4.3, otherwise.

$$\mu_i(\overline{s}, \cdot) = \operatorname*{argmax}_{o \in O_1} Q(\overline{s}, o) \tag{4.3}$$

When receiving an experience tuple $\langle \overline{s}, o_1, o_2, r, \overline{s'} \rangle$, player 1 ignores the opponent components and updates its option-value Q via Eq. 4.4.

$$\begin{array}{lcl} Q(\overline{s}, o_1) & \leftarrow & Q(\overline{s}, o_1) + \alpha \left[r + \gamma V(\overline{s}) - Q(\overline{s}, o_1)\right] \\ V(\overline{s'}) & = & \max_{o'_1 \in O_1} Q(\overline{s'}, o_1) \end{array} \tag{4.4}$$

In summary, an opponent-oblivious Q-learning agent reasoning over options is the same optimistic reward-maximizer Q-learning agent [Watkins and Dayan, 1992], with actions replaced by options, and states by abstract states. The same caveats discussed in Example 2 apply: such agent is vulnerable to exploitation, although it can still learn strong policies via self-play or expert game analysis.

The state abstraction function ϕ directly interferes on the attainable performance and on the training time required to reach it: finer abstraction functions, that is, where more abstract states are generated, result in optimal policies closer to those of the full-state Markov Game over algorithms (MG_Π in Section 4.3.1). An extreme case is a one-to-one correspondence, where $\overline{S} = S$. An optimal policy in our stochastic game over options corresponds exactly to the optimal policy over algorithms of the MG_Π.

On the other hand, agents will need more training time to visit the higher number of abstract state-option pairs to produce precise estimates, so that coarser abstractions are preferred in this sense. An extreme case is shown in Example 3.

Example 3 *An extremely coarse state abstraction scheme.*

Let $\overline{S} = \{\overline{s_0}, \overline{s_t}\}$. *Let* ϕ *be such that, for each* $s \in S$, $\phi(s) = \overline{s_0}$ *if* s *is non-terminal and* $\phi(s) = \overline{s_t}$, *otherwise.*

In such abstraction scheme, agents have a one-shot interaction with the environment: each agent chooses an option when the game begins and their options act until the game finishes, as the next abstract state corresponds to a terminal state in the underlying game.

The aggregation scheme of Example 3 induces a normal-form game, henceforth referred to as the game of algorithm selection, defined as follows:

Definition 8 *The game of algorithm selection is a two-player zero-sum normal-form game, defined by a tuple* $(MG, \Pi, \mathcal{R})$, *where:*

- $MG = (N, S, A, R, T)$ *is the underlying Markov Game (see Definition 1 in Section 2.2);*
- $\Pi = \Pi_1 \times \Pi_2$ *is the collection of both players' portfolio of algorithms;*
- $\mathcal{R} : \Pi_1 \times \Pi_2 \to \mathbb{R}^2$ *is the payoff matrix.* $\mathcal{R}_i(\pi_1, \pi_2)$ *indicates player* i*'s payoff when player 1 selects algorithm* π_1 *and player 2 selects* π_2. *It is the expected reward of* π_1 *versus* π_2 *playing an entire match in the underlying game. Let* $Z \subset S$ *be the set of underlying game's terminal states:* $\mathcal{R}_i(\pi_1, \pi_2) = \sum_{s' \in Z} R_i(s') \cdot Pr(s'|\pi_1, \pi_2)$, *where* $Pr(s'|\pi_1, \pi_2)$ *is the probability of reaching terminal state s' given that* π_1 *played against* π_2.

The probability of reaching each terminal state $s' \in Z$, $Pr(s'|\pi_1, \pi_2)$ *is influenced by the possible trajectories determined by the policies in the underlying game.*

The game of algorithm selection is the stochastic game over options (SG_O in Definition 7) with the state abstraction scheme defined as in Example 3. The game of algorithm selection allows us to draw important conclusions on how algorithms interact in a given underlying game. Moreover, Nash Equilibrium over the game of algorithm selection specifies a safe policy over algorithms: a policy with guaranteed performance over a sequence of matches. Experiments over these aspects are discussed in Section

5.3, alongside approaches to handle non-stationary algorithms, which specify different policies as they learn from experience.

On the other hand, performance guarantees on the game of algorithm selection do not hold on the much more complex underlying game. Richer behaviors can arise when players can switch algorithms during a match, which is allowed by the more general, multiple-decision-point model of stochastic game over options. Experiments in this richer model are performed in Section 5.4.1.1 for an opponent-aware agent and in Section 5.4.1.2 for an opponent-oblivious agent.

4.4.2 Linear function approximation

State aggregation, discussed in Section 4.4.1 is a specific form of function approximation, which generalizes learning by sharing the values in primitive states belonging to the same abstract state. State aggregation defines a sharp frontier where possibly similar states will not share values if they belong to different abstract states. Moreover, the state abstraction scheme must be carefully designed so that the number of resulting abstract states is not too high to require too many training episodes nor too low so that the resulting policies perform poorly on the underlying game.

This section discusses linear function approximation as an approach to handle the state space complexity. It allows a smoother learning generalization than state aggregation: similar states tend to have similar values as there is no sharp frontier to divide the states. The similarity depends only on the values assumed by the identified features used to describe the state.

Here we discuss linear function approximation on the Markov Decision Process over algorithms (MDP_Π in Definition 5). To briefly recall, this model assumes an opponent-oblivious agent, which observes the primitive state $s \in S$, selects an algorithm $\pi \in \Pi$, and the algorithm outputs an action $a_1 \in A$. The transition function processes a_1 alongside the opponent's hidden action $a_2 \in A$ and generates the next state $s' \in S$ according to the probability $T(s, a_1, a_2, s')$ specified by the game engine. The agent then observes s' and the received reward $R(s')$ for reaching it, repeating the decision cycle.

With linear function approximation, the state is described by a feature vector with n features: $\mathbf{f}(s) = \langle f_1(s), \ldots, f_n(s)\rangle$. Each algorithm $\pi \in \Pi$ has an associated weight vector $\mathbf{w}^\pi = \langle w_1^\pi, \ldots, w_n^\pi\rangle$. Then, instead of storing the algorithm value associated with every state, $Q(s, \pi)$, in tabular form, we approximate it with $\tilde{Q}(s, \pi, \mathbf{w})$ via Eq.

4.5, where $\mathbf{w} = \bigcup_{\pi \in \Pi}\{\mathbf{w}^\pi\}$ is the set of all weight vectors.

$$\tilde{Q}(s, \pi, \mathbf{w}) = \mathbf{f}(s) \cdot \mathbf{w}^\pi = \sum_{i=1}^{n} f_i(s) \cdot w_i^\pi \tag{4.5}$$

Hence, we store $|w| = |f|$ instead of $|S|$ values for each algorithm. Usually, $|f| \ll |S|$, resulting in a compact representation. Moreover, the value of an algorithm π for a given state s, $\tilde{Q}(s, \pi, \mathbf{w})$ is propagated to other states, according to their similarity with s regarding their feature values. Thus, algorithms in unvisited states can also have precise estimates of $\tilde{Q}$, if similar states have been visited.

As before, a policy over algorithms $\mu : S \times \Pi \to [0, 1]$ maps a state to a probability distribution over algorithms. For a given state s, an ϵ-greedy policy selects a random algorithm with probability ϵ, or the best known algorithm, i.e. $\text{argmax}_{\pi \in \Pi} \tilde{Q}(s, \pi, \mathbf{w})$, otherwise.

When the agent interacts with the environment, selecting an algorithm π in state s, reaching state s' and receiving reward $R(s')$, it receives a sample of $Q(s, \pi)$, which is the target of $\tilde{Q}(s, \pi, \mathbf{w})$. This sample is $R(s') + \gamma\tilde{Q}(s', \pi', \mathbf{w})$, where π' is some algorithm chosen in s'. On-policy control methods such as Sarsa use π' as the one actually returned by $\mu(s', \cdot)$. Weights are thus updated to reduce the prediction error, δ. Equation 4.6 shows the update rule of Sarsa(0), a specific version of Sarsa(λ) [Rummery and Niranjan, 1994], which does not use eligibility traces[7].

$$\begin{array}{rcl} \delta & = & R(s') + \gamma\tilde{Q}(s', \pi', \mathbf{w}) - \tilde{Q}(s, \pi, \mathbf{w}) \\ w_i^\pi & \leftarrow & w_i^\pi + \alpha\delta f_i(s) \end{array} \tag{4.6}$$

Our state aggregation model forces the agent to stick with an algorithm until the primitive state maps to a new abstract state, whereas our linear function approximation model does not. In other words, the agent can select a new algorithm in every game state. However, as algorithm-values are generalized across similar states, it is likely that the best algorithm in a given state remains the best in its neighbors, so that the agent can maintain an algorithm executing for extended periods.

4.5 Summary

This chapter discussed a strategic reasoning framework that attempts to mimic the human approach on complex games. As human players usually rely on previously

[7]Eligibility traces allow updating the value not only of the current, but of past choices as well with the most recent reward.

trained courses of actions, or strategies, a computer could rely on off-the-shelf game-playing algorithms. We analyzed how our concept of algorithms relates to options, or temporally-extended actions, in reinforcement learning.

We presented formal models of interaction when both players are algorithm selectors, focusing on game-theoretic aspects, and a more general one where the opponent can virtually select any action, where we focus on underlying game performance. In the latter case, we argued that ignoring opponent's actions can be necessary, as it is unfeasible to evaluate how an algorithm performs against each possible opponent's action. We are aware, however, that the resulting policies of training an opponent-oblivious agent have no game-theoretic guarantees and could be exploited. Nevertheless, the stronger the resulting policy, the harder it would be to exploit it.

Algorithms reduce the number of choices to consider in complex domains, but oftentimes huge state spaces are also an issue. We proceeded by discussing two possible approaches to handle the state space: state-aggregation, where similar states are grouped into clusters, and linear function approximation, where state values are propagated to similar ones, by representing these states using features. We showed that algorithm selection with state aggregation results in multi-step options, whereas function approximation virtually results in one-step options. We further discussed an extreme case of state aggregation, with a single decision point, and framed it as a normal-form game: the game of algorithm selection, which may aid in an initial analysis of interactions among algorithms and game-theoretical implications in a complex underlying game.

In summary, our models focus either on game-theoretic aspects or attainable performance in the underlying game: the opponent-aware models focus on game-theoretic aspects, although restricting the types of opponents considered. In other words, the resulting policies might perform poorly against opponents that do not behave as assumed. The opponent-oblivious models focus on performance, disregarding the possible opponent types.

Our algorithm selection framework satisfies the guidelines outlined in Section 1.1:

(G1) An agent must play without resorting to the game's forward model: we model algorithm selection within a reinforcement learning framework, such that model-free approaches can be used;

(G2) The approach must not depend on powerful hardware: the hierarchical reasoning provided by our algorithm selection framework aims to avoid reasoning at raw state and action representations, which would require long training sessions on distributed systems to reach a reasonable performance. Moreover, state aggregation and linear function approximation are computationally fast as they do not involve convolutions often required by deep learning approaches operating on raw state/action representations [LeCun et al., 1998]. This guideline is further verified in our experiments (Chapter 5);

(G3) The approach must handle all aspects of the underlying game: as we are not concerned with implementation details of algorithms, we can use game-playing programs or scripts, which are typically able to play entire matches of complex games. This guideline is further verified in our experiments (Chapter 5) as well.

Chapter 5

Experiments

This chapter presents experiments with our strategic reasoning framework for complex computer games. We start by illustrating how one benefits by learning over algorithms rather than over low-level actions via synthetic experiments in a multi-armed bandit (Section 5.1). We then describe StarCraft and μRTS, the real-time strategy games used as testbeds for our strategic reasoning framework (Section 5.2).

We instantiate the game of algorithm selection, a one-shot algorithm selection problem, in Section 5.3, using StarCraft as our underlying game. We illustrate game-theoretic properties in a stationary scenario, approaches to track non-stationarity and results of tournament participations of a functional StarCraft bot.

We analyze our framework in scenarios with multiple decision points in Section 5.4, using μRTS as our underlying game. We evaluate two approaches to generalize learning across μRTS states: state aggregation and linear function approximation.

State aggregation (Section 5.4.1), allows us to instantiate the Stochastic Game over Options (defined in Section 4.3.1) and investigate a multiagent reinforcement learning method, for seemingly the first time in real-time strategy games. We also evaluate a Q-learning agent reasoning over options for the more general case, when an opponent is not an algorithm selector. Our Q-learning agent achieves competitive performance against state-of-the-art search-based approaches, but is unable to consistently outperform them.

Linear function approximation (Section 5.4.2) promotes a smoother learning generalization than state aggregation. Our Sarsa agent using this method consistently outperforms the search approaches.

Section 5.5 closes the chapter, analyzing overall aspects of all experiments.

5.1 Synthetic experiments

When presenting our model of algorithms (Section 4.2), we argued that, intuitively, it is easier to find the best choice in a reduced set of algorithms than in a large set of actions, but the performance of the algorithm selector is restricted to that of its best algorithm. In other words, an algorithm selector will eventually be surpassed by an action selector. Experiments in this section illustrate this situation in a simplified scenario: a single-agent, single-state problem: a multi-armed bandit [Sutton and Barto, 1998, Chapter 2]. We generate synthetic multi-armed bandits and portfolios of algorithms, varying the number of actions (arms), the algorithms' strength (the likelihood of selecting the best action) and the number of algorithms (the portfolio size). We denote the set of multi-armed bandit actions as A and the portfolio of algorithms as Π.

We compare an action-selection reinforcement learning agent, denoted P_A, and an algorithm-selection reinforcement learning agent, denoted P_Π. We use the term *choice* when referring to either algorithms or actions. Whether the choice is an algorithm or action depends on the type of agent under consideration. Both P_Π and P_A are Q-learning [Watkins and Dayan, 1992] agents with ϵ-greedy choice selection. For both, we considered α and ϵ starting as 1, and decaying with rate 0.999 after each trial. Choice-values (Q) are initialized with zeros.

We denote an algorithm by the policy π it specifies, which is a probability distribution over actions ($\pi : A \rightarrow [0, 1]$). We artificially generate algorithms to compose the portfolio of the algorithm selector as follows: all generated algorithms prune 95% of non-optimal actions, by assigning probability 0 to them. The optimal action, a^*, is assigned a certain probability $\pi(a^*)$ and the remaining actions' probabilities are uniformly selected from $1 - \pi(a^*)$ and normalized. We generate $\pi(a^*)$ according to Gaussian and Uniform distributions:

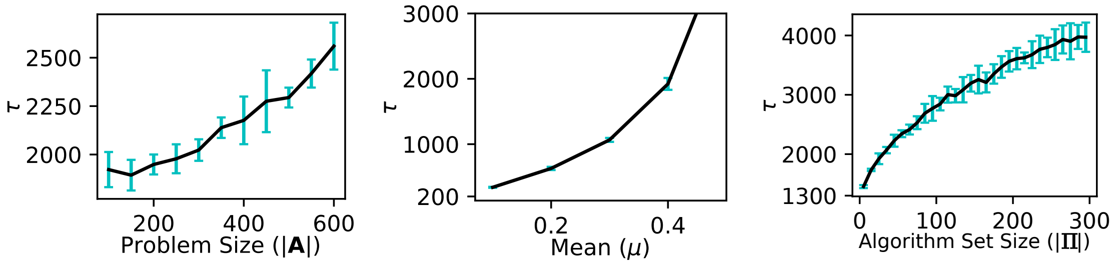
1. Gaussian: $\pi(a^*)$ is drawn from $N(\mu, \sigma)$ and truncated to the interval $[0, 1]$, where μ is the mean (a parameter we vary) and σ is fixed as 0.2;

2. Uniform: $\pi(a^*)$ is drawn from $U(0, u)$, where u is a upper-bound (a parameter we vary).

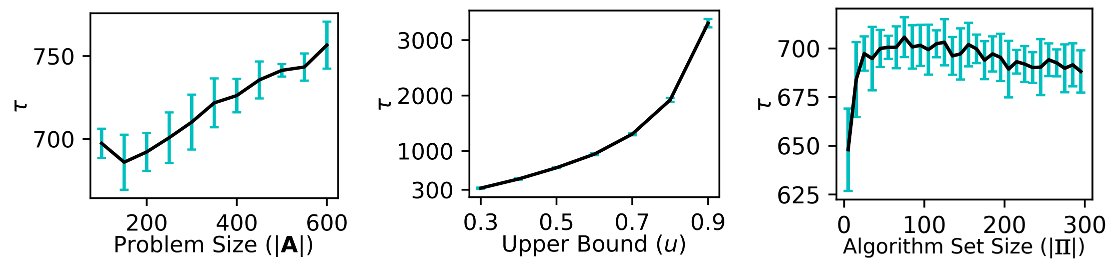
The portfolio strength is thus related to μ and u, as higher values will generate stronger algorithms.

Agents interact with a multi-armed bandit in 10000 trials, or iterations. We measure the meeting points, that is, the number of iterations needed for the cumulative reward of the action-selector P_A to meet that of the algorithm selector P_Π, and for the probability of selecting the best action by P_A to meet[1] that of P_Π according to the number of actions ($|A|$), the algorithms' strength (μ or u) and the portfolio size ($|\Pi|$). When changing one parameter, we fix the others as $|A| = 100$, $\mu = 0.4$, $u = 0.5$ and $|\Pi| = 25$. Each parameter combination is simulated in 1000 experiments. In each experiment, the multi-armed bandit is created with mean reward r_a sampled from $N(0, 1)$ for each action $a \in A$. When selecting a, the actual reward is sampled from $N(r_a, 0.25)$.

Figure 5.1 shows the meeting point (τ), where the cumulative reward of P_A meets that of P_Π as $|A|$, u or μ, and $|\Pi|$ vary. The figure shows average results of 5 repetitions of the whole procedure, and error bars show the 95% confidence interval.



(a) Gaussian



(b) Uniform

Figure 5.1: Meeting point (τ) of cumulative rewards regarding the number of actions ($|A|$), algorithms strength (μ or u), and portfolio size ($|\Pi|$). $\pi(a^*)$ of algorithms in (a) are generated with the Gaussian process, whereas $\pi(a^*)$ of those in (b) are generated with the Uniform process. A point beyond the y axis means that $\tau > 10000$ iterations.

The meeting point τ grows with $|A|$ because the action-selection agent must explore a larger action space to discover the best action. The algorithms' strength, determined by μ or u, does not affect how long P_A takes to find the best action. However, stronger algorithms make P_Π accrue more rewards whereas P_A is still exploring to discover the best action.

[1]We assume that the algorithm selector initially performs better than the action selector, which is usually the case (see Example 2 in Section 4.2).

The seemingly surprising result is τ growing with the portfolio size $|\Pi|$ in Gaussian-generated algorithms (Fig. 5.1a), as P_Π should take more time to find the best algorithm. However, although it might take time to find the best algorithm, the sub-optimal algorithms might be strong enough, as their chance of selecting the best action is always non-zero in our experiments. For the Uniform-generated algorithms (Fig. 5.1b), τ initially grows, but tends to decrease after $|\Pi| > 100$, which does not happen with the Gaussian-generated algorithms, because the Gaussian generation process with $\pi(a^*) \sim N(0.4, 0.2)$ can generate stronger algorithms than the Uniform process with $\pi(a^*) \sim U(0, 0.5)$.

We now compare the meeting point of P_A and P_Π regarding the probability of selecting the best action (a^*). The algorithm-selection agent, P_Π, selects a^* with probability $p^\Pi_{a^*}$, given by Equation 5.1, where ϵ is the probability of exploration, that is, selecting a random algorithm, and Q is the algorithm-value function. $Q(\pi)$ denotes the expected reward of selecting algorithm $\pi \in \Pi$. Equation 5.1 is the ϵ-greedy policy over algorithms weighted by the probability of the algorithms select a^*: the left component is the greedy choice, where $\pi^+ = \operatorname{argmax}_{\pi\in\Pi} Q(\pi)$ and the right component is the random choice. The probability of P_Π selecting a^*, therefore, depends on the probability of its algorithms selecting a^*.

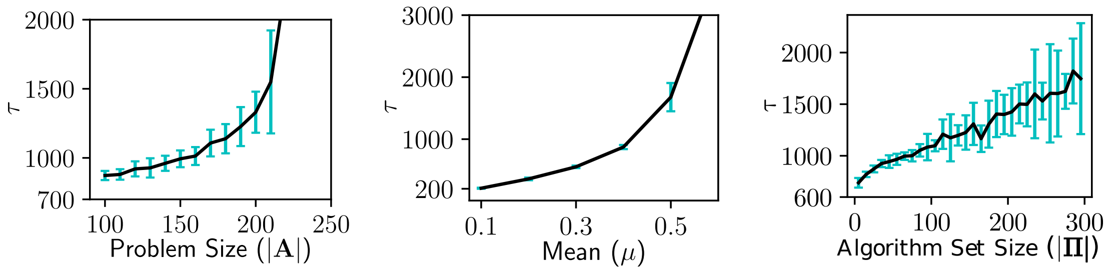
$$p^\Pi_{a^*} = (1 - \epsilon) \cdot \pi^+(a^*) + \frac{\epsilon}{|\Pi|} \sum_{\pi\in\Pi} \pi(a^*) \tag{5.1}$$

The action-selection agent, P_A, selects a^* with probability $p^A_{a^*}$ given by Equation 5.2, which is the usual ϵ-greedy selection over actions. Q is the action-value function, such that $Q(a)$ denotes the expected reward of selecting action $a \in A$.

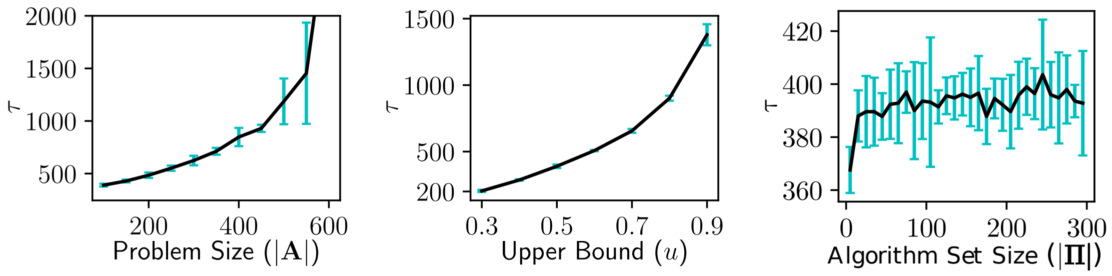
$$p^A_{a^*} = \begin{cases} (1-\epsilon), & \text{if} \operatorname{argmax}_{a\in A} Q(a) = a^* \\ \frac{\epsilon}{|A|}, & \text{otherwise} \end{cases} \tag{5.2}$$

Figure 5.2 shows the meeting point τ, where $p^A_{a^*}$ meets $p^\Pi_{a^*}$ as $|A|$, u or μ, and $|\Pi|$ vary. As before, the error bars show the 95% confidence interval on 5 repetitions.

The meeting point τ grows with statistical significance, under all parameters considered, for both algorithm generation processes. Similarly to the cumulative rewards (Fig. 5.1), the more actions, the more P_A has to explore to find the best one. Likewise, greater μ or u generate stronger algorithms. For these, the curves tend to grow exponentially. The meeting point τ also grows with $|\Pi|$. This happens because all algorithms have non-zero $\pi(a^*)$, and the likelihood of having strong algorithms in the portfolio increases with its size. Propositions 1 and 2 in [Tavares et al., 2018a] prove



(a) Gaussian



(b) Uniform

Figure 5.2: Meeting point (τ) of $p^{A}_{a^*}$ and $p^{\Pi}_{a^*}$ regarding the number of actions ($|A|$), algorithms strength (μ or u), and portfolio size ($|\Pi|$). $\pi(a^*)$ of algorithms in (a) are generated with the Gaussian process, whereas $\pi(a^*)$ of those in (b) are generated with the Uniform process. A point beyond the y axis means that $\tau > 10000$ iterations.

this claim for the Uniform and Gaussian algorithm generation processes, respectively.

In our analysis, we assume that $\pi(a^*)$ comes from certain probability distributions. In practice, however, this does not happen because if the optimal action was known, one would just create an algorithm with $\pi(a^*) = 1$ rather than sampling it from a certain distribution.

Our algorithm generation processes attempt to model possible ways the algorithms, scripts and/or domain-specific heuristics are created. The uniform distribution could model the case where there is not yet an established framework for developing strong algorithms, such that a designer is unable to develop an algorithm whose $\pi(a^*)$ is greater than a certain upper bound. The Gaussian distribution, on the other hand, could model a situation with common knowledge or an established framework to develop strong algorithms (e.g., Monte Carlo Tree Search for computer Go [Browne et al., 2012]). Then, in a set of algorithms, we could expect that there will be a mean and a variance over $\pi(a^*)$, which might vary with different design decisions or parameter configurations.

To extend the multi-armed bandit analysis to multi-state scenarios, we might think of each state as a multi-armed bandit, and the best arm as the one with the highest expected sum of discounted rewards.

In general, one works with a fixed algorithm portfolio, so that $|\Pi|$ is fixed. Moreover, the optimal action is unknown, as is the probability of an algorithm selecting it. However, the increase in the meeting point with the number of actions does not depend on these factors: the more actions available, the more advantageous it is to learn over algorithms, regardless of the criteria (cumulative rewards or probability of selecting the best action). In scenarios with huge action spaces, such as real-time strategy games, the number of actions is usually greater than the training iterations, so that it is unlikely for an agent to find the optimal action. Trading optimality for shorter-term performance is reasonable in this domain and that's the main idea pursued in this dissertation, now illustrated with synthetic experiments.

Experiments in this section were performed in cooperation with Siva Anbalagan and Leandro Marcolino, appearing in [Tavares et al., 2018a], alongside further theoretical analysis and the experiments with linear function approximation of Section 5.4.2.

5.2 Real-time strategy testbeds

5.2.1 StarCraft

StarCraft: Brood War, or simply StarCraft is a real-time strategy game released in 1998 by *Blizzard Entertainment*, which became a reference in the genre. The game is placed in a futuristic world, where three races, with distinct characteristics, battle in a science fiction plot [Blizzard, 2016]. In terms of gameplay, the three races can be described as follows [Liquipedia, 2012]:

- *Zerg*: a insectoid savage race, which attacks with large amounts of cheap and weak units;
- *Protoss*: an alien race with advanced technology, characterized by powerful and expensive units;
- *Terran*: representing the future of human race, with units of intermediate cost and power.

Players harvest mineral and gas resources to train units and construct buildings. More advanced and/or powerful units or buildings require more resources. In real-time strategy games, the technology tree lists the dependencies required to create certain entities or unlock certain abilities. The more powerful units, buildings and abilities are usually located deeper in the technology tree. Table 5.1 shows the number of units, buildings[2] and the technology tree depth of each race and Figure 5.3 presents a game screenshot, with a Terran base.

Race	#units	#buildings	Tech. tree depth
Zerg	13	17	6
Protoss	14	16	6
Terran	13	18	7

Table 5.1: Number of units, buildings and technology tree depth of each StarCraft race.

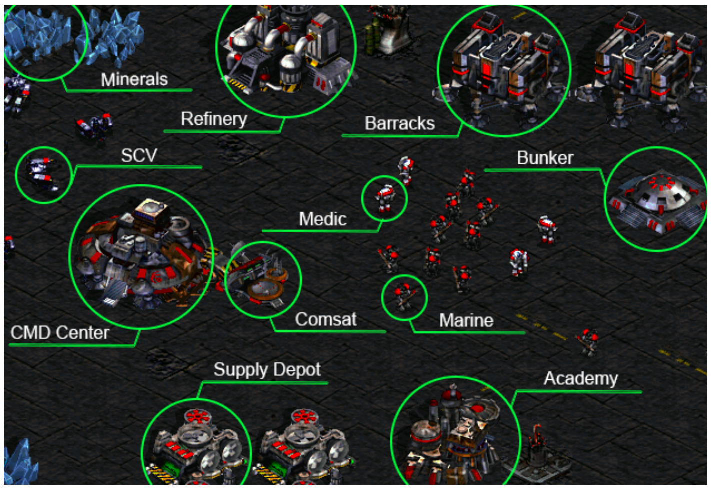


Figure 5.3: Screenshot of a StarCraft Terran base, highlighting specific units and buildings.

StarCraft gameplay has many challenging details, such as:

- Special abilities: some units can become invisible, others can burrow themselves on ground, others have splash damage, whose attacks have an area of effect, and so on;
- Spells: there are many spells available, which include healing or shielding allies, area damages, sight reduction, etc.;
- Air and ground units: air units are not affected by ground obstacles. Besides, some maps have isolated terrain, reachable only by air. Air units increase combat

[2]Counting Terran add-ons, that is, small construction attached to other buildings, as buildings themselves.

complexity, as some units are anti-air specialists whereas others cannot attack air units. Moreover, air transport units can carry ground units either to cut distances or to reach isolated terrain;

- High and low ground: units in high ground can see units in low ground, but the opposite only happens when the unit in high ground attacks. Moreover, units in low ground can miss attacks to units in high ground.

To win a match, a player has to destroy all enemy buildings. Each race has unique units and buildings, generating distinct gameplay styles and yet the game is balanced such that there's no better race than the other: good use of the race's abilities is up to the players. An example match of StarCraft can be seen in `https://youtu.be/CLSlqG9f4AQ`.

StarCraft has a programming interface called BWAPI, aimed at the development of software-controlled players (bots). BWAPI allows a program to retrieve information on the game state and send commands to control its units [BWAPI, 2015]. Its advent and evolution allowed the incremental adoption of StarCraft as an artificial intelligence research platform [Ontañón et al., 2013]. In StarCraft, there is no forward model available: one cannot query the game engine for the next state given the current state and performed actions. The only way to discover the next state is from actual experience: issue the actions and observe the state generated by the game engine, without the possibility of backtracking.

The rising interest in StarCraft as a research platform resulted in the appearance of tournaments, in which academics and enthusiasts send their bots to compete. The appearance of Churchill [2014]'s tournament manager allowed for a significant increase in the number of matches played in tournaments, reducing the effect of randomness, by automating the match configuration and result collection. In the present days, three tournaments happen each year:

- CIG StarCraft AI Competition, promoted by the *IEEE Computational Intelligence in Games* conference;

- AIIDE StarCraft Competition, promoted by the *Artificial Intelligence and Interactive Digital Entertainment* conference.

- SSCAIT (Student StarCraft AI Tournament), hosted in the *Czech Technical University*, in Prague and in the *Comenius University*, in Bratislava.

5.2.2 μRTS

The μRTS platform, illustrated in Figure 5.4, is a real-time strategy (RTS) game designed to facilitate artificial intelligence research [Ontanón, 2013], especially for search-based approaches. It simplifies RTS games by having a single race, fewer types of units and buildings (six in total, detailed next), a shallow technology tree (with three levels), and a simpler combat model (with no special abilities, spells, different terrain heights or air units).

The significant simplifications, compared to StarCraft, aim to provide a simpler and cleaner experiment testbed to perform initial validation of approaches, without the need to handle all the details of a complete commercial game [Ontanón, 2013]. Moreover, μRTS provides a forward model to foster the development of search and planning methods: it is possible to query the game engine for the next state given a state and the performed actions, without having to actually perform the actions and experience the resulting state.

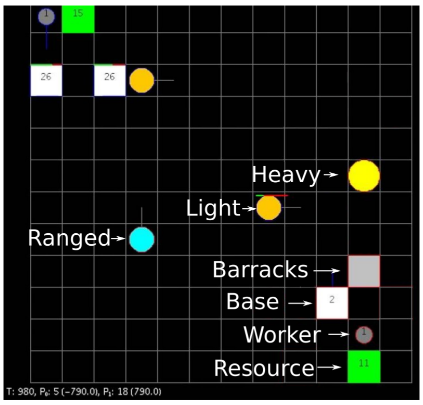


Figure 5.4: A screenshot of μRTS.

In μRTS, there are two types of buildings: *Bases* and *Barracks*. Bases produce *Workers* and Barracks produce military units. Workers construct buildings, harvest *resources* (needed to construct buildings and produce units) and have limited melee combat ability. Military units are either *Heavy*, *Light* or *Ranged*. Heavy is a strong but slow melee combat unit. Light is a weak but fast melee combat unit. Ranged are weak combat units, but can attack from distance. To win a μRTS match, a player must destroy all buildings and kill all adversary units.

Since 2017, a μRTS tournament is promoted by the CIG (*IEEE Computational Intelligence in Games*) conference.

5.3 The game of algorithm selection

Experiments in this section evaluate the game of algorithm selection, stated in Definition 8 in Section 4.4.1.

In this model, at the beginning of an underlying game match, agents select an algorithm to play on their behalf. Selected algorithms play the entire match and agents observe the final outcome. Thus, the game is modeled as a normal-form game, whose payoff matrix displays the relative performance among algorithms. In this section, we are more interested in game-theoretic aspects of algorithm selection, namely, equilibrium selection policies and safe opponent exploitation.

Our experiments were performed with StarCraft (See Section 5.2.1) as the underlying game, using StarCraft software-controlled players (bots) as algorithms. A StarCraft bot defines a game-playing policy, mapping states to actions, satisfying our definition of algorithm. Hence, in this section's experiments, agents select a bot to play a StarCraft match on their behalf. The agent outcome is that of its selected bot. We use the terms bot and algorithm interchangeably hereafter.

5.3.1 Methodology

Our experiments are round-robin tournaments among various agents, each implementing an algorithm selection method. Each agent faces every other for 1000 matches. Tournaments are executed in stationary and non-stationary scenarios. In the stationary scenario, algorithms do not learn, whereas in the non-stationary scenario, they can employ learning mechanisms. An algorithm employing a learning mechanism defines different policies as it gains experience. Its performance against other algorithms may vary, and this causes non-stationarity in the game of algorithm selection's payoff matrix.

In StarCraft, bots usually learn by updating scores of their hard-coded strategies, such as build-orders, based on match results against each specific opponent. This is called intergame learning in [Ontañón et al., 2013]. Bots with a wider range of strategies and/or more efficient scoring mechanisms perform better in the long run than their more limited counterparts[3].

The portfolio of algorithms in our experiments is the same for both stationary and non-stationary scenarios, although it was determined under stationary conditions, as follows: we built a payoff matrix, shown in Table 5.2, by simulating 100 matches among

[3]For an example of how bots performance vary as they learn, one can refer to the AIIDE 2017 tournament learning curve at `https://www.cs.mun.ca/~dchurchill/starcraftaicomp/2017/win_percentage_graph.html`.

eight Protoss bots of AIIDE 2015 tournament [Churchill et al., 2015]: UAlbertaBot, Ximp, Xelnaga, CruzBot, NUSBot, Aiur, Skynet and SusanooTricks, with learning capabilities disabled - which ensures stationarity. All matches were played on a single map, Fortress[4], to avoid variability due to different maps. We then estimated the payoffs of all pairs of bots by the rate of victories minus the rate of defeats[5]. A positive value means that row bot dominates the column bot, winning more than 50% matches. This is enough for a rational agent to prefer one bot over the other.

Bot	UAlb	Ximp	Xeln	Cruz	NUSB	Aiur	Skyn	Susa
UAlberta		**0.94**	**0.96**	**1.00**	**1.00**	**0.94**	**0.84**	**1.00**
Ximp	-0.94		**1.00**	**0.94**	**0.94**	**0.72**	**0.50**	**1.00**
Xelnaga	-0.96	-1.00		-0.48	**0.72**	**0.46**	**0.46**	**1.00**
CruzBot	-1.00	-0.94	**0.48**		**0.60**	**0.34**	-0.68	**0.98**
NUSBot	-1.00	-0.94	-0.72	-0.60		**0.48**	**0.94**	**0.88**
Aiur	-0.94	-0.72	-0.46	-0.34	-0.48		**0.58**	**1.00**
Skynet	-0.84	-0.50	-0.46	**0.68**	-0.94	-0.58		**1.00**
Susanoo	-1.00	-1.00	-1.00	-0.98	-0.88	-1.00	-1.00	

Table 5.2: Payoff matrix of AIIDE 2015 Protoss bots on Fortress map. Values in bold indicate that the row bot dominates the column adversary.

UAlbertaBot and Ximp dominate all bots[6] and SusanooTricks is dominated by all others. Thus, for the algorithm portfolio, we removed UAlbertaBot and Ximp, because dominant bots are obvious choices (Nash Equilibrium in pure strategies) and SusanooTricks, because dominated bots would never be chosen by rational agents. Thus, the portfolio of algorithms in our experiments contains Xelnaga, CruzBot, NUSBot, Aiur and Skynet. Figure 5.5 illustrates the dominance among the remaining bots, based on the payoff matrix in Table 5.2. The fact that all bots have incoming edges means that any bot is dominated by at least another bot, and the outgoing edges indicate that any bot dominates at least another bot. Moreover, the figure indicates the many cyclical interactions among bots.

In this situation, Nash Equilibrium exists only in mixed strategies, i.e., a probability distribution over bots. Table 5.3 shows the calculated Nash Equilibrium for our portfolio. This equilibrium - valid for the stationary scenario - was calculated via Game Theory Explorer [Savani and von Stengel, 2015]. To calculate the equilibrium, we filled the matrix diagonals with zeros. This means that a bot would draw against itself, or win and lose an equal amount of matches.

[4]Fortress is symmetric map with 4 starting locations, each with the same amount of resources (`http://liquipedia.net/starcraft/Fortress`).

[5]Table A.1 in the Appendix A shows the win rate.

[6]Ximp becomes dominant after we remove UAlbertaBot.

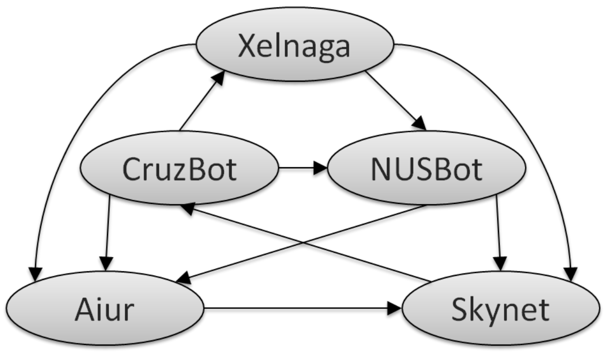


Figure 5.5: Dominance graph for AIIDE 2015 Protoss bots on Fortress map, learning disabled.

Table 5.3: Nash Equilibrium among selected bots.

Strategy	Probability
Xelnaga	41.97%
CruzBot	28.40%
NUSBot	0%
Aiur	0%
Skynet	29.63%
Total	**100%**
Expected payoff	0

In equilibrium, NUSBot and Aiur have zero probability because, although they dominate other bots, Xelnaga dominates them and their dominated bots (see Fig. 5.5). The expected payoff of zero means that an agent selecting algorithms according to the equilibrium policy is expected to win and lose an equal number of matches.

For both stationary and non-stationary algorithm selection tournaments, the following agents, or algorithm selection methods, competed:

- Skynet: always select Skynet;
- Xelnaga: always select Xelnaga.
- Reply-matrix: select the best response against the opponent's last choice, by looking at the payoff matrix;
- Freq-matrix: same as reply-matrix, but select the best response against the opponent's most frequent choice;

- Reply-hist: same as reply-matrix, but the best response is computed by looking at the history of past matches, simply by counting which option has beaten the opponent's last choice more times;
- Freq-hist: same as reply-hist, but counters the opponent's most frequent choice;
- ϵ-greedy: select a random algorithm with probability ϵ, and its most victorious algorithm with probability $(1 - \epsilon)$;
- UCB (Upper-Confidence Bound): selects the algorithm with highest mean reward + upper-confidence bound, calculated via the UCB1-tuned formula of Auer et al. [2002];
- Exp3: the Exp3 algorithm of Auer et al. [1995] (described below);
- Nash: plays according to the Nash Equilibrium policy in Table 5.3;
- ε-Nash: attempts to exploit opponent with probability ε (by playing freq-matrix) and plays the safe strategy (Nash Equilibrium) with probability 1 - ε.
- minimax-Q: the minimax-Q algorithm of Littman [1994] (see Section 2.3.2).

Reply-matrix, freq-matrix, reply-hist and freq-hist are variations of Fictitious play [Brown, 1951][7]. Freq-matrix is the most similar to fictitious play, except that it initializes the opponent's move count with zeros and counters the opponent's most frequent choice rather than the mixed strategy induced by its empirical frequencies. The other methods become increasingly different: freq-hist does not query the payoff matrix to calculate the best-response; reply-matrix counters only the opponent's last move, as a myopic freq-matrix; and reply-hist is a myopic freq-hist.

Exp3 is a method for adversarial multi-armed bandits[8]. Each choice (arm) has an associated weight, which is initialized as zero and increases as the respective arm gives rewards. A factor $\kappa > 0$ determines how much the probability of selecting an arm increases in proportion to the weights. The probability of choosing an arm has two elements, balanced by a factor $\beta \in [0, 1]$: a greedy element, proportional to each choice weight, and a uniform element, to bring randomness and confuse the adversary.

[7]Fictitious play calculates a best response against its opponent by assuming it will play according to the distribution induced by the empirical frequency of past choices. It turns out that the empirical frequencies of two players using this method converges to the Nash Equilibrium of a zero-sum game [Shoham and Leyton-Brown, 2009, Chapter 7].

[8]In adversarial multi-armed bandits, at each round, the opponent selects the payoff of each bandit's arm [Auer et al., 1995].

Exp3 has theoretical performance guarantees without assumptions on the opponent, given that κ and β are initialized properly.

The characteristics of algorithm selector agents is summarized in Table 5.4. Dummy refer to weak agents, that select the same choices regardless of the opponent behavior. Repeaters usually need a number of trials to detect that an option is no longer the best, thus they are likely to repeat choices. GT stands for game-theoretic: agents that explicitly account for the opponent and have theoretical performance guarantees. Adaptive agents use match outcomes to update their internal structures, so their algorithm selection policies might change. Reply-matrix does not have any of those characteristics, thus its row is empty.

	Dummy	Repeater	GT	Adaptive
Skynet	✓			
Xelnaga	✓			
Reply-matrix				
Freq-matrix		✓		
Reply-hist				✓
Freq-hist		✓		✓
ϵ-greedy		✓		✓
UCB		✓		✓
Exp3			✓	✓
Nash			✓	
ε-Nash			✓	
minimax-Q			✓	✓

Table 5.4: Characteristics of the algorithm selection tournament agents.

In the algorithm selection tournament, agents that need to calculate best responses (freq-matrix and reply-matrix) do so via Table 5.2, whereas Table 5.3 guides agents based on Nash Equilibrium (Nash and ε-Nash). In the non-stationary scenario, the best responses and equilibrium calculated that way might refer to a situation that is no longer valid, because these methods do not update the payoff matrix.

We built pools of StarCraft matches to speed up the algorithm selection tournament. When two agents select their algorithms, instead of starting an actual StarCraft match with the corresponding bots, we draw a previously recorded result from the pool. The pool has no records of a bot playing against itself, so when contestants select the same bot, victory is randomly awarded. The result is fed back to the contestants, which update their internal structures if necessary and the match is removed from the pool. For a new contest between other algorithm selectors, the pool is restored.

Agent	Parameter	Value	Meaning
ϵ-greedy	ϵ	0.2	Exploration probability
ε-Nash	ε	0.4	Exploitation probability
minimax-Q	α	0.1	Step size
	ϵ	0.1	Exploration probability
Exp3	β	0.1	Uniform and greedy balance factor
	κ	1.0	Weight update factor

Table 5.5: Agents' parameters.

The pool of matches for the stationary scenario was generated with bots' learning capabilities disabled. When two agents choose their algorithms, we randomly draw a previously recorded result from the pool. The match pool for the non-stationary scenario was generated with bots' learning capabilities enabled. When two algorithm selection methods choose their algorithms, unlike the stationary case, we sequentially draw results from the pool because consecutive records from a pair of bots in the pool reflect their updated knowledge as they played against each other.

All matches among StarCraft bots (Table 5.2 and match pool generation) were executed with the StarCraft AI Tournament Manager[9]. The algorithm selection tournaments were executed with the *StarCraft Nash* engine[10], which implements the described agents and mechanisms to use the match pool.

5.3.2 Results

We ran algorithm selection tournaments to test each scenario: stationary and non-stationary. In each repetition of a tournament, each agent plays 1000 matches against every other agent. Results are averaged over 30 repetitions. Agents' parameters are shown in Table 5.5.

Values in Table 5.5 were determined in prior experiments and ensured a good tradeoff between exploration and exploitation for ϵ-greedy, exploitation vs exploitability for ε-Nash and a good overall performance for minimax-Q and Exp3[11]. Moreover, the payoff matrix for minimax-Q is initialized optimistically with ones.

[9]We added the option to disable bots' learning capabilities in a fork of the old tournament manager in `http://github.com/andertavares/StarcraftAITournamentManager`. An updated version of the original software now integrates this feature.

[10]`https://github.com/DanielKneipp/StarcraftNash`

[11]The results of those prior experiments appear in `https://github.com/DanielKneipp/StarcraftNash/wiki/Test-Results`.

Figure 5.6 shows the overall win rate of each agent across all adversaries in the stationary scenario, ordered from left to right.

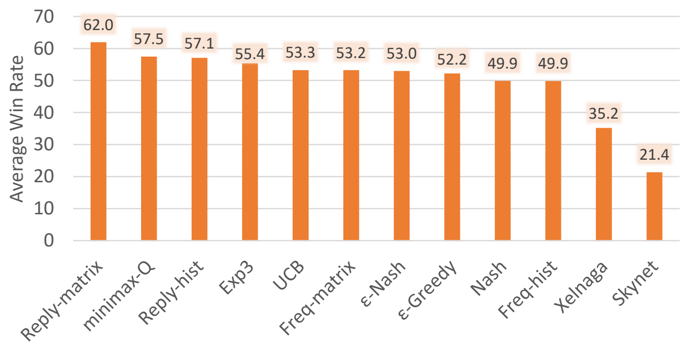


Figure 5.6: Overall win rate in the stationary scenario.

Reply-matrix was the most successful agent in the stationary scenario, followed by minimax-Q. Nash reached the expected accumulated payoff of zero by winning roughly 50% of its matches in the stationary scenario. ε-Nash's overall performance is slightly better than Nash's. Table 5.6 details the performance of these agents against other ones grouped by their characteristics (see Table 5.4).

	Dummy	**Repeaters**	**Game-theoretic**
Reply-matrix	87.35	64.87	49.24
Nash	50.20	49.05	50.05
ε-Nash	65.32	51.32	50.39

Table 5.6: Win rate of specific agents against groups of opponents

Reply-matrix performs well against dummy and repeater opponents. This happens because countering the opponent's last choice is a good policy against those who tend to choose the same option repeatedly. Countering the last choice had no effect on the methods with theoretical guarantees, against which reply-matrix breaks even. Nash breaks even against all groups of opponents. The equilibrium policy is safe and its theoretical performance guarantee holds regardless of the opponent's strength. ε-Nash successfully exploits the dummy opponents and breaks even against the others, suggesting that deviating from the safe policy is useful. Table A.2 in Appendix A details the pairings among all agents in the stationary scenario.

Figure 5.7 shows the overall win rate of each agent across all adversaries in the non-stationary scenario, ordered from left to right. The figure also shows the performance of each agent in the stationary scenario for an easy visualization of improvement or degradation.

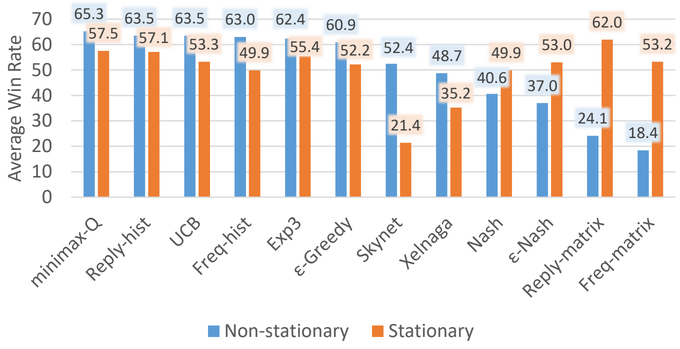


Figure 5.7: Overall win rate in the non-stationary scenario, with stationary performance shown for comparison.

The performance of non-adaptive agents degrades in the non-stationary scenario. This happens because the non-adaptive agents rely on a previously estimated payoff matrix (Table 5.2). However, as algorithms learn, that payoff matrix becomes outdated because the relative performance between algorithms changes. Table 5.7 illustrates this, by showing the win rate of reply-matrix versus Skynet in stationary and non-stationary scenarios.

Scenario	Win rate (%)
Stationary	93.92
Non-stationary	31.80

Table 5.7: Win rate of reply-matrix versus Skynet in both scenarios.

Reply-matrix would always choose NUSBot, as it is the best response for Skynet as per Table 5.2. In the stationary scenario, reply-matrix wins 93.92% of matches against Skynet, which, rounded-up, is exactly the win rate of NUSBot versus Skynet in Table 5.2. In the non-stationary scenario, reply-matrix would still select NUSBot versus Skynet as it relies in Table 5.2. However, Skynet learns how to defeat NUSBot and reply-matrix does not adapt by selecting another bot against Skynet: reply-matrix (selecting NUSBot) wins only 31.80% of matches against Skynet. The numbers in Table 5.7 were extracted from Tables A.2 and A.3 in Appendix A.

The performance of all adaptive agents improves in the non-stationary tournament. Paired t-tests between agents' performances in both scenarios showed that, for all agents, the difference (degradation or improvement) in performance from stationary to non-stationary is significant, with p-values much smaller than 0.0001.

Reply-hist and freq-hist perform much better than reply-matrix and freq-matrix - their non-adaptive counterparts. The multi-armed bandit methods (UCB, Exp3 and

ϵ-greedy) perform similarly: their overall win rate in both scenarios and their increase in performance from stationary to non-stationary is similar.

Methods that respond to opponent's last choice - reply-matrix and reply-hist - were among the most successful in the stationary and non-stationary scenarios, respectively. Those methods resemble human behavior in rock, paper, scissors [Wang et al., 2014]. Although it is easy to second-guess the idea of countering the last choice, no agent has a second-guess mechanism. More generally, opponents that act cyclically or answer deterministically to the agent based on a finite history of interactions are prone to exploitation if modeled as finite automata, as in [Carmel and Markovitch, 1996].

The minimax-Q agent performed well in both stationary and non-stationary scenarios. Its win rate is the highest on the non-stationary scenario. This suggests that, although convergence guarantees are no longer valid, a fixed step size is useful to track non-stationarity. The minimax-Q agent can be seen as the counterpart of Nash for non-stationary scenarios. Whereas our Nash agent relies on an equilibrium policy of a fixed payoff matrix, minimax-Q iteratively updates the matrix and calculates a new equilibrium. Its policy makes use of all historical information, weighting recent experiences more than past ones.

5.3.3 A functional StarCraft bot

This section presents a brief description of MegaBot and results of its participation in StarCraft AI Tournaments. A detailed description is given in Appendix B. MegaBot is a fully-capable StarCraft bot we designed to test the one-shot version of our algorithm selection framework in StarCraft AI Tournaments. It is composed of a meta-reasoning module, implementing minimax-Q, and a portfolio of three other bots: the AIIDE 2015 versions of Skynet, Xelnaga and NUSBot. They are not the strongest competitors, so that a good performance of MegaBot is due to its meta-reasoning mechanism rather than to a strong portfolio.

The meta-reasoning module maintains a matrix with a line per portfolio member and a column per tournament opponent (as each opponent is a bot, which is treated as an algorithm). MegaBot does not know all opponents beforehand, but columns are added as new opponents are faced.

MegaBot's parameters are α, the step size for the payoff matrix updates and ϵ, the exploration probability in the ϵ-greedy selection method. In all tournaments, MegaBot competes with $\alpha = 0.2$ and $\epsilon = 0.1$. The payoff matrix entries are initialized optimistically with ones to encourage premature exploration of all choices.

In StarCraft tournaments, crashes and timeouts are punished with defeats. The

2016 version of MegaBot did not have a mechanism to account for crashes, but the 2017 version was improved with a crash discount mechanism.

MegaBot competed in 2016 and 2017 AI tournaments hosted in CIG and AIIDE conferences as well as in the Student StarCraft AI Tournament (SSCAIT) organized by the Games & Simulations Research Group, Czech Technical University. In CIG and AIIDE tournaments, all competitors face each other during several rounds and the winner is determined by the highest win rate. CIG 2016 was divided in two stages with all entries in the first stage and the top 50% passing to the second stage. CIG 2017, AIIDE 2016 and 2017 followed a single-stage format. In SSCAIT, there is a pre-tournament stage, where bots play along the year, with matches broadcast live. Bots can accumulate knowledge during this stage. Next, in the tournament's round-robin stage, each bot plays against all others for two rounds. There is a follow-up stage where the top 16 bots play a knockout tournament. However, we focus on the round-robin stage, because MegaBot has not made into the follow-up stage.

Figure 5.8 shows MegaBot's win rate per tournament progress, which is shown in percentage because each tournament has a different number of rounds[12]. Blue lines are from 2016 tournaments, red lines from 2017. Solid lines are from AIIDE and dashed lines are from CIG. We show the results of each stage in CIG 2016 separately. Labels at the right show the final win rate. SSCAIT results are not shown in this graph because we do not have direct access to the win rate of each round separately.

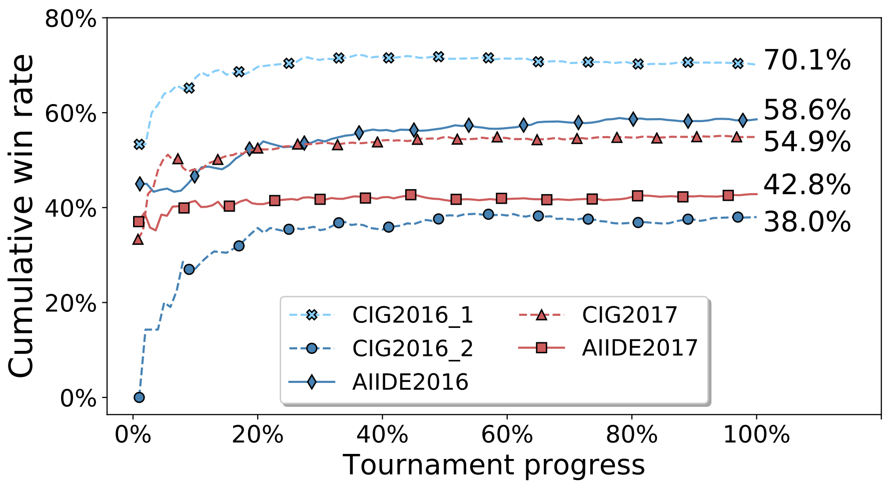


Figure 5.8: MegaBot win rate per tournament progress. Markers help identify each tournament, they do not correspond to data points.

[12]Each bot played the following number of rounds and matches per round in the tournaments: CIG 2016 first stage: 100 rounds of 15 matches each; CIG 2016 second stage: 100 rounds of 7 matches each; CIG 2017: 125 rounds of 19 matches each; AIIDE 2016: 90 rounds of 20 matches each; AIIDE 2017: 110 rounds of 27 matches each.

The minimax-Q learning mechanism implemented in MegaBot was effective as the win rate increases in all tournaments. The less noticeable increase is in AIIDE 2017, where the initial win rate was 37.0% and the final was 42.8%, an increase of 5.8%. The most noticeable increase is in the second stage of CIG 2016, where the initial win rate was 0.0% and the final was 38.0%. MegaBot has performed well against the weaker adversaries of CIG 2016, as the win rate of the first stage is the highest (70.3%). On the other hand, MegaBot did not perform well against the stronger adversaries, as the win rate of the second stage is the lowest (38.0%). All curves in Fig. 5.8 stabilize. Lower settling points indicates that there are more entries stronger than any portfolio member of MegaBot as it is unable to find an algorithm to defeat those adversaries, as it happened in the second stage of CIG 2016.

To compare the performance of MegaBot along the years, Fig. 5.9 shows: (a) the final victory rate and (b) the number of opponents left behind in the rankings. The number of opponents left behind is reported in percentage as each tournament has a different number of participants. We aggregate the two rounds of CIG 2016 with a weighted average from each stage's number of matches and the victory rate.

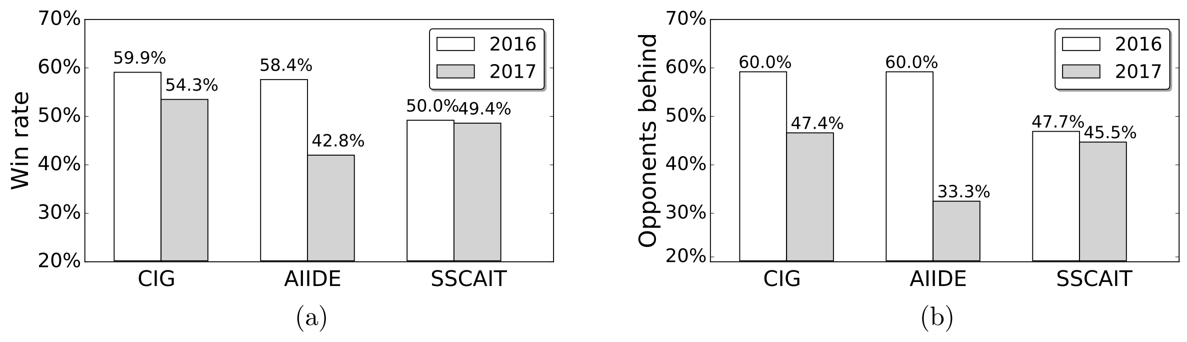


Figure 5.9: MegaBot tournament performance: (a) victory rate and (b) opponents left behind in ranking.

From 2016 to 2017, both the win rates and the percentage of opponents behind MegaBot dropped, despite the introduction of the crash discounting mechanism. As MegaBot's portfolio and selection mechanisms have not changed, this indicates that it faced stronger opponents. Every year, researchers and StarCraft programming enthusiasts observe the new techniques and build on them, preparing stronger entries for the competitions to come. MegaBot has a portfolio of algorithms from 2015 and as time passes, they get increasingly outdated. Figure 5.10 shows the percent of new and updated opponents. SSCAIT 2016 is not shown because the data on bots' updates is not available[13]. From 2016 to 2017, the number of new and updated entries increased in CIG and in AIIDE.

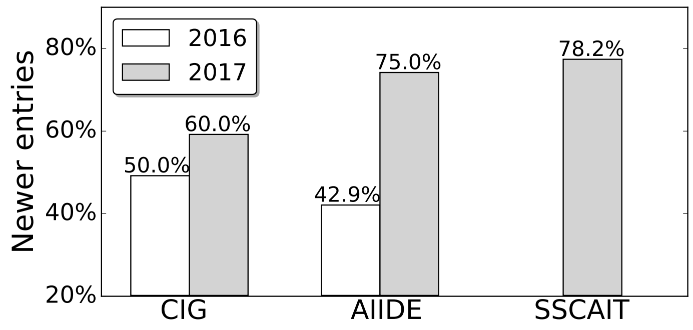


Figure 5.10: New or updated entries in the competitions

Even with the portfolio limitation, the participation of MegaBot in StarCraft tournaments has shown that algorithm selection is a promising approach. In its debut year (2016), MegaBot placed in the top half of all competitions. Moreover, in all tournaments but AIIDE 2017, MegaBot had a noticeable increase in performance from initial to final tournament rounds.

5.3.4 Discussion

The results of our experiments in the stationary game of algorithm selection are consistent with computer rock-paper-scissors tournaments [Billlings, 1999]: deviating from the safe policy in equilibrium was useful against weak opponents, just as much as protecting against exploitation from strong adversaries. Human behavior in rock-paper-scissors contests also exhibit deviation from equilibrium to obtain higher payoffs [Wang et al., 2014]. In our tournament, ε-Nash combines Nash's safety with freq-matrix's exploitation of weak opponents. This is aligned with [McCracken and Bowling, 2004], where previously weak rock-paper-scissors agents performed better when enhanced with safe exploitation techniques. Those experiments aimed at bringing game-theoretic reasoning to real-time strategy games, discussing aspects such as Nash Equilibrium and safe opponent exploitation. This was possible with the game of algorithm selection under stationary conditions.

The non-stationary scenario is more realistic, as we enable algorithms' learning mechanisms. In this setting, non-adaptive methods did not perform well. On the other hand, adaptive methods were able to cope with the evolving nature of the non-

[13]`https://sscaitournament.com/index.php?action=2016`

stationary game. Among these, minimax-Q had the best performance and was selected for the meta-reasoning of MegaBot, our StarCraft AI Tournament competitor.

In principle, minimax-Q would require a great number of matches to achieve a good performance. In fact, each agent played 1000 games against each other in our experiments, which is an order of magnitude more matches than in actual StarCraft tournaments (see, for example Churchill et al. [2015]). Nevertheless, MegaBot's performance suggest that algorithm selection via minimax-Q is a promising approach in StarCraft tournaments as well. As MegaBot's portfolio members are not the strongest competitors, its learning curves indicate the effectiveness of its meta-reasoning module. The increased learning rate (0.2 for MegaBot in the tournaments compared to 0.1 in the experiments) helped mitigating the effect of a smaller number of matches in the learning process.

A limitation of our formal model is the assumption of common knowledge of agents' algorithm portfolio, which in general might not happen. Without the opponent's portfolio, there is no payoff matrix to work with. MegaBot works around this issue by adding new columns to the payoff matrix as it faces new opponent algorithms. It does so by identifying the opponent's algorithm by its name. However, this method overlooks a characteristic of many StarCraft bots: they have hard-coded behaviors, such as build-orders and follow-ups and switch among those behaviors from match to match according to experience. That is, StarCraft bots are already algorithm selectors themselves[14], but MegaBot's method considers the opponent as a single algorithm.

By treating each opponent as a single algorithm and thus as a single column in its payoff matrix, MegaBot reduces the game-theoretic reasoning of minimax-Q to an ϵ-greedy selection method against each opponent. To use the full game-theoretic capabilities of minimax-Q, MegaBot must recognize which algorithm the opponent has selected to update its score accordingly. To accomplish this, MegaBot must label the algorithms according to the actions it can observe, rather than by the name of its selector. Then, with an effective opponent modeler, MegaBot could narrow the equilibrium calculation to consider algorithms with likelihood above certain thresholds, given the observed actions. However, existing opening prediction and opponent modeling approaches (e.g. Weber and Mateas [2009]; Synnaeve and Bessiere [2011]; Stanescu and Čertický [2016]) are based on replay analysis, requiring extensions to work on live matches.

[14]Although disabling learning for the stationary experiments can harm the performance of some bots (making them select a single build-order or randomly switch them between matches), this is not important from the point of view of the algorithm selector, as it is only interested on the algorithm's performance, rather than on its internal mechanisms.

Another limitation of this section's experiments is that our game of algorithm selection is a one-shot decision problem, where agents select algorithms to play an entire match in the underlying game on their behalf. However, minimax-Q is general enough to tackle multi-stage settings, where decision points are associated with world states. Here we use a simplified, single-stage version of the method. Section 5.4 explores multi-stage algorithm selection, albeit not in StarCraft. This happens because the current bots incorporated in MegaBot's portfolio are unable to continue a game started by a different bot. This happens mostly because of unsatisfied preconditions: the portfolio member requires certain data created only by itself in previous match stages. When another portfolio member plays those previous stages, the required data is missing. This can be seen as the lack the Markov property from the portfolio members: the state information they have access is not enough to make a decision.

Experiments in this section required the following computational infrastructure:

- Match pool construction: we ran a StarCraft tournament using 8 Virtual Machines, each with 1 core of Intel Xeon E52650 v3 @ 2.30 GHz CPU and 1GB of RAM. Each machine hosts a StarCraft instance, loaded with a bot, and plays via network against a bot hosted in other machine. That is, we ran at most 4 matches in parallel.

- Algorithm selection tournament: the algorithm selection tournament was executed in a notebook with 2 cores of Intel(R) Core(TM) i7-5500U @ 2.40GHz CPU and 8GB of RAM.

The most computationally expensive part is the match pool construction, not because of our strategic reasoning framework, but because the bots are pitched against each other in actual StarCraft matches. The algorithm selection tournament demands simple hardware, as there is no actual execution of StarCraft matches: results are loaded from the pool. The reasoning procedure of agents is simple, involving updates to the value of a choice given the received outcome and a fast calculation or lookup for the next choice. Minimax-Q is an exception, as it requires solving a linear program to determine the selection policy.

Experiments of this section were presented in [Tavares et al., 2016] (stationary) and [Tavares et al., 2018b] (stationary, non-stationary and MegaBot's performance).

5.4 Multiple decision points

This section presents experiments evaluating the strategic reasoning framework in scenarios with multiple decision points. Experiments were performed in μRTS, a simplified real-time strategy (RTS) game designed to facilitate AI research (see Section 5.2.2). μRTS has various state-of-the-art adversarial search approaches implemented, and it would be interesting to compare reinforcement learning approaches against them. Moreover, μRTS has simple scripts that are able to produce valid actions regardless of the state they are in, as opposed to some StarCraft bots, which are unable to resume a match started by a different bot. In this work, μRTS is configured with full visibility (perfect information) for the players.

μRTS has four simple built-in scripts: *Worker*, *Ranged*, *Heavy* and *Light* rushes. We implemented two in addition to these: *Expand* and *BuildBarracks*. They work as follows:

- Worker rush: trains workers continuously, make one of them gather resources, and send all others to attack the nearest enemy unit;
- Light rush: trains one worker and make it gather resources. Then builds a barracks, which trains Light units constantly, sending them to attack the nearest enemy unit;
- Heavy rush: same as Light rush, but the barracks trains Heavy units;
- Ranged rush: same as Light rush, but the barracks train Ranged units;
- Expand: send a worker to construct a new base close to a resource location;
- BuildBarracks: send a worker to construct a new barracks.

Expand and BuildBarracks do not perform combat actions themselves. Instead, Expand increases resource collection and worker production rate, whereas BuildBarracks increases the military units production rate.

Our strategic reasoning framework simplifies the number of choices to consider in each state, but the number of states in real-time strategy games remains too large for traditional reinforcement learning techniques (see Section 2.4 for a discussion on the complexity if real-time strategy games). In this section we investigate two approaches to generalize learning across states: state aggregation (Section 4.4.1) and linear function approximation (Section 4.4.2).

5.4.1 State aggregation

This section presents experiments with our strategic reasoning framework with multiple decision points in μRTS, using state aggregation to generalize learning across states.

As discussed in Section 4.4.1, in state aggregation, a state abstraction function $\phi : S \to \overline{S}$ groups several primitive states from S into an abstract state in the set $\overline{S}$ of abstract states. More precisely, the set $\overline{S}$ is a partition of S induced by ϕ. Tabular reinforcement learning methods then store entries for choices in each abstract state instead of for each primitive state.

Our set $\overline{S}$ of abstract states is constructed by identifying eight simple features in μRTS. First, we divide the game duration equally into five intervals, related to the match stage: opening, early, mid, late and end. The other seven features account for material advantage between player and opponent in each type of entity: resources, bases, barracks, workers, heavy, light and ranged units. As a simplification, the material advantage features, for each entity type, are discretized in three values:

- Ahead: if the player controls two or more entities than the opponent;
- Even: if the difference between entities controlled by the player and its opponent is between +1 and -1;
- Behind: if the opponent controls two or more entities than the player.

In total, we have $5 \times 3^7 = 10935$ non-terminal abstract states. A terminal is created as an additional abstract state. It is reached whenever a player wins the game or time runs out. That is, all primitive terminal states in μRTS are mapped to the abstract terminal state. Our state abstraction function ϕ then maps a primitive μRTS state to an abstract state by checking the game termination conditions, and then determining the stage from the current game time and counting the entities of each type owned by the player and its opponent to determine the values of the features[15].

Our reward signal is the player score minus the opponent's. The score function counts the material a player has. The function is provided by μRTS, being used as a state evaluation function by some search approaches. The score of a player i is calculated according to Eq. 5.3, where B_i is the amount of resources owned by the player, W_i is the amount of resources being carried by player i's workers, U_i are the

[15]The options induced by out state aggregation scheme and portfolio of algorithms (see Definition 6) are guaranteed to terminate. Even though the material advantage may not change due to the lack of combats or production of units, the game time always goes forward and will eventually reach a posterior game stage. This guarantees that players will always have multiple decision points within a match.

units player i controls, $c(u)$ is the cost to produce a unit, $hp(u)$ are the current hit points of a unit and $hp_{max}(u)$ are the maximum hit points of a unit.

$$\text{score}_i = 20 \cdot B_i + 10 \cdot W_i + \sum_{u \in U_i} 40 \cdot c(u) \cdot \sqrt{\frac{hp(u)}{hp_{max}(u)}} \tag{5.3}$$

In Section 5.4.1.1, we evaluate minimax-Q, which shows robustness when its assumptions are satisfied. As far as we know, this is the first time a multiagent reinforcement learning method is employed in a real-time strategy game. In Section 5.4.1.2, we perform experiments with an opponent-oblivious algorithm selector.

Unless otherwise stated, experiments in this section were executed with μRTS configured with 3000 frames of timeout on the map "basesWorkers24x24".

5.4.1.1 Opponent-aware

This section presents experiments evaluating algorithm selection with multiple states: the stochastic game over options (SG_O in Definition 7). We use a multiagent reinforcement learning approach with game-theoretic foundations, namely, minimax-Q [Littman, 1994], hence, our agent is aware of the opponent's presence. In our model, the decision points are the abstract states, which group primitive states as discussed in Section 5.4.1, and the players' actions are the algorithms/options they can choose (we use the terms algorithms and options interchangeably). The formalism requires complete information, so that all players know each other's portfolio of algorithms.

Littman [1994] introduces minimax-Q and performs experiments on a simplified soccer game with 22 states, where 2 are terminals, and 5 possible actions for each non-terminal state. Experiments in this section aim to reproduce those of Littman [1994] in a much more complex domain, namely μRTS, thanks to our algorithm selection framework.

We follow the same methodology of Littman [1994]: we train minimax-Q in self-play and against a random algorithm selection policy. The resulting policies are named MM and MR, respectively. Similarly, as a reference, we also train Q-learning in self-play and against a random policy, naming the resulting policies as QQ and QR.

Training was performed in 120 thousand episodes (or μRTS matches), with $\epsilon = 0.1$ and α starting at 1 and decaying at rate 0.999961624 per episode to reach 0.01 in 120 thousand episodes. All matches were executed in the map "basesWorkers24x24" to reduce the variability due to different maps.

To test robustness, we train a new instance of Q-learning against each resulting policy held fixed ($\alpha = \epsilon = 0$). Q-learning considers the opponent as part of the environment and attempts to learn an optimal policy in that augmented environment. A stationary environment augmented with a stationary adversary remains stationary. Thus, the best policy found by Q-learning is one that maximally exploits that opponent. Hence, the resulting policies of this training process are the "nemesis" of each MM, MR, QQ and QR.

Additionally, we test the resulting policies against the Monte Carlo Algorithm Selection (MCAS) approach of [Sailer et al., 2007], which is an algorithm selector (see Section 3.4). MCAS uses Monte Carlo simulations among algorithms (or scripts) to fill a normal-form game payoff matrix at each decision point. Then, an equilibrium policy is calculated and an algorithm is selected accordingly. For a fair comparison, we restrict MCAS to decide in the same decision points of our learning approaches, namely, when the abstract state changes. We configure MCAS to simulate the match for up to 200 frames ahead, using a state evaluation function to indicate the reached state value with a total time limit of 5 seconds for each decision. MCAS uses the four combat-capable scripts of μRTS: Worker, Heavy, Light and Ranged rushes (see Section 5.4 for a description). The other two (Expand and BuildBarracks) are out of MCAS's portfolio because they do not perform combat actions and would thus lose all simulations.

Our tests consist of 100 matches of MM, MR, QQ and QR policies against each adversary: a random algorithm selection policy, their nemeses and Sailer et al. [2007]'s MCAS. Figure 5.11 shows the result of each test, averaged over 5 repetitions. Each repetition is the training of a policy and its respective test, rather than 5 tests of a single trained policy.

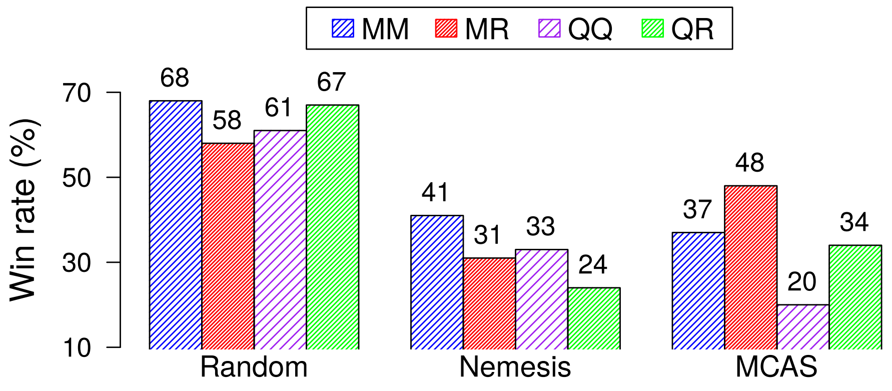


Figure 5.11: MM, MR, QQ and QR policies against the three adversaries

In general, the minimax-Q's resulting policies (MM and MR) were more robust against the range of opponents than the Q-learning's resulting policies (QQ and QR). For the ones trained in self-play, MM fares better than QQ against MCAS and the respective nemesis. Similarly, MR fares better than QR against MCAS and the respective nemesis, although QR fares better than MR against the random opponent. This is expected because Q-learning finds a policy against the specific training adversary, whereas minimax-Q reasons strategically by considering not only what the opponent does, but what it could do.

The results in Fig. 5.11 suggest that minimax-Q has not converged to the equilibrium policy in 120 thousand training episodes. Otherwise, the performance of MM and MR would not vary on the same test opponent: prior to convergence, the training opponent can have an impact as different parts of the state space and joint-actions are sampled [Littman, 1994]. In the limit, MM and MR would be policies in equilibrium, performing equally against the same test opponent. In our experiments, MM fared better than MR against random and its respective nemesis, but MR fared better against MCAS.

5.4.1.2 Opponent-oblivious

Previous sections presented experiments where both players were algorithm selectors and their portfolio of algorithms was of common knowledge. This assumption, albeit unrealistic in general, is required in the stochastic game over options (Definition 7).

Experiments in this section were performed with an opponent-oblivious agent, namely, a Q-learning agent that considers the opponent as part of the environment, following the approach described in Section 4.3.2. With our abstract state representation (described in Section 5.4.1), the agent must map abstract states to algorithms in its portfolio. This is an instance of learning over options in a Markov Decision Process [Sutton et al., 1999a].

Each run of our approach consists of 5000 training episodes against an adversary, where each episode is a μRTS match. We then extract the resulting policies and run 100 test matches against the same adversary. Parameters were determined from preliminary experiments as follows: α decays exponentially from 1 to 0.01 in the 5000 episodes (that is, with decay rate of 0.99907939), $\gamma = 0.9$ and $\epsilon = 0.1$. The search methods (our adversaries) were executed with their default parameters in μRTS. For example, the think time is 100 milliseconds per game frame. For stability, we run 5 repetitions of each experiment and report the average.

We use the following state-of-the-art search methods as our opponents to evaluate our learning over options approach:

- Monte Carlo Algorithm Selection, or MCAS for short [Sailer et al., 2007]: at each decision point, it fills a normal-form game payoff matrix with Monte Carlo simulations between pairs of algorithms. It was also tested in the opponent-aware experiments (Section 5.4.1.1);

- Adversarial Hierarchical Task Network, or AHTN for short [Ontañón and Buro, 2015]: this approach extends the use of hierarchical-task network (HTN) planning, which encode domain knowledge via the definition of useful tasks in the domain, with a minimax-like game-tree search;

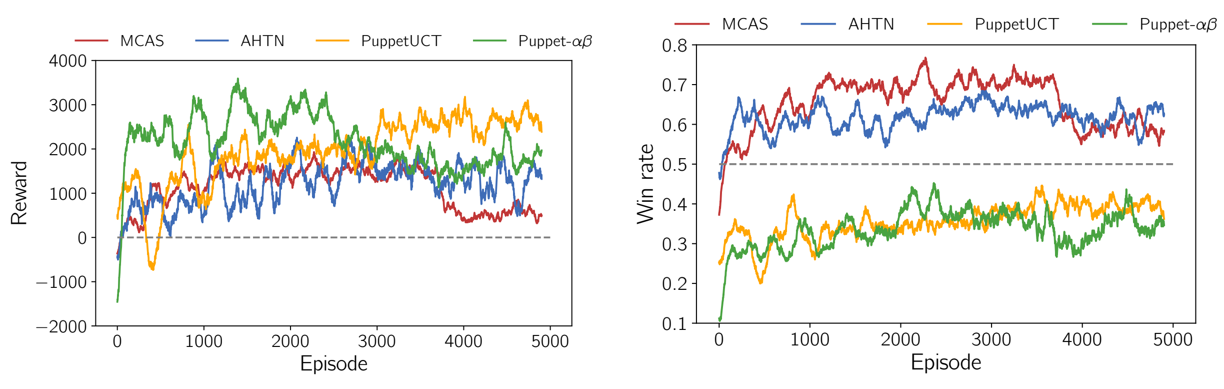
- Puppet-$\alpha\beta$ [Barriga et al., 2015]: PuppetSearch is a framework that augments the capabilities of game-playing scripts by means of move choices they expose to search procedures. By exposing a restricted set of actions, the search methods can look further ahead, finding potentially better solutions. Moreover, as the computational budget allows, more choices can be exposed so the search algorithms can investigate with a broader perspective. In [Barriga et al., 2015], StarCraft playing scripts are combined with a version of the α-β considering durations (ABCD) algorithm [Churchill et al., 2012]. This approach was ported to μRTS as well;

- PuppetUCT [Barriga et al., 2017]: this is a variant of Puppet-$\alpha\beta$, which replaces the ABCD algorithm with a version of the upper-confidence bound for trees considering durations (UCTCD) algorithm of Churchill and Buro [2013].

As in the previous experiments, the reward function measures material advantage: the player's scores minus the opponent's. Score is calculated via Eq. 5.3.

Figure 5.12 shows the training performance of Q-learning over options against each adversary, in terms of mean reward (a) and win rate (b) on the 5 repetitions. We show the running average on 100 episodes to display the overall trend of the training process. The baseline for the reward is zero: a superior value means that the player had more material than its opponent throughout the match. The baseline for the win rate is 0.5: a superior value means that the learning agent has defeated its adversary in most repetitions.

Rewards (Fig. 5.12a) become positive after a number of episodes against all opponents. As the reward function reflects material advantage, the agent is being able to maintain more units than its opponents during parts of the game. However, the plot shows rewards accumulated for an entire episode. It might be the case that the agent maintain material advantage for a period in the game but loses the match near its end. In such case, the reward accumulated in that episode might still be positive. Moreover, the rewards oscillate through the entire training period, even with the running averages. This might be due to actions of the adversary: the agent might



(a) Reward (b) Win rate

Figure 5.12: Running average on 100 episodes of mean reward and win rate in the 5 repetitions against each training adversary.

learn a policy that reaches states where the search opponent performs well, reducing the agent reward, and forcing it to change its policy again. It is also possible that the agent could learn more stable and stronger policies by training for more episodes. However, training against search methods consumes considerable time because the game speed is limited by the time those approaches need to calculate their actions[16].

The win rate during training (Fig. 5.12b) increases against all opponents, although the increment and final performances are weaker against PuppetUCT and Puppet-$\alpha\beta$ than against AHTN and MCAS. The learning agent was able to win the majority of matches against AHTN and MCAS, getting above the baseline (0.5), but has not succeeded against PuppetUCT and Puppet-$\alpha\beta$, remaining below the baseline during the entire training. This suggests that our approach was able to learn strong policies against AHTN and MCAS, but was unable to do the same against PuppetUCT and Puppet-$\alpha\beta$. Moreover, this indicates that the reward function is not directing the agent towards victories. The agent is likely acting myopically: even if it loses a match in the end, receiving a few negative rewards, it might be taking decisions that maximize its immediate reward in earlier game stages, ending up with a positive reward balance.

To evaluate resulting policies' strength, we run 100 test matches against the same training opponents. In the tests, ϵ and α are set to zero, to ensure the agent acts greedily according to the learned policy, without further updates. As a baseline, we also tested random and fixed policies over options against each adversary. Each fixed policy always chooses a single option. We tested a fixed policy for each available option with combat capabilities: Light, Heavy, Worker and Ranged rushes. Figure 5.13 shows the results. Heavy rush is omitted because it did not win any match in the tests.

[16]Training against AHTN takes about one day, whereas against PuppetUCT takes about a week.

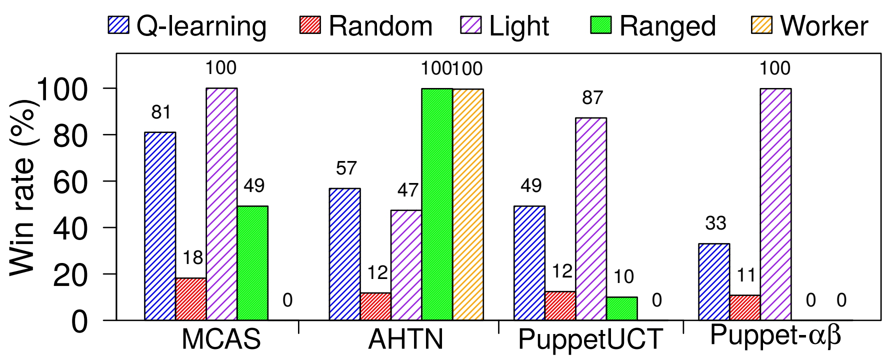


Figure 5.13: Victory rate of Q-Learning resulting policy, plus random and fixed policies (Light, Ranged, Worker) over options vs each search method. A policy with missing bar has zero victories against the specific adversary.

The resulting policy of Q-learning outperforms a random policy over options against all adversaries. Interestingly, however, for each adversary, there is a fixed policy over options that outperforms the Q-learning resulting policy. This means that each adversary can be defeated by a simple script. Nevertheless, learning is useful, because no fixed policy defeated all search approaches. Thus, an agent must learn which option (or combination of options) defeats each adversary.

Our approach learned a strong policy against MCAS, competitive policies against AHTN and PuppetUCT, and a poor policy against Puppet-$\alpha\beta$. Chronologically, MCAS [Sailer et al., 2007] was proposed earlier, followed by AHTN [Ontañón and Buro, 2015] and then by Puppet-$\alpha\beta$ [Barriga et al., 2015] and PuppetUCT [Barriga et al., 2017]. Our approach fared better against older approaches rather than newer ones, except by Puppet-$\alpha\beta$ which is older than PuppetUCT and imposed more difficulties to our approach.

Our Q-learning over abstract states for algorithm selection in real-time strategy games showed limited performance against some of the state-of-the-art search approaches, even though the agent tried to learn a specialized policy. The presented state aggregation model, although enabling the adoption of tabular Q-learning, still has a large number of abstract states (10935 as discussed in Section 5.4.1) compared to the number of training episodes (5000, in Section 5.4.1.2). Although many states will never be reached in real games, the number of possible states is large compared to the number of training episodes, so that some state-action pairs will have only a few visits to produce precise estimates. Training for more episodes would require a long time, as the search approaches are slow compared to Q-learning decision procedure. Another point of improvement is the reward function, which directs the agent to a myopic behavior regarding material advantage.

The identified limitations are tackled via the linear function approximation for better learning generalization, as well as a victory-oriented reward function in Section 5.4.2.

5.4.2 Linear function approximation

Previous sections have investigated tabular reinforcement learning methods with state aggregation. An opponent-aware minimax-Q agent (Section 5.4.1.1) showed robustness against a nemesis policy, but requires enumeration of opponent actions to estimate the joint-action-values. An opponent-oblivious Q-learning agent (Section 5.4.1.2) achieved competitive performance against some search approaches, but could not consistently defeat all of them. It showed limitations regarding the required training time due to the lack of effectiveness of generalization under state aggregation.

In this section we also evaluate an opponent-oblivious agent, so that we do not require enumerating the opponent's actions, but with a linear function approximation approach instead of state aggregation, to reduce the required training time to achieve strong game-playing performance. Experiments in this section were performed in co-operation with Siva Anbalagan and Leandro Marcolino, appearing in [Tavares et al., 2018a].

In our linear function approximation approach, a state $s \in S$ is represented by a feature vector with n features: $\mathbf{f}(s) = \langle f_1(s), \ldots, f_n(s) \rangle$. We associate each algorithm π in the portfolio Π with a weight vector: $\mathbf{w}^\pi = \langle w_1^\pi, \ldots, w_n^\pi \rangle$.

As we're selecting algorithms, the action-value function $Q(s, a)$ is replaced by an algorithm-value function $Q(s, \pi)$, which is approximated by $\tilde{Q}(s, \pi, \mathbf{w}) = \mathbf{f}(s) \cdot \mathbf{w}^\pi = \sum_{i=1}^{n} f_i(s) \cdot w_i^\pi$.

For the weight update, we use the Sarsa(0) method of Rummery and Niranjan [1994] with their neural network replaced by our linear function approximator. Each time the agent selects an algorithm π in state s, observes the next state s', the reward r and chooses the next algorithm π', Sarsa(0) updates each weight $w_i^\pi \in \mathbf{w}^\pi$ with: $w_i^\pi \leftarrow w_i^\pi + \alpha \left[R(s') + \gamma \tilde{Q}(s', \pi', \mathbf{w}) - \tilde{Q}(s, \pi, \mathbf{w}) \right] f_i(s)$, where α is the step size, or learning rate (see Eq. 4.6 and related discussion).

Our feature vector $\mathbf{f}$ is organized in four groups of components: unit count per quadrant, average unit health per quadrant, resources and game time. Except for the game time, the components are associated with a player $p \in \{1, 2\}$. Given a map with q square regions, or quadrants, y unit types and 2 players, the feature vector components are given as:

- Unit count per quadrant (ρ). For each player p: $\rho_p = \{{}^{p}u_1^1, \ldots, {}^{p}u_y^1, \ldots, {}^{p}u_1^q, \ldots, {}^{p}u_y^q\}$, where ${}^{p}u_i^j$ is the number of units of type i in quadrant j owned by player p. With 2 players, y unit types and q quadrants, there are $2yq$ features in this group;

- Average unit health per quadrant (η). For each player p, $\eta_p = \{h_p^1, \ldots, h_p^q\}$, where h_p^j is the average health of all units in quadrant j owned by player p. With 2 players and q quadrants, there are $2q$ features in this group;

- Resources (B): B_p denotes the amount of resources owned by player p. This group has 2 features, one for each player;

- Game time (t): t denotes the current game time (number of frames since the beginning). This group has a single feature.

Hence, our feature vector is $\mathbf{f} = \langle 1, \rho_1, \rho_2, \eta_1, \eta_2, B_1, B_2, t \rangle$, where the constant 1 is the bias term used in linear regression. In total, there are $1+2yq+2q+2+1 = 2yq+2q+4$ features. μRTS has $y = 6$ unit types[17] and we divided the map (basesWorkers24x24) in $q = 9$ quadrants in our experiments, yielding: $2 \cdot 6 \cdot 9 + 2 \cdot 9 + 4 = 130$ features.

Our algorithm portfolio contains the rush scripts (Worker, Light, Ranged and Heavy) plus the production rate boosters (Expand and BuildBarracks – Section 5.4 has a detailed description). As each algorithm has its set of weights, we have $130 \cdot 6 = 780$ weights to adjust. We select an algorithm in the portfolio using exponentially decaying ϵ-greedy (decayed after every training game).

We compare our linear function approximation approach for learning over algorithms (henceforth named FA) against the following state-of-the-art search approaches in μRTS: AHTN, PuppetUCT, Puppet-$\alpha\beta$, NaiveMCTS and StrategyTactics. AHTN, PuppetUCT and Puppet-$\alpha\beta$ are described in Section 5.4.1. StrategyTactics [Barriga et al., 2017] uses a convolutional neural network to predict the output of PuppetSearch, allowing more time to be used by the tactical search algorithm. NaiveMCTS [Ontañón, 2017] employs Monte Carlo Tree Search, but uses a sampling strategy based on combinatorial multi-armed bandits. StrategyTactics won the CIG 2017 μRTS competition[18], and NaiveMCTS was used as a baseline.

We used the map "basesWorkers24x24", and the best parametrization we found: $\alpha = 10^{-4}$, $\gamma = 0.9$, ϵ exponentially decaying from 0.2 against Puppet-$\alpha\beta$, PuppetUCT and AHTN; and decaying from 0.1 for NaiveMCTS and StrategyTactics, after every

[17]The unit types are in μRTS are: Heavy, Light, Ranged, Worker, Base, Barracks and Resource, but we disregard Resource as a unit type.

[18]`https://sites.google.com/site/micrortsaicompetition/competition-results`

game (decay rate ≈ 0.9984). Rewards are -1, 0 or 1 for defeat, draw and victory, respectively.

We perform two evaluations:

1. *Specific*: we trained FA in 500 games against Puppet-$\alpha\beta$, PuppetUCT and AHTN; and in 100 games against NaiveMCTS and StrategyTactics. The resulting policy is tested against the same training adversary. All games have maximum duration of 3000 frames, declared a draw on timeout;

2. *Generic*: we trained a new instance of FA in 500 games versus the *specific* policy obtained against PuppetUCT. The single resulting policy of this new FA instance is tested against all adversaries.

We report the test results, consisting of 100 matches between the given policy (specific or generic) and the adversary. Experiments were repeated 5 times for stability, and error bars show the 99% confidence interval, unless otherwise stated. We consider statistical significance as $p \leq 0.01$. Figure 5.14 shows the results of specific and generic policies (2 first bars of each group), alongside the individual combat-capable algorithms in the portfolio (4 remaining bars of each group) against each adversary. Results of individual algorithms against AHTN, PuppetUCT and Puppet-$\alpha\beta$ were obtained from Fig. 5.13[19].

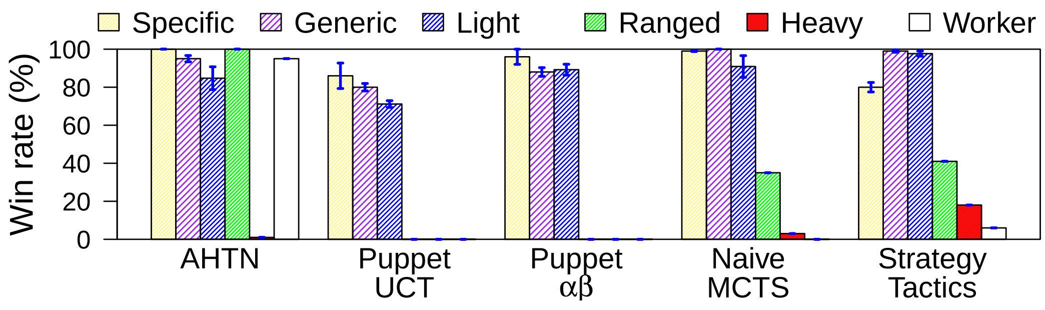


Figure 5.14: Algorithm selection with function approximation against state-of-the-art search approaches.

The generic and all specific policies significantly defeat all opponents, with win rates higher than 80%. These policies have similar win rates among themselves, but the generic is significantly better against StrategyTactics. This might happen because the generic further elaborates on a policy that was already strong (the specific trained against PuppetUCT).

[19]In Fig. 5.13, Heavy rush was omitted because it did not win matches, but it is added in Fig. 5.14 because it wins some matches against NaiveMCTS and StrategyTactics.

There is at least one single algorithm that defeats each adversary. However, unlike the Q-learning agent with state aggregation, here our performance with function approximation is either significantly better than all individual algorithms, as against PuppetUCT and NaiveMCTS, or statistically similar to the best algorithm, except for StrategyTactics, where LightRush is stronger than the specific policy. Moreover, in a subsequent test we further verified that the generic policy wins all matches against each individual algorithm.

Our function approximation approach for algorithm selection does not prevent the learning agent from switching among algorithms at every game frame. Thus, the agent must learn to repeatedly select algorithms to follow a course of action, potentially improving game-playing performance. Figure 5.15 shows the number of frames, on average, the *specific* policies choose an algorithm consecutively against each adversary. Error bars show the standard deviation. For example, the specific policy against AHTN selects RangedRush for about 10 frames in a row before switching to a new algorithm. If RangedRush is selected back again, it would be maintained for about another 10 frames.

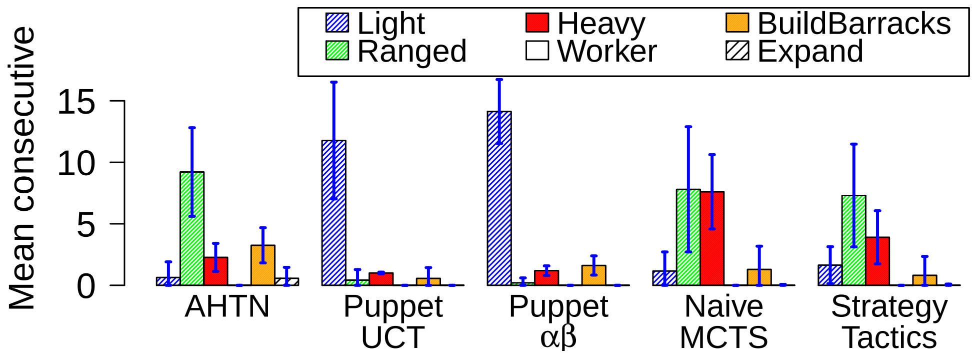


Figure 5.15: Consecutive selection of algorithms by the specific policies against each adversary.

The Light-, Ranged- or HeavyRush, which are “combat” scripts, are maintained longer than BuildBarracks, which is a “supporting” script. The FA agent maintains combat pressure and, from time to time, activates BuildBarracks briefly to have a worker start the barracks construction, changing to another combat script (they do not interrupt the construction). Expand is called rarely (only against AHTN) and WorkerRush is never called. WorkerRush is weak against the search approaches, except AHTN (see Fig. 5.14), so the FA agent learns to leave it unused. BuildBarracks supports the military scripts, whereas Expand supports WorkerRush and the harvest of additional resource fields. However, the map (basesWorkers24x24) has a single resource field for each player, and WorkerRush is not used, such that the creation of new bases or additional workers is unnecessary.

Although our model does not enforce repeated selections, the time feature might induce the FA agent to repeatedly select an algorithm. It has a similar effect of the time discretization in stages, used with state aggregation (Section 5.4.1). However, with function approximation, the agent learns the importance of the time feature for each algorithm, whereas in state aggregation, time was discretized prior to the experiments and has the same importance for all algorithms.

The most repeated algorithm is among the strongest against AHTN, PuppetUCT and Puppet-$\alpha\beta$. However, against NaiveMCTS and StrategyTactics the agent combines different algorithms (Ranged and Heavy), instead of favoring the strongest (LightRush). This suggests that previously weak scripts alone (see Ranged and Heavy in Fig. 5.14) can be combined to achieve a strong gameplay behavior.

5.4.3 Discussion

Our algorithm selection framework combined with state aggregation allowed the use of Q-learning and minimax-Q, two traditional, tabular reinforcement learning methods, in a real-time strategy game. Our opponent-aware experiments showed that explicitly accounting for the opponent is useful, as the resulting policies of Q-learning were less robust against their nemeses than those resulting from minimax-Q. On the other hand, the resulting policy of Q-learning obtained in 120 thousand episodes of self-play (QQ in Fig. 5.11) is not as strong against MCAS as the specialized policy obtained in 5 thousand episodes (Fig. 5.13). We noticed in Section 5.4.1.2 that our reward function might be directing the learning agents towards a myopic behavior, not necessarily correlated with victories in matches. This reward function is used in both opponent-oblivious and opponent-aware experiments.

Moreover, in the opponent-aware experiments, MCAS - an online search approach - imposed more difficulties to the four resulting policies than their nemeses, which were generated with 120 thousand training episodes against each fixed policy. This suggests that either the reward function or the proposed state aggregation scheme are not paying off in terms of achieved performance. Moreover, in the opponent-oblivious experiments, our Q-learning agent was not able to consistently defeat all state-of-the-art search-based approaches, despite attempting to obtain a specialized policy against each opponent.

Experiments with linear function approximation (Section 5.4.2) address such limitations: the reward function is directly based on the match result and the linear approximation scheme yields a smoother generalization of values across states than the proposed state abstraction.

With multiple decision points, function approximation achieved superior performance with less training episodes than state aggregation. With function approximation, the importance of the features was decided by the agent, in form of their weights, whereas our state aggregation scheme did not allow the agent to decide the importance of the features, because we manually constructed the abstraction scheme. Thus, performance with state aggregation could improve if we automate the state abstraction scheme using, for example, unsupervised state clustering techniques. This way, we could have better insights on the trade-off between the granularity of the abstract states and the learning effort of the agent. With automated state aggregation, we could also employ more features to construct the aggregate states, allowing a fairer comparison with function approximation: our experiments with function approximation had an order of magnitude more features than state aggregation, allowing a richer representation of game states.

In our experiments with linear function approximation, the agent can select a different algorithm every game frame, but learns to commit to an algorithm (especially the combat-centered ones). In contrast, in [Sutton et al., 1999b], the agent commits to an option up to its termination by design, but can interrupt the option if there's an advantage on switching to a different option. These are complementary ways to learn the termination condition of options: our agent learns to commit until it is advantageous to switch, whereas the agent of Sutton et al. [1999b] commits by design and learns to interrupt when it is advantageous.

In terms of computational requirements, the decision procedures of the opponent-oblivious Q-learning and Sarsa is a fast ϵ-greedy over the Q-function. The Q-function is a constant-time lookup for tabular Q-learning and a fast linear combination for Sarsa with linear function approximation. Minimax-Q requires solving a linear programming to determine the safe policy prior to the ϵ-greedy procedure. To estimate the memory requirements, we focus on storing the relevant information for decision-making, which are the components of the Q-function. We disregard the storage of μRTS-related structures, as those are common to all approaches that play the game. Moreover, we consider floating point numbers with four bytes, as the methods are implemented in Java[20] to play μRTS. We then estimate the memory requirements of our implementations as follows:

- Minimax-Q (opponent-aware with state aggregation): it stores a 6×6 algorithm-value table for each of the 10935 non-terminal states (the number of states is

[20]`https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html`

calculated in Section 5.4.1). In total, it stores $36 \times 10935 = 393660$ floating-point numbers, corresponding to approximately 1.50 megabytes;

- Q-learning (opponent-oblivious with state aggregation): it stores 6 algorithm-value entries for each of the 10935 non-terminal states. In total, it stores $6 \times 10935 = 65610$ floating-point values, corresponding to approximately 256.29 kilobytes;
- Sarsa (opponent-oblivious with linear function approximation): it stores 130 weights for each of the 6 algorithms, plus 130 feature values for the current state. In total, it stores $6 \times 130 + 130 = 910$ floating-point values, corresponding to approximately 3.55 kilobytes.

Experiments in this section were executed in a 40-core computer with 256GB of RAM memory. Each core is a Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz. Our methods are not CPU-intensive nor require large amounts of memory, but this infrastructure allowed the execution of many experiments in parallel, speeding up the collection of results. Moreover, this infrastructure benefited the search approaches, as they are CPU-intensive and require considerable memory to store the search trees. As an example in terms of processing, training our approaches with state aggregation in self-play for 120 thousand episodes takes about the same period as training against AHTN for 5 thousand episodes (about 24 hours).

The opponent-oblivious experiments with state aggregation are published in [Tavares and Chaimowicz, 2018] and the ones with linear function approximation are published in [Tavares et al., 2018a].

5.5 Summary

This chapter presented experiments investigating various aspects of our strategic reasoning framework.

First, we illustrated the benefits of learning over algorithms with limited training iterations in synthetic problems (Section 5.1). We proceeded with the simplest, one-shot algorithm selection model in Section 5.3, focusing on game-theoretic aspects and the implementation of a functional StarCraft bot (MegaBot). We analyzed safe algorithm selection policies and safe opponent exploitation techniques applied to real-time strategy games. However, these aspects hold if the opponent is an algorithm selector with known portfolio, respecting the one-shot decision process. In general, a player tends to switch its course of action to improve its situation during the game.

The one-shot decision model is an extreme simplification of the underlying game's state space (i.e. the primitive states): all non-terminal primitive states are mapped to a single abstract state. To enable richer behaviors, we considered multiple decision points, that is, more abstract states in Section 5.4. We first tested state aggregation (Section 5.4.1), selecting features and discretizing their values. In this model, our agent becomes an option-selection [Sutton et al., 1999a] agent, where each option starts in an abstract state, follows the corresponding algorithm's policy, and finishes on a different abstract state.

We tested opponent-aware and opponent-oblivious agents. The opponent-aware minimax-Q agent was outperformed by the search opponent, albeit showing more robustness than the opponent-oblivious agent, that is, better performance when trained against one opponent and tested against another. However, when tested against the specific training opponent, the opponent-oblivious Q-learning was competitive against some state-of-the-art search methods in μRTS, although it showed limitations suggesting the need of more training episodes, or a better learning generalization scheme.

We proceeded by testing linear function approximation in Section 5.4.2, which allows a smoother generalization of learning across primitive states than state aggregation. With less training episodes, function approximation achieved better performance than state aggregation: our Sarsa(0) agent consistently outperforms the state-of-the-art search approaches in μRTS. Moreover, the resulting policies are more robust, consistently winning opponents the agent has not trained against.

The better performance of function approximation is probably due to better generalization and a more flexible decision process. For example, state aggregation forces the agent to hold its current algorithm until the abstract state changes. With function approximation, the switching point is up to the agent, according to the value estimates it has for the current primitive state. In other words, the agent can "interrupt" the option execution, having a better control of its behavior. This is specially useful in adversarial scenarios, where the opponent might demand a quick player reaction, which could be stuck, unable to act, in a state aggregation scheme.

Our experiments allows us to verify the adherence of the framework to the guidelines (Section 1.1): we tested Q-learning, minimax-Q and Sarsa, which are model-free reinforcement learning approaches, satisfying guideline **(G1)**; Q-learning, minimax-Q and Sarsa's decision procedures is fast in comparison with the search opponents and require low memory with our state aggregation and linear function approximation schemes, satisfying guideline **(G2)**; and are able to play entire matches of either StarCraft or μRTS, rather than specific parts such as combats, satisfying guideline **(G3)**.

Results of this chapter were published as follows:

- Single decision point: [Tavares et al., 2016] (stationary) and [Tavares et al., 2018b] (stationary, non-stationary and MegaBot's performance);
- Multiple decision points:
  - State-aggregation: [Tavares and Chaimowicz, 2018][21] (opponent-oblivious);
  - Function approximation: [Tavares et al., 2018a].

Overall, reasoning over algorithms/options instead of primitive actions greatly simplifies the number of choices to consider in each state. However, to succeed in complex domains with huge state spaces, such as real-time strategy games, an efficient learning generalization scheme is essential. In our experiments, linear function approximation showed the best results. Thus, learning over algorithms with a function approximation scheme has an interesting potential in real-time strategy games.

[21]In this paper, Monte Carlo Algorithm Selection (MCAS) is referred to as Strategy Simulation (SS).

Chapter 6

Conclusion

This chapter presents an overview of the dissertation and discusses contributions, limitations, directions for future research and a reflection about the purpose of artificial intelligence research on games.

6.1 Overview

This dissertation addressed research challenges posed by complex computer games: huge action and state spaces, real-time interaction, competition, simultaneous actions, and long-term decision effects. Real-time strategy games are a prominent representative of such games. We proposed three guidelines: **(G1)** play the game without its forward model (in contrast with search-based approaches); **(G2)** do not rely on massive hardware infrastructure (e.g. hundreds of GPU and thousand of CPU cores used in recent deep learning approaches); and **(G3)** play entire matches of the game, not only specific portions such as combats.

We propose a strategic reasoning framework, which is basically a two-level hierarchical architecture coupled with a reinforcement learning framework, where the upper layer assesses the game situation and selects an algorithm in the lower layer to perform low-level game actions. The algorithm's performance is used to update its value and improve future selections. We use two known learning generalization schemes to promote similar responses to similar game states: state aggregation and linear function approximation. Our approach is inspired by human behavior, where previously trained courses of actions are recalled and responses to familiar situations are adapted to similar, albeit unfamiliar ones.

We investigate whether it is better to select algorithms or low-level actions, according to the number of actions in the domain, the number of algorithms in the portfolio, and their strength, via experiments in a synthetic problem. Our findings confirm the intuition that it is better to select algorithms when their quality or the number of low-level actions grow. That is, the action selector takes time to explore the action space, whereas the algorithm selector accumulates rewards in the meantime. However, once the action selector discover the best action, it eventually outperforms the algorithm selector as it is limited to its best algorithm (which might not always select the best action).

We instantiate our framework in two real-time strategy games: StarCraft and μRTS, first to evaluate game-theoretic aspects of algorithm selection and, second, to evaluate our performance against state-of-the-art search-based approaches. Regarding the game-theoretic aspects, we observed that insights from computer rock-paper-scissors tournaments also apply in real-time strategy games: deviating from pessimistic algorithm selection policies in equilibrium is useful to exploit weak opponents, although one must resort to equilibrium policies to avoid being exploited by a stronger opponent. Regarding game-playing performance, a state aggregation scheme for learning generalization for a Q-learning agent achieved competitive performance against some state-of-the-art search-based opponents, whereas a Sarsa agent with a linear function approximation scheme consistently defeated all of them.

6.2 Contributions

6.2.1 A model-free, lightweight, full-game capable approach

Our algorithm selection framework is a hierarchical architecture that leverages the performance of reinforcement learning approaches, when combined with linear function approximation. Besides, our framework does not require a game's forward model, demands low computational resources and is able to play entire matches of complex computer games, satisfying our design guidelines, as detailed next:

(G1) **An agent must play without resorting to the game's forward model:** we model algorithm selection within a reinforcement learning framework, using Q-learning, minimax-Q and Sarsa, which are model-free approaches. Our agents discover the value of the choices by actually interacting with the game, instead of simulating the outcomes by querying the game's engine. In this sense, it is easier to instantiate our approach in a broader range of games: our approach requires existing game-playing algorithms and a programming interface to read the game state and issue the commands. Search-based approaches require more complex programming infrastructure: either the programming interface has to support the generation of "fictitious" states to simulate the action outcomes and then effect the current game state with the actually issued actions, or the game's source code must be available so that the search approach designer can copy the game instance and simulate the remaining match from the current state.

(G2) The approach must not depend on powerful hardware: the decision procedures of Q-learning, minimax-Q and Sarsa is orders of magnitude faster than that of the search-based opponents. Besides, the state aggregation and linear function approximation schemes have low memory requirements. State aggregation stores the Q-function in tabular form, albeit with few entries as the number of abstract states is much less than the number of primitive states. Function approximation requires even less memory to store the feature's values and their respective weights. Moreover, with linear function approximation, relatively small training sessions, with hundreds of episodes, were enough to outperform the state-of-the-art search-based opponents.

(G3) The approach must handle all aspects of the underlying game: we used full-game-playing algorithms in our portfolio to play either StarCraft or μRTS matches, rather than specific (e.g. combat-centered) ones. This way, our approach plays entire matches of those games, being able to participate in artificial intelligence tournaments.

6.2.2 A metagame analysis

In gaming terminology, metagame refers to reasoning over aspects beyond the game[1]. It usually involves the study of how strategies (courses of actions) interact or a player's preferred style. An example is to look for past games of a specific player, detect his/her usual opening strategies and prepare a counter-strategy in the next match.

Our game of algorithm selection, instantiated in Section 4.4.1 and tested in Section 5.3, allowed us to observe how StarCraft game-playing algorithms interact. Our competition bot, tested in Section 5.3.3, made use of this mechanism, by attempting to discover the best algorithm in its portfolio to defeat its opponent.

Metagame analysis is useful not only to derive game-playing policies, but in game-balancing as well. For example, Liu and Marschner [2017] balance games by assigning advantages or handicaps to previously weak or strong strategies, respectively.

[1]`https://liquipedia.net/starcraft/Metagame`

6.2.3 An investigation of abstract versus low-level actions

Reinforcement learning methods have asymptotic guarantees of convergence to the optimal action-selection policy. However, in practice, it is often unfeasible to satisfy the required conditions (see Section 2.3.2).

On the other hand, hierarchical decision-making frameworks are useful to achieve a reasonable performance faster at the potential sacrifice of optimality.

In this dissertation we shed more light on the dilemma of learning over abstract versus low-level actions. In our case the abstract actions are our algorithms. Our findings in synthetic experiments (see Section 5.1) confirm the intuition that it is better to select algorithms when their quality or the number of low-level actions grow.

In our experiments with real-time strategy games, the number of actions is far greater than the number of training episodes, further justifying the adoption of our learning over algorithms approach.

6.2.4 Advancement in RTS game AI performance

Recent advances in computer games have been achieved through deep reinforcement learning architectures, which require long training sessions (hundreds of millions of episodes [Mnih et al., 2015]), sometimes in clusters with hundreds of GPU and thousands of CPU [OpenAI, 2018a].

Our strategic reasoning framework combined with linear function approximation, a well-known learning generalization scheme, consistently defeated state-of-the-art search approaches in μRTS (see Section 5.4.2), training for at most 500 episodes in much more accessible hardware infrastructure.

6.2.5 Software contributions

Our main software contribution is a tournament-capable StarCraft bot, MegaBot[2]. It implements our game of algorithm selection and uses a portfolio of three StarCraft bots. MegaBot succeeded in its debut year, by ranking among the top 50% competitors, but its success decreased in the following years. Nevertheless, it exhibited interesting learning curves, suggesting that its victories decreased because none of its portfolio members could defeat the newer and stronger bots that appeared. Those results appear in Section 5.3.3 and MegaBot's implementation details are shown in Appendix B.

[2]`https://github.com/andertavares/MegaBot`

Other software contributions include: our game of algorithm selection tournament engine[3], the engine for synthetic experiments on learning over algorithms versus actions[4], and our μRTS implementations with state aggregation[5] and linear function approximation[6].

We have also provided a modified version of the old StarCraft AI Tournament manager, which disables bots' learning capabilities[7]. Newer versions of the software[8] now provide this feature.

6.3 Limitations

6.3.1 Rigid architecture

Our strategic reasoning framework is rigid as it depends on existing algorithms. We do not consider the idea of creating a new sequence of actions to achieve a specific goal, or to modify an existing algorithm to potentially improve its performance.

Our architecture is also rigid in the number of layers. It is possible that multi-layer architectures achieve even more abstract representations to handle strategic, tactical and reactive aspects of real-time strategy games naturally.

6.3.2 Perfect information

An important aspect of most complex computer games, including real-time strategy games, is the fog-of-war: a player has access to information within its units visual range. In our StarCraft experiments (Section 5.3), we do not deal with fog-of-war, as our algorithm selection process is one-shot, without context, so that our agents do not look at the game state to make choices. Imperfect information is handled by the selected algorithms, which were programmed to do so.

However, in our μRTS experiments, the agents can switch algorithms during a game, based on the current state. Our approach and the test opponents require perfect information, and we enable it in the game.

[3]`https://github.com/DanielKneipp/StarcraftNash`
[4]`https://github.com/andertavares/syntheticmdps`
[5]`https://github.com/amandaccsantos/microrts`
[6]`https://github.com/SivaAnbalagan1/micrortsFA`
[7]`https://github.com/andertavares/StarcraftAITournamentManager`
[8]`https://github.com/davechurchill/StarcraftAITournamentManager`

6.4 Directions for future research

Future research could deepen our understanding on some aspects we identified in our experiments as well as address some of the limitations of our approach in this dissertation.

6.4.1 Better insights from the experiments

In our experiments with linear function approximation (Section 5.4.2), the agent learned when to activate and switch between algorithms. In other words, it learned the initiation and termination conditions of the options [Sutton et al., 1999b]. A further investigation can draw interesting insights on how the initiation and termination conditions of options in complex tasks can be learned.

The experiments with state aggregation could also provide insights in this direction. Our state aggregation scheme is handcrafted as we partition the state space by hand to generate the abstract states. Moreover, we forced the option to terminate when the agent reaches a new abstract state. An automatic or adaptive state aggregation scheme could result in a better partitioning of the state space, generating more useful options with better termination conditions. A research challenge in this sense is to derive metrics to evaluate the quality of the state aggregation scheme.

State aggregation is a form of non-linear function approximation. Learning can diverge when function approximation is combined with an off-policy methods such as Q-learning [Sutton and Barto, 1998, Section 8.5]. It might be interesting to investigate under which conditions, regarding either the state aggregation scheme, the characteristics of the scenario (e.g. map type and size) or the learning algorithm, are “safe” in terms of learning convergence.

Our experiments with multiple decision points (Section 5.4) were executed in a single μRTS map. It would be interesting to evaluate the robustness of the approach in different maps. Moreover, our experiments have human bias in the construction of the state aggregation scheme and in the definition of the features for linear function approximation. Deep reinforcement learning techniques [Mnih et al., 2015] could be employed to handle raw state representations and remove human bias, although they would require more hardware infrastructure for training.

6.4.2 More flexible hierarchical architectures

Learning or improving low-level behaviors is desirable. In this sense, Kulkarni et al. [2016] presents a two-level hierarchical framework, where the upper layer observes the environment state and chooses a goal, generating intrinsic rewards for the lower layer to learn a policy to achieve that goal. Later, Machado et al. [2017] presents an approach to discover the goals, rather than having them handcrafted.

However, in complex computer games, learning a low-level behavior means to handle the large action spaces of such games - a problem our strategic reasoning framework does not touch. Learning low-level behaviors in complex games is an open research problem, with OpenAI [2018a] showing some advances with a multi-step action selection: first select a type (e.g. attack or move), then a parameter (e.g. target or direction), for example. Low-level behavior could also be learned through the extraction of patterns in human gameplay.

A promising line of research is to use action abstractions [Hawkin et al., 2011], as in Moraes and Lelis [2018], where scripts filter out the actions to be considered by their search algorithm in each state. With reinforcement learning, the scripts or heuristics could be used to filter the actions whose values would be learned.

A dynamic approach to instantiate layers of reasoning in a task might lead to richer behaviors and a better understanding of the structure of the problem. Dynamically instantiating new layers of reasoning besides determining their behavior is an ongoing topic or research. See, for example, Digney [1998]; Barto and Mahadevan [2003]; Barry et al. [2011].

6.4.3 Handling imperfect and incomplete information

Dealing with partial observation or imperfect information (uncertainty regarding the current state) requires inferring the attributes of entities out of sight. Although there is a large body of research on reinforcement learning on partially observable domains[9], and games with imperfect information such as poker are being successfully tackled by AI techniques [Bowling et al., 2015], this topic has not gained much attention in real-time strategy games, with the exception of Weber et al. [2011].

Our algorithm selection model against general opponents (Section 4.3.2) enumerates all possible opponent actions, and we acknowledge that such enumeration is not feasible in practice. In our experiments (Section 5.4.1.2 and Section 5.4.2), we adopt an opponent-oblivious agent, which embeds the opponent in the environment.

A possible way of modeling the opponent without enumerating its actions would be to classify him into a pool of known behaviors and to react properly, as in the μRTS bot of Silva et al. [2018]. In StarCraft, game actions are classified into behaviors via replay analysis in [Weber and Mateas, 2009; Synnaeve and Bessiere, 2011; Stanescu and

[9]A highly cited survey on the topic dates back to 1982 [Monahan, 1982].

Čertickỳ, 2016]. A possible extension of those approaches would incorporate them in a StarCraft bot to work on live matches rather than performing offline replay analysis.

Modeling the problem as a game of incomplete information allows considering the opponent without enumerating its actions. In this model, the agent is aware of the opponent's presence, but does not know its actions, nor how it affects the environment transitions and rewards (hence the incomplete information). A possible approach would be to extend the adversarial multi-armed bandit model [Auer et al., 1995] to multi-stage decision problems. In the adversarial multi-armed bandit, the agent is aware that an opponent can shuffle the bandit's rewards and, without further assumptions on the opponent, the Exp3 method of Auer et al. [1995] is able to derive theoretically guaranteed arm-selection policies. We tested the Exp3 method in our one-shot decision experiments (Section 5.3), with interesting results. However, it is not clear on how the method could be extended to accommodate the values of future states while keeping its performance guarantees.

Another possible approach to consider the opponent without enumerating its actions is to model the problem as a Markov Decision Process with combined transition and reward functions. An example would be the additive reward, additive transition (AR-AT) games of Filar and Vrieze [2012], where each agent has its component to the functions. For example, the transition function is written as: $T(s, a_1, a_2, s') = T_1(s, a_1, s') + T_2(s, a_2, s')$, where T_i is the component of player i. However, the additive models are not general. As the components are probabilities, they cannot be negative, so that the resulting transition can never be smaller than its individual components.

In general, the opponent's action can reduce the chance of a state being encountered and increase the chance of another one happening, and a perhaps more general model would include a "disturbed transition function" $T(s, a_1, s') = T_1(s, a_1, s') + \xi$, from the point of view of player 1. In this function, the opponent's part is a "noise", which can assume any value. Modeling this noise implies modeling the opponent's behavior, and that could be an interesting research direction.

6.5 The purpose of game AI research

The recognition of games as a legitimate research subject has grown in recent years. The field has witnessed the growth of papers about AI in games on major AI research venues, such as the IJCAI, AAAI and NeurIPS conferences, as well as the growth of dedicated conferences such as the IEEE Conference on Computational Intelligence and Games (CIG) and the AAAI Artificial Intelligence and Interactive Digital Entertainment (AIIDE), and the establishment of a dedicated journal, the IEEE Transactions on Games (ToG - former Transactions on Computational Intelligence and AI in Games) [Yannakakis and Togelius, 2015]. Occasionally, however, the validity of doing research in games as a means of advancing science is questioned. Arguments sometimes mention that games are too distant from the real world, or that games are an entertaining activity, such that research on games is a purposeless rather than a serious activity.

Games may indeed be simplified or idealized representations of the real world. However, the empirical evaluation of many traditional algorithms (e.g. [Littman, 1994; Dietterich, 2000]) is performed on synthetic environments, often simpler than traditional games, created in the specific paper that presented the algorithm. This fact does not diminish the contributions that the algorithms present to the advancement of science, for they are the materialization of novel ideas, approaching theoretical or practical matters that were not touched before. Moreover, an algorithm that succeeds when tested on a game can be considered even more "solid": games, although being simpler than the real world, were built for humans, rather than machines, to play. That is, instead of building a synthetic testbed to validate the algorithm, an existing testbed is used; one where, initially, an algorithm was not supposed to tackle. In the words of Richard Sutton: "in practice, games end up being more real than anything we make up"[10].

It is true that doing research on games can be as enjoyable as playing a game. A book chapter on computers and games co-authored by Alan Turing even mentions that "it would be disingenuous of us to disguise the fact that the principal motive which prompted the work was the sheer fun of the thing" [Bates et al., 1953]. However, the enjoyment on game research does not impoverish the scientific contributions presented on the subject. For example, the increasingly complex and realistic game scenarios are benefiting the research on autonomous vehicles [Knight, 2016]. Besides, algorithmic advances in research on games as a general framework for multiagent decision-making can have serious implications in the areas of security, medical decision support and even war politics [Bowling et al., 2015]. Ultimately, the very book chapter co-authored by Alan Turing mentions that the research on games could be justified as a means to, among other things, tackle "the incompetence of the machine at taking an overall view of the problem which it is analysing" [Bates et al., 1953]. The work on this dissertation has the same spirit of that book chapter. It was performed with a keen

[10]Sutton is one of the pioneers of reinforcement learning, co-authoring the essential book on the field [Sutton and Barto, 1998]. The cited phrase was Sutton's response to a question posed during Gerry Tesauro's talk on reinforcement learning and games at the 2009 Multidisciplinary Symposium on Reinforcement Learning. The question starts at about 58 minutes of the video: `http://videolectures.net/icml09_tesauro_itfyrlg/`

interest and engagement towards the subject, and also with the noble goal of reducing the “incompetence of the machine”, because what we named “strategic reasoning” is a form of taking “an overall view of the problem” under analysis.

Bibliography

Abreu, D. and Rubinstein, A. (1988). The structure of Nash equilibrium in repeated games with finite automata. *Econometrica*, 56(6):1259--1281.

Aha, D. W., Molineaux, M., and Ponsen, M. (2005). Learning to win: Case-based plan selection in a real-time strategy game. In *Case-based reasoning research and development*, pages 5--20. Springer.

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multi-armed bandit problem. *Machine learning*, 47(2-3):235--256.

Auer, P., Cesa-Bianchi, N., Freund, Y., and Schapire, R. E. (1995). Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *Foundations of Computer Science. Proceedings, 36th Annual Symposium on*, pages 322--331. IEEE.

Barriga, N. A., Stanescu, M., and Buro, M. (2015). Puppet Search: Enhancing Scripted Behavior by Look-Ahead Search with Applications to Real-Time Strategy Games. In *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE)*.

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

Barry, J. L., Kaelbling, L. P., and Lozano-Pérez, T. (2011). DetH*: Approximate Hierarchical Solution of Large Markov Decision Processes. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1928--1935.

Barto, A. G. and Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341--379. ISSN 0924-6703.

Bates, A. M., Bowden, B. V., Strachey, C., and Turing, A. M. (1953). Digital computers applied to games. In Bowden, B. V., editor, *Faster Than Thought*, chapter 25, pages 286--290. Sir Isaac Pittman & Sons, London.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2012). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*.

Billlings, D. (1999). First International RoShamBo programming competition. Available at: `https://webdocs.cs.ualberta.ca/~darse/rsbpc1.html`. Accessed in 15/11/2016.

Blizzard (2016). Blizzard Entertainment: StarCraft. `http://us.blizzard.com/en-us/games/sc/`.

Bontrager, P., Khalifa, A., Mendes, A., and Togelius, J. (2016). Matching Games and Algorithms for General Video Game Playing. In *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE)*, pages 122--128.

Bošanskỳ, B., Lisỳ, V., Lanctot, M., Čermák, J., and Winands, M. H. (2016). Algorithms for computing strategies in two-player simultaneous move games. *Artificial Intelligence*, 237:1--40.

Bowling, M., Burch, N., Johanson, M., and Tammelin, O. (2015). Heads-up limit hold'em poker is solved. *Science*, 347(6218):145--149.

Brown, G. W. (1951). Iterative Solutions of Games by Fictitious Play. In Koopmans, T. C., editor, *Activity Analysis of Production and Allocation*. Wiley.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, 4(1):1--43.

Buro, M. and Furtak, T. (2004). RTS games and real-time AI research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS)*, volume 6370.

BWAPI (2015). BWAPI: An API for interacting with Starcraft: Broodwar. `http://bwapi.github.io`.

Carmel, D. and Markovitch, S. (1996). Learning Models of Intelligent Agents. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 62--67.

Churchill, D. (2014). StarCraft AI Competition - Tournament Manager Software. Available at: `http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/tm.shtml`. Accessed in 25/11/2016.

Churchill, D. and Buro, M. (2013). Portfolio Greedy Search and Simulation for Large-Scale Combat in StarCraft. In *IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1--8. IEEE.

Churchill, D. and Buro, M. (2015). Hierarchical Portfolio Search: Prismata's Robust AI Architecture for Games with Large Search Spaces. In *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE)*, pages 16--22.

Churchill, D., Buro, M., and Barriga, N. (2015). StarCraft AI Competition Results. Available at: `http://www.cs.mun.ca/~dchurchill/starcraftaicomp/2015/`. Accessed in 25/10/2016.

Churchill, D., Saffidine, A., and Buro, M. (2012). Fast Heuristic Search for RTS Game Combat Scenarios. In *AIIDE*.

Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2001). *Introduction to Algorithms*. MIT Press, Cambridge, Mass. ISBN 0070131511.

Cunha, R. L. de. F. and Chaimowicz, L. (2010). An Artificial Intelligence System to Help the Player of Real-Time Strategy Games. In *Proceedings of the 2010 Brazilian Symposium on Games and Digital Entertainment (SBGAMES)*, pages 71 –81. ISSN 2159-6654.

Dahlbom, A. and Niklasson, L. (2006). Goal-Directed Hierarchical Dynamic Scripting for RTS Games. In *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE)*, pages 21--28.

Dietterich, T. G. (2000). Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, 13:227--303. ISSN 10769757.

Digney, B. L. (1998). Learning Hierarchical Control Structures for Multiple Tasks and Changing Environments. *From Animals to Animats 5: Proceedings of the Fifth International Conference on the Simulation of Adaptive Behavior*, 5:321--330.

Fekete, S. P., Fleischer, R., Fraenkel, A., and Schmitt, M. (2004). Traveling salesmen in the presence of competition. *Theoretical Computer Science*, 313(3):377--392.

Filar, J. and Vrieze, K. (2012). *Competitive Markov decision processes*. Springer Science & Business Media.

Furtak, T. and Buro, M. (2010). On the Complexity of Two-Player Attrition Games Played on Graphs. In *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE)*, pages 113--119.

Gasser, R. (1996). Solving nine men's morris. *Computational Intelligence*, 12(1):24--41.

Genesereth, M. and Thielscher, M. (2014). General game playing. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(2):1--229.

Hausknecht, M. and Stone, P. (2015). Deep Recurrent Q-learning for Partially Observable MDPs. *arXiv preprint arXiv:1507.06527*.

Hawkin, J. A., Holte, R., and Szafron, D. (2011). Automated Action Abstraction of Imperfect Information Extensive-Form Games. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 681--687.

Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaśkowski, W. (2016). ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning. *arXiv preprint arXiv:1605.02097*.

Knight, W. (2016). Self-Driving Cars Can Learn a Lot by Playing Grand Theft Auto. Available at: `https://www.technologyreview.com/s/602317/self-driving-cars-can-learn-a-lot-by-playing-grand-theft-auto/`. Accessed in 03/04/2018.

Knuth, D. E. and Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293--326.

Kulkarni, T. D., Narasimhan, K., Saeedi, A., and Tenenbaum, J. (2016). Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation. In *Advances in Neural Information Processing Systems (NIPS)*, pages 3675--3683.

Lagoudakis, M. G. and Littman, M. L. (2000). Algorithm selection using reinforcement learning. In *ICML*, pages 511--518.

Lample, G. and Chaplot, D. S. (2016). Playing FPS games with Deep Reinforcement Learning. *arXiv preprint arXiv:1609.05521*.

Le Cun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1990). Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278--2324.

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

Levine, J., Congdon, C. B., Ebner, M., Kendall, G., Lucas, S. M., Miikkulainen, R., Schaul, T., and Thompson, T. (2013). General Video Game Playing. *Dagstuhl Follow-Ups*, 6.

Li, J. and Kendall, G. (2015). A hyper-heuristic methodology to generate adaptive strategies for games. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG).*

Li, L., Walsh, T. J., and Littman, M. L. (2006). Towards a Unified Theory of State Abstraction for MDPs. In *Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics*, pages 531--539.

Liang, Y., Machado, M. C., Talvitie, E., and Bowling, M. H. (2016). State of the Art Control of Atari Games Using Shallow Reinforcement Learning. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 17--25.

Liquipedia (2012). Portal:Beginners - Liquipedia StarCraft Brood War Wiki. Liquipedia - 16/04/2012. Available at: `http://wiki.teamliquid.net/starcraft/Portal:Beginners`. Accessed in 24/11/2016.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 157--163, New Brunswick, NJ. Morgan Kaufmann.

Littman, M. L. (1996). *Algorithms for sequential decision making.* PhD dissertation, Brown University.

Liu, A. and Marschner, S. (2017). Balancing zero-sum games with one variable per strategy. In *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE)*, pages 57--65.

Machado, M. C., Bellemare, M. G., and Bowling, M. H. (2017). A Laplacian Framework for Option Discovery in Reinforcement Learning. In *International Conference on Machine Learning (ICML)*, pages 2295--2304.

Mattson, M. P. (2014). Superior pattern processing is the essence of the evolved human brain. *Frontiers in neuroscience*, 8:265.

McCracken, P. and Bowling, M. (2004). Safe strategies for agent modelling in games. *AAAI Fall Symposium on Artificial Multi-agent Learning*, pages 103--110.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529--533.

Monahan, G. E. (1982). State of the art - a survey of partially observable Markov decision processes: theory, models, and algorithms. *Management Science*, 28(1):1--16.

Moraes, R. O. and Lelis, L. H. (2018). Asymmetric Action Abstractions for Multi-Unit Control in Adversarial Real-Time Games. In *AAAI Conference on Artificial Intelligence (AAAI).*

Nisan, N., Roughgarden, T., Tardos, E., and Vazirani, V. V. (2007). *Algorithmic Game Theory*, volume 1. Cambridge University Press Cambridge.

Ontanón, S. (2013). The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE).*

Ontañón, S. (2017). Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research*, 58:665--702.

Ontañón, S. and Buro, M. (2015). Adversarial Hierarchical-Task Network Planning for Complex Real-Time Games. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1652--1658.

Ontañón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., and Preuss, M. (2013). A survey of real-time strategy game AI research and competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, 5(4):293--311.

OpenAI (2017). More on Dota 2. OpenAI Blog - 16/aug/2018. Available at: `https://blog.openai.com/more-on-dota-2/`. Accessed in 10/jul/2018.

OpenAI (2018a). OpenAI Five. OpenAI Blog - 25/jun/2018. Available at: `https://blog.openai.com/openai-five/`. Accessed in 06/jul/2018.

OpenAI (2018b). The International 2018: Results. OpenAI Blog - 23/ago/2018. Available at: `https://blog.openai.com/the-international-2018-results/`. Accessed in 17/dec/2018.

Osborne, M. J. and Rubinstein, A. (1994). *A Course in Game Theory*. The MIT Press. ISBN 0262650401.

Peng, P., Yuan, Q., Wen, Y., Yang, Y., Tang, Z., Long, H., Wang, J., and Group, A. (2017). Multiagent Bidirectionally-Coordinated Nets for Learning to Play StarCraft Combat Games. Technical report.

Preuss, M., Kozakowski, D., Hagelback, J., and Trautmann, H. (2013). Reactive strategy choice in StarCraft by means of Fuzzy Control. In *IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1--8. IEEE.

Rice, J. R. (1976). The algorithm selection problem. *Advances in computers*, 15:65--118.

Rubinstein, R. (1999). The cross-entropy method for combinatorial and continuous optimization. *Methodology and computing in applied probability*, 1(2):127--190.

Rummery, G. A. and Niranjan, M. (1994). On-line Q-learning using connectionist systems. Technical report, University of Cambridge, Department of Engineering.

Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Upper Saddle River, N.J, second edition.

Sailer, F., Buro, M., and Lanctot, M. (2007). Adversarial planning through strategy simulation. In *IEEE Conference on Computational Intelligence and Games (CIG)*, pages 80--87. IEEE.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210--229.

Samuel, A. L. (1967). Some studies in machine learning using the game of checkers. II-recent progress. *IBM Journal of research and development*, 11(6):601--617.

Savani, R. and von Stengel, B. (2015). Game Theory Explorer: software for the applied game theorist. *Computational Management Science*, 12(1):5--33.

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. (2007). Checkers is solved. *Science*, 317(5844):1518--1522.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal Policy Optimization Algorithms. *CoRR*, abs/1707.06347.

Shapley, L. S. (1953). Stochastic games. *Proceedings of the National Academy of Sciences*, 39(10):1095--1100.

Shoham, Y. and Leyton-Brown, K. (2009). *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press.

Sigurdson, D. and Bulitko, V. (2017). Deep Learning for Real-Time Heuristic Search Algorithm Selection. In *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE)*, pages 108--114.

Silva, C. R., Moraes, R. O., Lelis, L. H. S., and Gal, K. (2018). Strategy generation for multi-unit real-time games via voting. *IEEE Transactions on Games*. ISSN 2475-1502.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484--489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. (2017a). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR*, abs/1712.01815.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., and Sifre, L. (2017b). Mastering the game of Go without human knowledge. *Nature*, 550(7676):354--359. ISSN 0028-0836.

Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I., and Postma, E. (2006). Adaptive game AI with dynamic scripting. *Machine Learning*, 63(3):217--248.

Stanescu, M., Barriga, N. A., and Buro, M. (2014). Hierarchical Adversarial Search Applied to Real-Time Strategy Games. In *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE)*, pages 66--72.

Stanescu, M. and Čertický, M. (2016). Predicting Opponent's Production in Real-Time Strategy Games With Answer Set Programming. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, 8(1):89--94.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Sutton, R. S., Precup, D., and Singh, S. (1999a). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181--211.

Sutton, R. S., Singh, S. P., Precup, D., and Ravindran, B. (1999b). Improved Switching Among Temporally Abstract Actions. In *Advances in Neural Information Processing Systems*, pages 1066--1072.

Świechowski, M. and Mańdziuk, J. (2014). Self-adaptation of playing strategies in general game playing. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, 6(4):367--381.

Synnaeve, G. and Bessiere, P. (2011). A bayesian model for opening prediction in RTS games with application to StarCraft. In *IEEE Conference on Computational Intelligence and Games (CIG)*, pages 281--288. IEEE.

Szita, I. and Lõrincz, A. (2007). Learning to Play Using Low-Complexity Rule-Based Policies: Illustrations through Ms. Pac-Man. *Journal of Artificial Intelligence Research*, 30:659--684.

Tavares, A., Azpúrua, H., Santos, A., and Chaimowicz, L. (2016). Rock, Paper, StarCraft: Strategy Selection in Real-Time Strategy Games. In *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE)*, pages 93--99.

Tavares, A. R., Anbalagan, S., Marcolino, L. S., and Chaimowicz, L. (2018a). Algorithms or Actions? A Study in Large-Scale Reinforcement Learning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2717--2723. International Joint Conferences on Artificial Intelligence Organization.

Tavares, A. R. and Chaimowicz, L. (2018). Tabular Reinforcement Learning in Real-Time Strategy Games via Options. In *IEEE Conference on Computational Intelligence and Games (CIG)*, pages 229--236.

Tavares, A. R., Vieira, D. K. S., Negrisoli, T., and Chaimowicz, L. (2018b). Algorithm Selection in Adversarial Settings: From Experiments to Tournaments in StarCraft. *IEEE Transactions on Games*.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58--68.

Toubman, A., Roessingh, J. J., Spronck, P., Plaat, A., and van den Herik, J. (2016). Rapid Adaptation of Air Combat Behaviour. In *Prestigious Applications of Intelligent Systems (PAIS)*.

Uriarte, A. and Ontañón, S. (2014). Game-tree Search over High-level Game States in RTS Games. In *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE)*, pages 73--79.

Uriarte, A. and Ontañón, S. (2016). Improving Monte Carlo Tree Search Policies in StarCraft via Probabilistic Models Learned from Replay Data. In *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE)*, pages 100--106.

Usunier, N., Synnaeve, G., Lin, Z., and Chintala, S. (2016). Episodic Exploration for Deep Deterministic Policies: An Application to StarCraft Micromanagement Tasks. Technical report.

Verwey, W. B. and Abrahamse, E. L. (2012). Distinct modes of executing movement sequences: reacting, associating, and chunking. *Acta psychologica*, 140(3):274--282.

Wang, C., Chen, P., Li, Y., Holmgård, C., and Togelius, J. (2016). Portfolio Online Evolution in StarCraft. In *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE)*, pages 114--120.

Wang, Z., Xu, B., and Zhou, H.-J. (2014). Social cycling and conditional responses in the Rock-Paper-Scissors game. *Scientific Reports*, 4:5830.

Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279--292. ISSN 0885-6125.

Weber, B. G. and Mateas, M. (2009). A data mining approach to strategy prediction. In *IEEE Conference on Computational Intelligence and Games (CIG)*, pages 140--147. IEEE.

Weber, B. G., Mateas, M., and Jhala, A. (2011). A Particle Model for State Estimation in Real-Time Strategy Games. In *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE)*, pages 103--108.

Wender, S. and Watson, I. (2012). Applying Reinforcement Learning to Small Scale Combat in the Real-Time Strategy Game StarCraft:Broodwar. In *IEEE Conference on Computational Intelligence and Games (CIG)*, pages 402--408. ISSN 2325-4270.

Whitehead, S. D. and Lin, L.-J. (1995). Reinforcement learning of non-Markov decision processes. *Artificial Intelligence*, 73(1-2):271--306.

Xu, L., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2008). SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565--606.

Xu, L. D. (1994). Case based reasoning. *IEEE Potentials*, 13(5):10--13.

Yannakakis, G. N. and Togelius, J. (2015). A Panorama of Artificial and Computational Intelligence in Games. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):317--335.

Appendix A

Additional data on the game of algorithm selection

This chapter presents additional data on the game of algorithm selection experiments. Table A.1 shows the percent of victories among AIIDE 2015 bots on the Fortress map.

The expected performances of Table 5.2 were extracted from these values, as follows: 1·probability of victory −1·probability of defeat. The probabilities were estimated from the victory rates in Table A.1.

Bot	UAlb	Ximp	Xeln	Cruz	NUSB	Aiur	Skyn	Susa
UAlberta	-	**97**	**98**	**100**	**100**	**97**	**92**	**100**
Ximp	3	-	**100**	**97**	**97**	**86**	**75**	**100**
Xelnaga	2	0	-	26	**86**	**73**	**73**	**100**
CruzBot	0	3	**74**	-	**80**	**67**	16	**99**
NUSBot	0	3	14	20	-	**74**	**97**	**94**
Aiur	3	14	27	33	26	-	**79**	**100**
Skynet	8	25	27	**84**	3	21	-	**100**
Susanoo	0	0	0	1	6	0	0	-

Table A.1: Win percent among AIIDE 2015 Protoss bots on Fortress map. Values in bold indicate that the row bot dominates the column adversary, by winning more than 50% matches.

Table A.2 presents all pairings of algorithm selectors in the stationary algorithm selection tournament. The averages in the last column were used to generate Figure 5.6.

Table A.3 presents all pairings of algorithm selectors in the non-stationary algorithm selection tournament. The averages in the last column were used to generate Figure 5.7.

	Exp3	**Freq-hist**	**Freq-matrix**	**minimax-Q**	**Nash**	**Reply-matrix**	**Reply-hist**	**Skynet**	**UCB**	**Xelnaga**	**ϵ-greedy**	**ε-Nash**	**Average**
Exp3		51.54	50.99	49.60	49.94	50.28	49.38	85.01	50.63	71.38	50.62	50.15	55.41
Freq-hist	48.46		48.79	45.44	51.82	36.88	50.60	80.49	40.03	50.13	48.47	47.35	49.86
Freq-matrix	49.01	51.21		46.22	51.60	37.05	37.56	93.92	42.62	80.80	48.60	46.94	53.23
minimax-Q	50.40	54.56	53.78		49.24	50.86	49.50	92.21	51.95	78.43	52.08	49.35	57.49
Nash	50.06	48.18	48.40	50.76		52.54	49.53	46.86	49.76	53.55	49.87	49.32	49.89
Reply-matrix	49.72	63.12	62.95	49.14	47.46		50.31	93.97	70.75	80.74	62.66	50.64	61.95
Reply-hist	50.62	49.40	62.44	50.50	50.47	49.69		83.22	62.80	54.46	60.84	53.55	57.09
Skynet	14.99	19.51	6.08	7.79	53.14	6.03	16.78		25.24	31.27	20.41	33.75	21.36
UCB	49.37	59.97	57.38	48.05	50.24	29.25	37.20	74.76		76.28	52.78	50.61	53.26
Xelnaga	28.62	49.87	19.20	21.57	46.45	19.26	45.54	68.73	23.72		28.96	35.61	35.23
ϵ-greedy	49.38	51.53	51.40	47.92	50.13	37.34	39.16	79.59	47.22	71.04		49.81	52.23
ε-Nash	49.85	52.65	53.06	50.65	50.68	49.36	46.45	66.25	49.39	64.39	50.19		52.99

Table A.2: Pairings of the algorithm selection tournament with stationary payoff matrix

	Exp3	**Freq-hist**	**Freq-matrix**	**minimax-Q**	**Nash**	**Reply-matrix**	**Reply-hist**	**Skynet**	**UCB**	**Xelnaga**	**ϵ-greedy**	**ε-Nash**	**Average**
Exp3		43.53	85.52	49.50	73.54	82.26	44.58	66.80	45.54	71.80	48.34	74.72	62.37
Freq-hist	56.47		91.12	53.50	73.67	91.59	49.96	49.81	51.39	50.18	56.77	68.81	63.02
Freq-matrix	14.48	8.88		10.18	19.21	34.12	9.09	31.80	28.92	8.30	13.83	24.00	18.44
minimax-Q	50.50	46.50	89.82		74.86	83.30	45.88	73.17	48.35	80.28	50.79	74.58	65.28
Nash	26.46	26.33	80.79	25.14		64.48	27.79	25.63	25.31	45.46	37.17	62.53	40.64
Reply-matrix	17.74	8.41	65.88	16.70	35.52		8.49	31.85	25.51	8.27	17.04	30.13	24.14
Reply-hist	55.42	50.04	90.91	54.12	72.21	91.51		55.20	51.17	49.91	56.03	72.50	63.55
Skynet	33.20	50.19	68.20	26.83	74.37	68.15	44.80		29.62	76.10	33.48	71.76	52.43
UCB	54.46	48.61	71.08	51.65	74.69	74.49	48.83	70.38		75.12	54.53	74.54	63.49
Xelnaga	28.20	49.82	91.70	19.72	54.54	91.73	50.09	23.90	24.88		32.61	68.93	48.74
ϵ-greedy	51.66	43.23	86.17	49.21	62.83	82.96	43.97	66.52	45.47	67.39		70.52	60.90
ε-Nash	25.28	31.19	76.00	25.42	37.47	69.87	27.50	28.24	25.46	31.07	29.48		37.00

Table A.3: Pairings of the algorithm selection tournament with non-stationary payoff matrix

Appendix B

MegaBot

This chapter presents a more detailed description of MegaBot, evaluated in Section 5.3.3.

MegaBot is a fully-capable StarCraft bot designed to evaluate our algorithm selection framework in actual competition environments. The bot is composed of a meta-reasoning module, implementing the algorithm selection mechanism, and a portfolio of algorithms. The portfolio of MegaBot is composed by the AIIDE 2015 versions of Skynet, Xelnaga and NUSBot. They placed 8th, 9th and 13th, respectively, in that competition [Churchill et al., 2015]. In terms of tournament performance, they were not the strongest competitors. Thus, a good performance of MegaBot is due to its meta-reasoning mechanism rather than to a strong portfolio. Moreover, these three bots interact in a cyclical way: Skynet beats Xelnaga, which beats NUSBot, which beats Skynet, suggesting complementary abilities. This cycle does not appear in Figure 5.5 because it is related to a tournament with bots' learning disabled. The AIIDE 2015 tournament results in [Churchill et al., 2015] (with learning enabled) show the cyclical interaction.

The meta-reasoning module is basically an implementation of minimax-Q, reduced for one-shot decision scenarios (as in Section 5.3). We keep a payoff matrix with 3 lines and a column per opponent bot. The matrix is updated via Eq. 2.3, where we assign a score of 1 to victory and -1 to defeat. Ties are not considered, because they are broken by the in-game score in tournament matches.

Our selection mechanism is ϵ-greedy, as in the original implementation of minimax-Q [Littman, 1994]: select a random algorithm with probability ϵ, and select according to the equilibrium policy with probability $1 - \epsilon$. We exploit the fact that, in StarCraft tournaments, we have access to the name of the current opponent in a match, so we know the matrix column we are playing against. The equilibrium policy in this situation is to select the highest scoring algorithm from that column. This situation could also be framed as a contextual bandit [Sutton and Barto, 1998, Section 2.10]. In each match, we face a multi-armed bandit whose payoffs are determined by the matrix column related to the current opponent. ϵ-greedy over this bandit corresponds to our minimax-Q selection policy, which considers a single column in the payoff matrix.

MegaBot's source code[1] has the meta-reasoning module plus the code of all portfolio members in a single project. The source code of the members was edited in some places to handle naming conflicts. When MegaBot compiles, it generates a monolithic StarCraft AI client module, bundled with the meta-reasoning and the portfolio members. This monolithic AI client can be loaded and injected into the game normally via BWAPI [BWAPI, 2015].

MegaBot's parameters are α, the step size for the payoff matrix updates and ϵ, the exploration probability in the ϵ-greedy selection method. In all tournaments, MegaBot competes with $\alpha = 0.2$ and $\epsilon = 0.1$. In each tournament, the payoff matrix is initialized optimistically with ones to encourage initial exploration of all choices. We use a greater learning rate (α) than that of our experiments (see Section 5.3.1) because actual StarCraft tournaments have fewer matches than our experiments, thus we directed MegaBot to learn faster at the potential risk of being more affected by noisy outcomes, which could happen, for example, if our selected bot is winning but crashes.

In StarCraft tournaments, crashes and timeouts are punished with defeats. A crash is a bot malfunction that interrupts the game, and a timeout corresponds to a delay longer than tolerable to send a command to the game. There is a number of frames where the bot is allowed to timeout, and it is punished with defeat when it surpasses a tolerable limit. A disqualification by timeout is detected by a tournament manager module, which finishes the game normally, and the bot is informed of its defeat. When an entry crashes, on the other hand, the game ends abnormally and the entry is not informed of its defeat.

The 2016 version of MegaBot did not have a mechanism to account for crashes, so that the value in its payoff matrix is not properly updated when a match terminates abnormally. Thus, the portfolio member that caused the crash remain likely to be selected in a future match. The 2017 version received a mechanism to account for crashes via a counter, initialized with zero. When a match starts, we increment the crash counter of the selected algorithm against the current opponent. If the match ends

[1]`https://github.com/andertavares/MegaBot`

normally, the cleanup function is called, the counter is reset to zero and the selected algorithm's score is updated normally. In the next match, if the crash counter is not zero, we know that a crash happened in the previous match, so we assign a loss to the crashing algorithm and update its score prior to the selection process.

Appendix C

Other games mentioned in this dissertation

Throughout the text, we mention several computer games from genres other than real-time strategy games, which are our testbeds. We centralize their description in this chapter. Each section has a brief description of the game, alongside a figure and a link to a gameplay video.

C.1 Breakout

In Breakout (Figure C.1), the player controls a horizontal sliding paddle to bounce a ball. The upper part of the screen has bricks that disappear and give points to the player when hit by the ball. Difficulty increases when the paddle size decreases and the ball speed increases upon specific events.

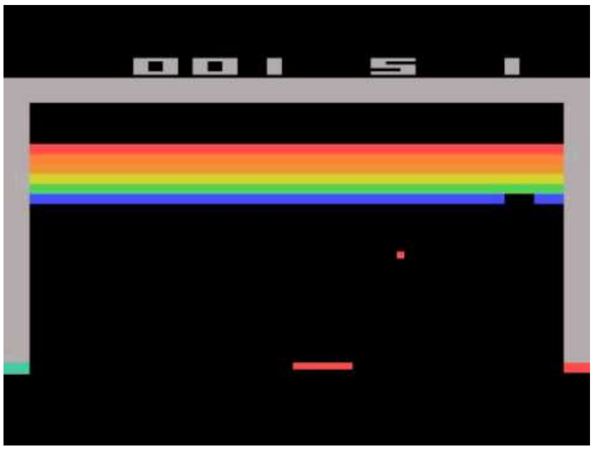


Figure C.1: Breakout screenshot.

Breakout is mainly a game of reflexes: to succeed, the player must observe the ball trajectory to intercept it using the paddle. Moreover, it is a game that frequently rewards the player, as the bricks are hit. The game was first featured in arcades, then ported to several platforms, including the Atari 2600. Example video: `https://youtu.be/Up-a5x3coC0`.

C.2 Frostbite

Frostbite is an Atari 2600 platform game in which the player controls an Eskimo that jumps between four lines of moving ice blocks. The Eskimo must deviate from enemies and collect items to score. By jumping back and forth, an igloo grows at the top of the screen, and the player can enter it to complete a level.

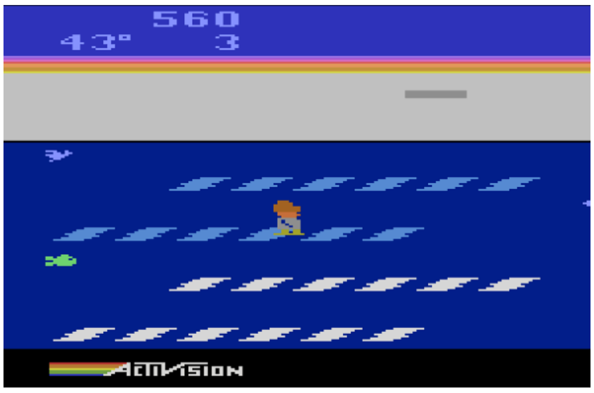


Figure C.2: Frostbite screenshot.

There is a countdown, based on the Eskimo body temperature, to build the igloo and pass the level. The Eskimo dies from frostbite if the time runs out. Frostbite is a game of reflexes with elements of long-term planning: the player must quickly deviate from enemies and correctly jump to the next ice block, but the ultimate goal is to build the igloo. The game constantly rewards the player after each successful jump to a different row of ice blocks. Example video: `https://youtu.be/iajqWQjchjY`.

C.3 Montezuma's Revenge

Montezuma's Revenge (Figure C.3) is an Atari 2600 game, where the player controls a character named Panama Joe to find the treasure of emperor Montezuma, hidden deep in his fortress. The player must explore several rooms in the emperor's fortress, climbing ropes and ladders, leaping gaps, avoiding enemies and collecting items.

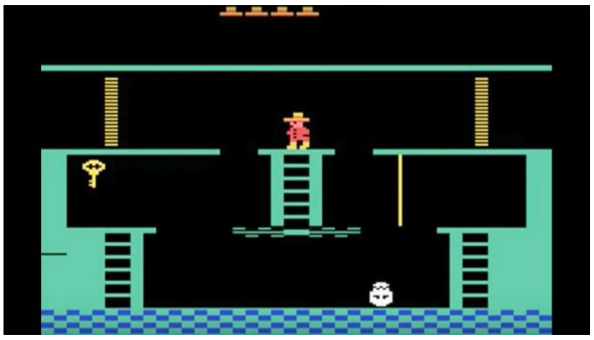


Figure C.3: Montezuma's Revenge screenshot

Montezuma's Revenge requires long-term planning: some items, such as keys, are useful far away from where they are collected. Moreover, rewards are sparse, as the player only scores by collecting items, which don't appear in every fortress room. Example video: `https://youtu.be/Klxxg9JM5tY`.

C.4 Prismata

Prismata (Fig. C.4) is a hybrid strategy game, mixing elements from real-time strategy, card games and tabletop strategy games. Players start with the same units, randomly chosen from a list of about 100 units. At each turn, players may purchase units, and use them to attack, block, produce resources or cast their special abilities. The goal of the game is to eliminate all opponents' units.

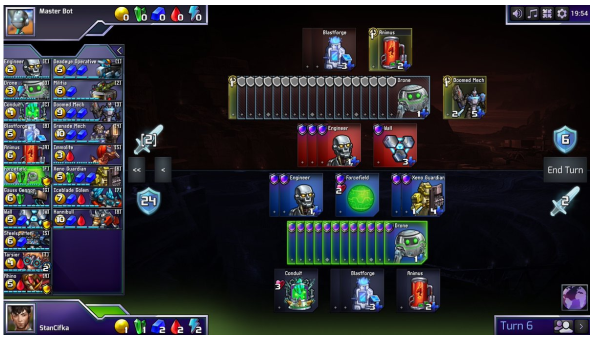


Figure C.4: Prismata screenshot.

Example video: `https://youtu.be/aq5_JwDsCgg`.

C.5 Doom

Doom (Figure C.5) is a 1993 first-person shooter game, considered a reference in the genre. The player controls a marine through three episodes of eight mandatory levels, plus a ninth optional hidden one. The player must survive by shooting every enemy in sight, in a 3D environment.

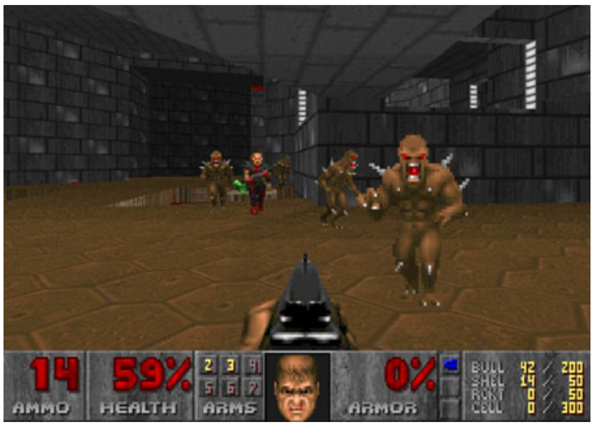


Figure C.5: Doom screenshot.

Computationally, Doom's campaign is a difficult game for its large state space (a 3D world) and the need of long-term planning: although surviving is accomplished by killing enemies in sight, the player must traverse long distances to pass a level, activating door-opening switches and look for armor, weapons, ammunition, and health kits to increase its chances of surviving. Deathmatches reduce the long-term planning requirements as the only goal is to eliminate the other players. Example video: `https://youtu.be/8mEP4cflrd4`.

C.6 Dota 2

Dota 2 (Figure C.6) is a multiplayer online battle arena (MOBA) game, a genre also referred to as Action Real-Time Strategy (ARTS), as a derivative of real-time strategy games. In MOBAs, each player controls a hero. Two teams of heroes battle to destroy each other's main structures, named "Ancients" in Dota 2. Before the game, players select their heroes from a pool of over a hundred characters. Their characteristics vary greatly, ranging from physically-strong warriors to physically-weak supporting spellcasters. Dota 2 has a single map, split into three lanes, each with three towers for each team. Towers must be destroyed in sequence for a team to advance towards enemy territory.

AI-controlled minion creatures, named creeps, spawn regularly from each base. Players gain gold and experience by killing enemy creeps. Experience levels up the hero, enhancing its abilities. Gold can be used to buy items that recover the hero's attributes and/or provide special abilities.

Figure C.6: Dota 2 screenshot.

Computationally, MOBA games such as Dota 2 are complex: the state space is enormous (every combination of unit attributes, positions and items forms a different state) and each hero has various possible actions to perform at every moment. Actions involve movements, attacks, use of items or spells, some of which have target parameters. In real-time strategy games, a player controls many units with many possible actions, such that the combinatorial complexity of actions is superior to MOBAs, where the player controls a single hero. On the other hand, MOBAs have the additional challenge of teamwork: as each player controls a single hero, he/she must coordinate with its teammates to pursue a good team behavior. Example video: `https://youtu.be/vA7RwB_G1Mk`.