Monte-Carlo Tree Search
for Artificial General Intelligence in Games

Monte-Carlo Tree Search
for Artificial General Intelligence in Games

Dissertation

to obtain the degree of Doctor at the Maastricht University,
on the authority of the Rector Magnificus Prof. dr. Rianne M. Letschert
in accordance with the decision of the Board of Deans,
to be defended in public on
Wednesday, November 13, 2019, at 16:00 hours

by

Chiara Federica Sironi

Supervisor:
Prof. dr. ir. R.L.M. Peeters
Co-supervisor:
Dr. M.H.M. Winands

Assessment Committee:
Prof. dr. G. Weiss (*chair*)
Prof. dr. Y. Björnsson (Reykjavik University)
Prof. dr. M. Gyssens (Hasselt University, transnational University Limburg)
Dr. M. Preuss (Leiden University)
Prof. dr. ir. J.C. Scholtes

This research has been funded by the Netherlands Organisation for Scientific Research (NWO) in the framework of the project GoGeneral, grant number 612.001.121.

Dissertation Series No. 2019-32
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN 978-94-6380-553-7

Printed by ProefschriftMaken, de Bilt
Cover design by ProefschriftMaken, de Bilt
Design of cover picture by Annalisa Sironi, Seregno, Italy.

©2019 C.F. Sironi, Maastricht, The Netherlands.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronically, mechanically, photocopying, recording or otherwise, without prior permission of the author.

Preface

After receiving my Master's degree in Computer Science two things were on my mind. First, I wanted to find something fun to do in the large amount of free time that suddenly became available after graduation. Second, I wanted to find a Ph.D. position in order to continue my studies. Finding something fun to do was easy. I came across an on-line course promising that by the end of the last lecture I would have been able to create a computer program that plays any abstract game you can think of. Excited by the challenge, I enrolled in this course on a topic I had never heard of before, called *general game playing*. Finding a Ph.D. position was not as quick. It took a few months of searching before I was accepted as a Ph.D. candidate at the Department of Data Science and Knowledge Engineering (DKE) of Maastricht University. Surprisingly enough, the main topic of my Ph.D. turned out to be exactly the same one of the on-line course. However, I soon realized that it is enough to follow a few weeks of on-line lectures to learn how to create a general game-playing program, but it takes more than four years of effort to make this program only slightly more intelligent. This thesis is the result of the research I performed during my Ph.D. and is also the result of the effort of all the people that supported me in the process, and whom I would like to thank.

First of all, I wish to thank my daily supervisor, Mark Winands, for his invaluable help during these years and for teaching me much about research. I am grateful for all the time and effort he spent in guiding me during my Ph.D. and his immense knowledge about search in games has been of great help in the realization of this thesis. I also wish to thank Ralf Peeters for being my promotor and giving me useful feedback.

I would also like to thank all the people with whom I collaborated during the past years. I enjoyed visiting the Game AI research group at QMUL, UK and I am especially grateful to Jialin Liu, Diego Pérez-Liébana, Raluca Gaina, Ivan Bravi and Simon Lucas for their valuable insights on the GVG-AI framework and on Evolutionary Algorithms. I also enjoyed working with Cezary Siwek and Jakub Kowalski, from the University of Wrocław, Poland. I wish to thank them for the interest they showed in my work and for giving me the opportunity to collaborate with them. Their insights on FPGAs were a valuable addition to this thesis. Furthermore, I wish to thank Tom Vodopivec, from the University of Ljubljana for all the interesting talks we had in Maastricht and for sharing his work and giving me advice. I am also grateful to Stephan Schiffel, from Reykjavik University for his advice on GDL and PropNets and for making the time I spent visiting the Cadia lab enjoyable.

I also wish to thank all my colleagues for making DKE a pleasant work environment. My gratitude also goes to the IT-support team, especially to Peter Geurtz for always taking care of the servers and making sure they are up-to-date and ready to run experiments.

Of course, I would like to thank all the people that shared the office with me during these years and made it a fun place to work at. My very first office mates, Chun, Hendrik and Libo. My senior Ph.D. office mates, Firat, Nasser and Shuang, I learned a lot from them. Arjun and Yiyong deserve my thanks for keeping me company in the office and for all the fun lunch breaks we had, together with Seethu and Amir as well. My thanks also go to my current office mates and colleagues at Tapijn, Cameron, Dennis, Eric, Lianne, Matthew, Tahmina and Walter. I enjoy the interesting discussions on games that we have in the office, the occasional after-work drinks and spending time playing with their cool Ludii game system. Furthermore, I wish to thank all other Ph.D. fellows and other colleagues that shared with me the good (and sometimes hard) times of work life at DKE: Bijan, Christos, Dario, Esam, Jordi, Katharina, Kirill, Li, Lucas, Maria, Meysam, Monica, Tom, Wei and Zhenglong.

My thanks also go to my friend Olivia for her moral support and for always knowing the best places where to hang out in Maastricht. My friends Alessandro, Daniela, Federica, Isabella and Sonia deserve my thanks as well. They proved that distance does not matter when there is real friendship. I am also grateful to Els and Harrie, for all the help they gave me when I first moved to the Netherlands and for always being there for me. Special thanks go to my boyfriend Joost, for always being by my side, cheering me on and keeping me motivated. He deserves praise also for showing an infinite amount of patience every time I was stressed for an approaching deadline. Last but not least, my gratitude goes to all my family, to my sister Annalisa and especially to my parents Luisa and Federico. They have been supportive throughout all my life, always encouraging each of my decisions. Grazie per tutto quello che avete fatto per me.

Chiara Sironi, 2019

Acknowledgements

The research has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems. This research has been funded by the Netherlands Organisation for Scientific Research (NWO) in the framework of the project GoGeneral, grant number 612.001.121.

Table of Contents

Preface v

Table of Contents vii

List of Figures xi

List of Tables xii

List of Algorithms xv

1 Introduction 1

1.1 Games . . . 2
1.2 Games and Artificial Intelligence . . . 5
1.3 Games and Artificial General Intelligence . . . 7
1.4 Search Techniques . . . 10
1.5 Problem Statement and Research Questions . . . 12
1.6 Thesis Overview . . . 14

2 Search Techniques 17

2.1 Tree Search in Games . . . 17
2.2 Monte-Carlo Methods . . . 20
2.2.1 Flat Monte-Carlo Search . . . 20
2.2.2 Multi-Armed Bandit Algorithms . . . 22
2.3 Monte-Carlo Tree Search . . . 23
2.3.1 "Open-Loop" MCTS for Non-Deterministic Games . . . 25
2.4 The UCT Selection Strategy . . . 30
2.4.1 Simultaneous Move UCT . . . 31
2.5 MCTS Enhancements . . . 36
2.5.1 Selection Strategy Enhancements . . . 36
2.5.2 Play-out Strategy Enhancements . . . 38
2.5.3 Transposition Tables . . . 41
2.5.4 Tree Reuse . . . 47
2.6 Discussion . . . 49

3 Test Environments 51
3.1 Stanford General Game Playing . . . 51
3.1.1 Game Description Language . . . 52
3.1.2 Game Management . . . 57
3.1.3 Competition . . . 58
3.1.4 GGP Base Agent . . . 59
3.2 General Video Game AI . . . 61
3.2.1 Video Game Description Language . . . 61
3.2.2 Game Management . . . 64
3.2.3 Single-Player Planning Competition . . . 65
3.2.4 GVG-AI Agents . . . 66
3.3 Discussion . . . 68

4 Optimizing Propositional Networks 71
4.1 Background . . . 72
4.1.1 Propositional Networks . . . 72
4.2 PropNet Implementation . . . 75
4.2.1 Initialization . . . 75
4.2.2 Optimizations . . . 76
4.2.3 PropNet-based Reasoner . . . 85
4.3 Experiments . . . 86
4.3.1 Setup . . . 86
4.3.2 Comparison of Single Optimizations . . . 88
4.3.3 Comparison of Combined Optimizations . . . 90
4.3.4 Comparison of PropNet Reasoner and Prover . . . 92
4.3.5 Game-Playing Performance . . . 94
4.4 Encoding PropNets on Field Programmable Gate Arrays . . . 95
4.4.1 FPGA-PropNet Reasoner Implementation . . . 95
4.4.2 FPGA-PropNet Reasoner Performance . . . 97
4.5 Chapter Conclusions and Future Research . . . 100

5 Rapid Action Value Estimation Variants 103
5.1 All-Moves-As-First . . . 104
5.2 RAVE Variants . . . 105
5.2.1 Rapid Action Value Estimation . . . 105
5.2.2 Generalized Rapid Action Value Estimation . . . 106
5.2.3 History Rapid Action Value Estimation . . . 108
5.3 Experiments . . . 108
5.3.1 Games . . . 109
5.3.2 Setup . . . 109
5.3.3 Tuning the Equivalence Parameter K . . . 111
5.3.4 Matching RAVE Variants against UCT . . . 111
5.3.5 Matching RAVE Variants against UCT with MAST . . . 115
5.3.6 Matching RAVE Variants against Each Other . . . 117
5.3.7 Memory Usage . . . 119
5.4 Chapter Conclusions and Future Research . . . 120

6 On-line Search-Control Parameter Tuning for MCTS 123
6.1 Related Work 125
6.2 Design of Self-Adaptive MCTS 126
6.2.1 Integration of Parameter Tuning with MCTS 126
6.2.2 Formulation of the Parameter Tuning Problem 127
6.3 Allocation Strategies 128
6.3.1 Discrete Allocation Strategies 129
6.3.2 Continuous Allocation Strategy 142
6.4 Empirical Evaluation 144
6.4.1 Setup 144
6.4.2 On-line Parameter Tuning for the SP Agent 150
6.4.3 On-line Tuning for the AP Agent 150
6.4.4 On-line Parameter Tuning Validation 157
6.4.5 Parameters Inter-dependency 157
6.4.6 Tuning Six Parameters with Different Time Constraints . . . 159
6.4.7 Best On-line Tuning Agent vs CadiaPlayer 161
6.4.8 On-line Parameter Tuning in Real-time Settings 162
6.4.9 Discussion 166
6.5 Chapter Conclusions and Future Research 167

7 Comparing Randomization Strategies for Search-Control Parameters in MCTS 171
7.1 Related Work 172
7.2 Search-Control Parameter Randomization 173
7.3 Empirical Evaluation 175
7.3.1 Setup 175
7.3.2 Comparison of Parameter Randomization Strategies 178
7.3.3 Randomization per Simulation vs Fixed Parameters 180
7.3.4 Randomization per Simulation vs Parameter Tuning 185
7.3.5 Comparison of Default Parameter Values, Parameter Randomization and On-line Parameter Tuning 186
7.3.6 Search Tree Analysis 191
7.3.7 Parameter Randomization in Real-time Settings 197
7.4 Chapter Conclusions and Future Research 201

8 Conclusions and Future Research 203
8.1 Answers to the Research Questions 203
8.1.1 Speeding Up Game Rule Interpretation 203
8.1.2 Local and Global Information in the Selection Strategy . . . 204
8.1.3 On-line Search-Control Parameter Tuning 205
8.1.4 Search-Control Parameter Randomization 206
8.2 Answer to the Problem Statement 207
8.3 Recommendations for Future Research 208
8.3.1 Specific Recommendations 208
8.3.2 General Recommendations 210

References **213**

Appendices

A Example of Game Descriptions **235**
A.1 GDL Description for Tic Tac Toe 235
A.2 VGDL Game Description for Zelda 238

B Game Rules **241**
B.1 Games for the Stanford GGP Project 241
B.2 Games for the GVG-AI Project 245

C Supplementary Results for Chapter 4 **249**

D Supplementary Results for Chapter 5 **255**

E Supplementary Results for Chapter 6 **257**

F Supplementary Results for Chapter 7 **265**

Index **267**

Valorization **269**

Summary **273**

Curriculum Vitae **279**

SIKS Dissertation Series **281**

List of Figures

2.1 Representation of a game tree. . . . 19
2.2 Outline of Monte-Carlo Tree Search. . . . 23
2.3 Representation of a game tree with chance nodes. . . . 27
2.4 "Open-loop" representation of a game tree for a non-deterministic game. 28
2.5 Representation of a game tree for a simultaneous move game with two players. . . . 31
2.6 Sequential action selection in a turn of a two-player simultaneous move game. . . . 33
2.7 Computation of cumulative statistics for Player 1 in a state of a simultaneous move game with two players. . . . 34
2.8 Statistics memorized by each player in a state, when DUCT is applied on a simultaneous move game with two players. . . . 35
2.9 Extraction and update of N-Grams after an MCTS simulation. . . . 40
2.10 Use of N-Grams to compute the NST value of an action. . . . 41
2.11 Application of UCT on a search tree built without using a transposition table. . . . 44
2.12 Application of UCT1, UCT2, UCT3 and UCT4 on a search graph built using a transposition table. . . . 45
2.13 Tree reuse in MCTS. . . . 47
2.14 Example of graph reuse in MCTS with transposition tables. . . . 48

3.1 Representation of a state of Tic Tac Toe with GDL. . . . 52
3.2 Representation of a level of Zelda with VGDL. . . . 64

4.1 GDL game description for a simple game. . . . 73
4.2 Grounded GDL game description for a simple game. . . . 73
4.3 PropNet structure example. . . . 74
4.4 Examples of changes to the PropNet structure after applying Opt0. . 79
4.5 Changes to the PropNet structure after applying Opt1. . . . 80
4.6 Examples of changes to the PropNet structure after applying Opt2. . 82
4.7 Changes to the PropNet structure when applying Opt3. . . . 84
4.8 Tradeoff between MCTS play-out speed and distinct Monte-Carlo evaluations. . . . 97

5.1 UCT vs AMAF statistics update. . . . 105
5.2 Use of AMAF statistic for the GRAVE strategy. . . . 107

6.1 Interleaving on-line tuning with MCTS. . . . 126
6.2 MAB and HE representation of the combinatorial action-space of the parameter tuning problem. . . . 130
6.3 Overview of the *exploration* and *exploitation* phases of NMC. . . . 133
6.4 Overview of the *generation* and *evaluation* phases of LSI. . . . 134
6.5 Overview of the execution of EA. . . . 140
6.6 Overview of the execution of NTBEA. . . . 141

6.7 Win percentage of AP_{NMC}, AP_{LSI}, AP_{EA} and AP_{NTBEA} tuning six parameters with different time constraints. 160

7.1 Randomization strategies. 174
7.2 Win percentage of the agents with the parameter randomization strategies against the agent with fixed default parameter values. 178
7.3 Maximum MCTS tree depth over game turns of AP, $AP_{SIM\text{-}RND}$ and AP_{NTBEA}. 191
7.4 Average effective branching factor over game turns of AP, $AP_{SIM\text{-}RND}$ and AP_{NTBEA}. 192
7.5 Trees built for Breakthrough by the AP, $AP_{SIM\text{-}RND}$ and AP_{NTBEA} agents, respectively. 193
7.6 Trees built for Knightthrough by the AP, $AP_{SIM\text{-}RND}$ and AP_{NTBEA} agents, respectively. 194
7.7 Trees built for Quad by the AP, $AP_{SIM\text{-}RND}$ and AP_{NTBEA} agents, respectively. 195
7.8 Trees built for Connect Four by the AP, $AP_{SIM\text{-}RND}$ and AP_{NTBEA} agents, respectively. 196

C.1 Speed of the Prover without and with cache over different game turns for the games of Amazons, Battle, Breakthrough, and Chinese Checkers with 1, 2, 3 and 4 players. 250
C.2 Speed of the Prover without and with cache over different game turns for the games of Chinese Checkers with 6 players, Connect Four, Othello, Pentago, Skirmish, and Tic Tac Toe. 251
C.3 Speed of the optimized PropNet without and with cache over different game turns for the games of Amazons, Battle, Breakthrough, and Chinese Checkers with 1, 2, 3 and 4 players. 252
C.4 Speed of the optimized PropNet without and with cache over different game turns for the games of Chinese Checkers with 6 players, Connect Four, Othello, Pentago, Skirmish, and Tic Tac Toe. 253

List of Tables

4.1 Comparison of single optimizations. 89
4.2 Comparison of combined optimizations. 91
4.3 Comparison of the PropNet reasoner with the Prover and effect of the cache. 93
4.4 Win percentage of the PropNet agent against the Prover agent. . . . 94
4.5 Comparison of the FPGA-PropNet with the software PropNet and the Prover, based on running random simulations from the initial game state. 98

4.6 Initialization time and memory usage of the FPGA-PropNet and the software PropNet. . . . 99

5.1 Win percentage of P_{RAVE}, P_{GRAVE} and P_{HRAVE} against P_{UCT} for different values of K. . . . 112
5.2 Win percentage of P_{RAVE}, P_{GRAVE} and P_{HRAVE} with best K against P_{UCT} with 1s play-clock and start-clock. . . . 113
5.3 Simulations per second of P_{UCT}, P_{RAVE}, P_{GRAVE} and P_{HRAVE}. . . . 113
5.4 Win percentage of P_{RAVE}, P_{GRAVE} and P_{HRAVE} with $K = 250$ against P_{UCT} with 10s play-clock and start-clock. . . . 114
5.5 Win percentage of $P_{\text{RAVE-MAST}}$, $P_{\text{GRAVE-MAST}}$ and $P_{\text{HRAVE-MAST}}$ with best K against $P_{\text{UCT-MAST}}$. . . . 115
5.6 Win percentage of $P_{\text{RAVE-MAST}}$ against P_{RAVE}, $P_{\text{GRAVE-MAST}}$ against P_{GRAVE} and $P_{\text{HRAVE-MAST}}$ against P_{HRAVE}. . . . 116
5.7 Win percentage of all possible combinations of agents with and without MAST. . . . 118
5.8 Average number of move statistics per node of P_{RAVE} and P_{GRAVE}. 119

6.1 Default values, discrete and continuous domains of the parameters considered in the experiments on the Stanford GGP project. . . . 146
6.2 Default values, discrete domains and sub-optimal values of the parameters considered in the experiments on the GVG-AI project. . . . 149
6.3 Win percentage of the on-line tuned SP agent that tunes two parameters with different allocation strategies against the SP agent with default parameter values. . . . 151
6.4 Win percentage of the on-line tuned AP agent with different allocation strategies that tune two parameters against the AP agent with default parameter values. . . . 152
6.5 Win percentage of the on-line tuned AP agent with different allocation strategies that tune four parameters against the AP agent with default parameter values. . . . 153
6.6 Win percentage of the on-line tuned AP agent with different allocation strategies that tune six parameters against the AP agent with default parameter values. . . . 154
6.7 Variation (%) of visited nodes per second of the on-line tuned AP agent that tunes four parameters with respect to the off-line tuned AP agent. . . . 156
6.8 Variation (%) of MCTS iterations per second of the on-line tuned AP agent that tunes four parameters with respect to the off-line tuned AP agent. . . . 158
6.9 Win percentage of SP_{NTBEA} and AP_{NTBEA} against agents that randomize parameter values before each game run. . . . 159
6.10 Win percentage of the on-line tuning AP_{LOCAL} agent that does not consider interdependency among parameters, and of the on-line tuning AP agents that achieve the highest win percentage against off-line tuned AP. . . . 160

6.11 Win percentage of AP and AP_{NTBEA} (1s start- and play-clock) against CadiaPlayer (10s start- and play-clock). 161
6.12 Win percentage of MaastCTS2 with fixed parameter values (MP), with sub-optimal fixed parameter values ($MP_{SUB\text{-}OPT}$), tuned on-line with the MAB strategy (MP_{MAB}) and tuned on-line with the NTBEA strategy (MP_{NTBEA}), with game tick set to 40ms. 163
6.13 Variation (%) of MCTS iterations per tick (40ms) of MaastCTS2 tuned on-line with the MAB strategy (MP_{MAB}) and tuned on-line with the NTBEA strategy (MP_{NTBEA}) with respect to MaastCTS2 with fixed parameter values (MP). 164
6.14 Win percentage of MaastCTS2 with fixed parameter values (MP), with sub-optimal parameter values ($MP_{SUB\text{-}OPT}$), tuned on-line with the MAB strategy (MP_{MAB}) and tuned on-line with the NTBEA strategy (MP_{NTBEA}), with game tick set to 100ms. 165

7.1 Default values, discrete domains and sub-optimal values of the parameters considered in the experiments on the Stanford GGP project. 176
7.2 Default values, discrete domains and sub-optimal values of the parameters considered in the experiments on the GVG-AI project. . . . 177
7.3 Comparison of parameter randomization strategies against each other for different numbers of randomized parameters. 179
7.4 Comparing all feasible values of C with value randomization of C per simulation. 181
7.5 Comparing all feasible values of ϵ_{MAST} with value randomization of ϵ_{MAST} per simulation. 182
7.6 Comparing all feasible values of K with value randomization of K per simulation. 183
7.7 Parameter randomization against parameter tuning. 185
7.8 Win percentage of $AP_{SIM\text{-}RND}$ and AP_{NTBEA} against the agent with default parameter values, AP. 187
7.9 Win percentage of AP, $AP_{SIM\text{-}RND}$ and AP_{NTBEA} against the agent with sub-optimal fixed parameter values. 188
7.10 Win percentage of AP, $AP_{SIM\text{-}RND}$ and AP_{NTBEA} (1s start- and play-clock) against CadiaPlayer (10s start- and play-clock). 189
7.11 Win percentage of MaastCTS2 with fixed parameter values (MP), with sub-optimal parameter values ($MP_{SUB\text{-}OPT}$), with randomization per simulation ($MP_{SIM\text{-}RND}$) and tuned on-line with the NTBEA strategy (MP_{NTBEA}), with game tick set to 40ms. 198
7.12 Win percentage of MaastCTS2 with fixed parameter values (MP), with sub-optimal parameter values ($MP_{SUB\text{-}OPT}$), with randomization per simulation ($MP_{SIM\text{-}RND}$) and tuned on-line with the NTBEA strategy (MP_{NTBEA}), with game tick set to 100ms. 199

B.1 Characteristics of the games of the Stanford GGP project used for the experiments. 242

B.2 Characteristics of the games of the GVG-AI project used for the experiments. . . . 246

D.1 Win percentage of $\mathrm{P}_{\text{UCT-MAST}}$ against P_{UCT} with 1s play-clock and start-clock. . . . 255

E.1 Win percentage of the AP agent that tunes four parameters on-line with the MAB allocation strategy with or without using a batch of simulations to evaluate each parameter combination. . . . 260
E.2 Win percentage of the AP agent that tunes four parameters on-line with the MAB allocation strategy with or without using a batch of simulations to evaluate each parameter combination. . . . 260
E.3 Win percentage of the AP agent that tunes four parameters on-line with the NMC allocation strategy for different combinations of values for C_g and C_l. . . . 261
E.4 Win percentage of the AP agent that tunes four parameters on-line with the LSI allocation strategy, for different values of the factor κ. . . 262
E.5 Win percentage of the AP agent that tunes four parameters on-line with the EA allocation strategy, for different values of the elite size μ. 262
E.6 Win percentage of the AP agent that tunes four parameters on-line with the NTBEA allocation strategy, for different number of generated neighbors X. . . . 263
E.7 Win percentage of the AP agent that tunes four parameters on-line with the NTBEA allocation strategy, using different lengths for the n-tuples. . . . 263

F.1 Win percentage of the AP agent that randomizes two, four and six parameters with different randomization strategies, against the AP agent with fixed default parameter values. . . . 266

List of Algorithms

1 Pseudocode for Monte-Carlo Search. . . . 21
2 Pseudocode for Monte-Carlo Tree Search. . . . 26
3 Pseudocode for “open-loop” Monte-Carlo Tree Search. . . . 29
4 Remove constant-value components. . . . 76
5 Remove true components. . . . 77
6 Remove anonymous propositions. . . . 78
7 Detect and remove constant-value components. . . . 81
8 Remove components with no outputs. . . . 84
9 Pseudocode for the MAB allocation strategy. . . . 129
10 Pseudocode for the HE allocation strategy. . . . 131
11 Pseudocode for the NMC allocation strategy. . . . 132
12 Pseudocode for the LSI allocation strategy. . . . 135
13 Pseudocode for the EA allocation strategy. . . . 138
14 Pseudocode for $LModel$. . . . 139

15 Pseudocode for the NTBEA allocation strategy. 140
16 Pseudocode for the CMA-ES allocation strategy. 143

Chapter 1

Introduction

This thesis investigates how search can be utilized to support *Artificial General Intelligence* (AGI) in games. The aim of AGI is creating Artificial Intelligence (AI) that can perform multiple different tasks in multiple different environments, can autonomously manage itself and possesses the ability to adapt to perform any new task that it might have never performed before (Goertzel and Pennachin, 2007). Similarly, the goal of the research area of general game playing (GGP)[1] is to create programs that are able to play many (video) games with different properties without requiring any human intervention. This means that programs cannot rely on game-specific and prior knowledge and have to automatically adapt to each new game.

Among the main human competences that AGI should aim to reproduce, Goertzel (2014) identifies planning. Also Yannakakis and Togelius (2015) mention search and planning as a relevant subfield of AGI research in games. Given that games can model a wide variety of computationally hard problems (Schaul, Togelius, and Schmidhuber, 2011; Yannakakis and Togelius, 2018), they can be considered a reasonable subset of all the planning tasks that we would like an AGI to be able to perform. Therefore, GGP seems particularly suitable as a test domain to improve search and planning in the direction of achieving AGI.

A search technique that can be directly applied to GGP is *Monte-Carlo Tree Search* (MCTS), because in its basic form it does not require any domain-specific knowledge (Browne *et al.*, 2012). MCTS is a simulation-based search technique for making decisions in a given (game) domain. It incrementally builds a tree representation of the search space of the problem, focusing on promising actions. To estimate the value of the actions it uses Monte-Carlo simulations. MCTS has already been successful in GGP both for abstract games (e.g. board games) (Finnsson and Björnsson, 2010) and video games (Perez-Liebana *et al.*, 2016). However, for many games it is still far from performing at the same level of game-specific programs or human experts. Therefore, this thesis investigates new techniques to improve MCTS in order to advance AGI further.

[1]In this thesis the term *general game playing* and the acronym GGP are used to indicate the broad field that encompasses general game playing for any type of game, such as board games, video games, etc.

This chapter is structured as follows. First, Section 1.1 gives an introduction to games and describes some properties that are useful to classify those used in this thesis. Section 1.2 discusses the role of games in AI research, while Section 1.3 extends the discussion to the importance of games for research on AGI. Next, Section 1.4 summarizes the search methods that are most relevant for this work. The problem statement and the four research questions for this thesis are formulated in Section 1.5. Finally, Section 1.6 gives an overview of the thesis.

1.1 Games

Since ancient times, *games* have been part of human culture (Murray, 1952). The oldest game we have evidence of, Senet, is an Egyptian game dating back approximately to 3 500 B.C. that has been played for more than 3 000 years (Piccione, 1980). The exact rules of this game are unknown, but the main goal of the two players is to get their own pawns across a board divided into 3 rows of 10 cells each. Other games that originated a long time ago are Backgammon, Go and Chess (Murray, 1952; Parlett, 2018). Early versions of Backgammon come from Persia and are about 5 000 years old, the origins of Go can be found in China, probably about 3 000–4 000 years ago, and it seems that precursors of Chess originated in India before the 6th century. These games are still played nowadays, together with thousands of other games, and new games are constantly being developed. The popularity of games throughout history and in current days is probably because they couple fun and entertainment together with intellectual activity and competition. Moreover, by playing games people can train their skills and compare them with others.

In general, a *game* is defined by a set of *rules* and involves one or more *players* that want to reach a *goal*. To achieve their goal they try to modify the *state* of the game by performing *actions* that are admissible according to the rules. How the state changes when actions are performed and what conditions determine the end of the game is also specified by the rules. There are different types of games, which can be classified according to multiple properties. Below is a list of properties that are relevant for the games considered in this thesis. It is also important to remind that these properties might influence the performance of the AI techniques designed to play games.

Equipment. Games can differ in the equipment they require to be played. Games that require a board are referred to as *board games*, of which Chess is a famous example. Games like Bridge and Poker, which are played with cards, are referred to as *card games*. An example of *tile-based game* is Dominoes, which requires rectangular tiles marked with numbers. There are also games, like Rock-Paper-Scissors, that do not require any equipment at all, and games, like Monopoly and Risk, that require multiple types of equipment (e.g. a board, dice, various types of tokens and cards with different purposes). So-called *Eurogames* are a particular type of board games that also use different types of equipment, like cards and other pieces. An example is the game Catan. Games with no or little equipment, like Chess and Poker, are sometimes referred to as *abstract games*. This thesis uses this particular meaning of the term, although

there seems to be no global consensus on a single definition of abstract games. Finally, a category that is new with respect to the ones just mentioned is *video games*, sometimes also called *digital games*, which require a digital device to be played on, like a computer or a console.

Number of players. Games can involve different numbers of players. *One-player* games require only one player and can be seen as *optimization problems* or *puzzles.* In such games the player aims at the best possible outcome without having to take into consideration any other player. Some examples are Solitaire and Sudoku. When talking about video games, the term *single-player* games is more commonly used. A game is called *single-player* when there is only one playable character, though there could also be a number of Non-Player Characters (NPCs) that behave according to a fixed strategy and can influence the game outcome. This means that the player has to consider them when making decisions. Examples of classic video games with a single-player mode are Pac-Man and Super Mario Bros. Finally, we can distinguish between *two-player* games and games that require more than two players, the *multi-player* games. Both in two- and multi-player games the players have to take into account each other's actions. The two categories are usually identified separately because multi-player games add an extra level of difficulty with respect to two-player games. For multi-player games it is usually harder to find an equilibrium and the possibility of coalition formation among the players adds an extra challenge. Checkers, Chess and Go are well-known examples of two-player games, while Chinese Checkers, Monopoly and Risk are examples of multi-player games. This thesis focuses on two- and multi-player abstract games, and on single-player video games.

Game flow. The game flow can be discrete or continuous. In a game with a discrete flow, also called *turn-based* game, time is discretized into turns and each player has to wait for her[2] turn to play an action. Turn-based games can be further distinguished into *sequential move* games (or *turn-taking* games), where players move one per turn, and *simultaneous move* games, where more than one player can move in the same turn. Chess and Go are examples of sequential move games, while Rock-Paper-Scissors and Diplomacy are examples of simultaneous move games. When players can perform an action at any point in time and the game flow is continuous (despite some actual fine-grained discretization necessary to implement the game on a machine) we talk about *real-time* games. Most video games, like the ones in this thesis, are real-time. All board games used in this thesis are turn-based, with either sequential or simultaneous moves.

Observability. There are games where at all times all the players have all the information that defines the current game state. These games are called *perfect-information* games. Some examples are Chess, Checkers, and in general most of the classic board games, where the players can see the complete board and the placement of the pieces on it. If at any time during the game any of the

[2]Where gendered forms cannot elegantly be avoided, the generic feminine is used throughout this thesis in order to avoid cumbersome forms such as "he or she", "(s)he", or "her/his".

players is missing some information on the current game state and only has access to a partial observation, then the game is said to have *imperfect information*. Most of the card games, like Poker and Bridge are an example of this category, because a player does not know which cards the other players have. It is important to make a distinction between imperfect-information and *incomplete-information* games (Harsanyi, 1967). In games with incomplete information the players do not have all the information about the structure of the game. For example, they might not know their payoffs or available actions, or the rules of the game. An example of game with incomplete information is World of Warcraft, a massively multi-player on-line role-playing game where players do not have complete knowledge about the environment nor about other players and their goals. By contrast, games where each player knows everything about the structure of the game are said to have *complete information*. Examples of this category are Chess and Poker. All the games in this thesis have perfect information, although simultaneous move games can be considered as having imperfect information because the player does not know which actions the opponents will select in each turn.

Determinism. Games for which the next state is solely determined by the actions of the players are called *deterministic*. In a deterministic game, given a state and an action of the players, there is only one possible next state. If the game state can be changed by any event other than the players' actions (e.g. by the roll of a die or by NPC's actions in a video game), then for each state and action of the players multiple next states can be reached. If the players know the probabilities of reaching each possible state by playing an action then the game is *stochastic*. If these probabilities are unknown to the players the game is *non-deterministic*. An example of a deterministic game is Chess. Backgammon is a stochastic game because the probabilities of each outcome of the die are known ($\frac{1}{6}$ for each face). Ms. Pac-Man (a later variant of Pac-Man) is a non-deterministic game because there is randomness in the ghosts' movement. The abstract games this thesis considers are all deterministic, while the video games can be either deterministic or not.

Payoff. A two- or multi-player game where the sum of the payoffs of the players is always zero is called a *zero-sum* game. To maintain a sum of zero, it means that whenever a player has a gain in her payoff there has to be a corresponding loss for one or more of the other players. A zero-sum game is a special case of *constant-sum* game, which is a game where the payoffs of the players always add up to a constant value. Zero-sum and constant-sum games usually foster competition among the players. We can also have *variable-sum* games, where players can aggregate gains and losses. Variable-sum games might still be competitive, but can also be cooperative if the players can increase the sum of their payoffs by cooperating. This thesis considers both constant-sum and variable-sum games.

1.2 Games and Artificial Intelligence

Over the years, research on Game AI has been applied in many categories of games, going from classic board games to card games and video games. Each category is characterized by different combinations of the properties listed in the previous section, and each of them poses different challenges to the research community. This section briefly presents the history of Game AI and discusses the major breakthroughs in the field.

Since the creation of the first computers there has been interest in classic board games as test-bed for AI techniques. The reason is that these games provide an abstract representation of real-world problems, and have the advantage of being well-formed, structured and easy to model and simulate, while still being reasonably complex to tackle. Moreover, the ability for a human to solve a game or play it well has always been coupled with intelligence and rational thought. Thus, the better a computer program can perform on a game the more it is considered to be intelligent (Shannon, 1950; Genesereth and Thielscher, 2014).

Even before AI was established as a research field in 1956, there had been work investigating whether computers could be programmed to play board games. Turing (1953) defined one of the first Chess-playing algorithms that was based on the ideas of *minimax* (Neumann and Morgenstern, 1944). At the same time, Shannon (1950) described an even more complete set of ideas on how a computer could play Chess. He proposed a representation for board positions, an evaluation function, quiescence search, and some ideas for selective game-tree search. Samuel completed in 1955 the implementation of the first Checkers program that was able to beat weak amateur players (Samuel, 1959). This program was able to improve its evaluation function by repeatedly playing against itself, and for this reason, it is regarded as one of the first game-playing programs capable of learning.

In later years also other games, like Backgammon, attracted the interest of the AI research community, because of the increased complexity given by their stochastic component. However, despite all the initial enthusiasm, up until the 1990s the majority of game-playing programs were still performing below human level. One of the first breakthroughs for research in game AI was when TD-Gammon, a Backgammon computer program implemented by Gerald Tesauro in 1992, reached a play level close to top human players (Tesauro, 1992; Tesauro, 1995). Not much time later, in 1994, the Checkers program created by Schaeffer *et al.* (1996), Chinook, became world champion. Schaeffer *et al.* (2007) continued improving their work until Checkers was finally solved by proving it is a draw and showing the strategy to achieve it. However, the most recognized milestone for game AI happened in 1997, when Deep Blue, a chess-playing program created by IBM, defeated the world Chess champion Garry Kasparov (Hsu, 2002).

After all this success, research on board games started focusing on another challenge, the game of Go. Compared to games like Chess and Checkers, Go has a much larger branching factor and search space. It is also hard to incorporate expert knowledge in a game-playing program for Go. For these reasons, it has been considered one of the most complex board games by many researchers and it has been recognized as a grand challenge for AI (Müller, 2002; Gelly *et al.*, 2012). For many years research in Go kept progressing steadily, though the level of computer programs remained

close to amateur human players. It was around 2006, when MCTS was first proposed (Kocsis and Szepesvári, 2006; Coulom, 2007a) that the level of Go computer programs started substantially increasing. In 2007 the computer program MoGo defeated a 5-dan professional human player on the reduced 9×9 Go board without any handicap[3] (Gelly and Wang, 2007; Gelly and Silver, 2008). In subsequent years, the programs MoGo (Gelly *et al.*, 2006; Gelly and Silver, 2011), Crazy Stone (Coulom, 2007b) and Zen were able to beat professional players on the standard 19×19 Go board with different handicaps. In 2016 AlphaGo (Silver *et al.*, 2016), a computer program created by Google DeepMind, was the first that managed to beat one of the best Go players, the professional 9-dan player Lee Sedol, without any handicap. In 2017 a new improved version of AlphaGo was designed, AlphaGo Zero, which could beat AlphaGo and was not using any kind of pre-coded domain knowledge (Silver *et al.*, 2017). In the same year, the same approach was shown to work also in the games of Chess and Shogi, for which the generalized player AlphaZero was able to beat the best playing programs (Silver *et al.*, 2018).

These classic board games, however, have all the same features: they have a discrete search and action space, they are played on a two-dimensional and discrete board, they are turn-based and most of them are deterministic and with perfect information. This is why the game AI community has devoted its attention also to card games. Poker, for example, is interesting for the community because of properties such as imperfect information, the presence of multiple competing players that need to be taken into account and modeled, the necessity of risk management and the presence of deception (Billings *et al.*, 2002). As proof of the recent progress of AI in poker, Bowling *et al.* (2015) report that one of the many variants of poker, heads-up limit Texas Hold'em, has been solved.

In the last twenty years, interest in video games has also been growing (Yannakakis and Togelius, 2018). First, video games are popular with the public and their industry is constantly growing, demanding increasingly smarter AI characters and more realistic and entertaining game play for the users, fostering research in this direction (Laird and VanLent, 2001). Second, they also require a diversified range of thinking skills to be played. They require the ability to orientate and move in 3D environments, knowing how to deal with the game physics. Moreover, the player should be able to respond quickly to events if the video game is in real time, taking into account all the NPCs that can change the game state independently of the player and asynchronously. To reach a strong level of play an agent needs all this range of skills, which are not trivial to implement, therefore research is still needed in this area. Finally, besides game playing, (video) games are interesting also for research on AI that can automatically generate content for them, i.e. *procedural content generation*, or adapt to the level of the player (Togelius *et al.*, 2011).

Most of the current video game-playing programs are still far from reaching top human-level performance. However, as evidence of the ongoing effort of the research community, numerous competitions were established in the past decade as benchmarks for AI techniques. They have different purposes and focus on differ-

[3]A handicap means that a certain number of stones of the color of the computer program are placed on the board before the start of the game to make up for the difference in strength with respect to the human player.

ent categories of video games. Pac-Man is probably one of the *arcade* games that attracted more attention and its variants are still actively being used as a tool for research (Rohlfshagen *et al.*, 2018). This game is challenging thanks to its real-time and stochastic components, while at the same time having relatively simple rules and a small set of available actions. Numerous competitions have been organized for the game of Pac-Man and its variants. The most recent one and still running is the *Ms. Pac-Man Versus Ghost Team Competition* (Williams, Perez-Liebana, and Lucas, 2016), based on Ms. Pac-Man and started in 2016 as an updated version of the previous *Ms. Pac-Man Versus Ghost Competition* (Rohlfshagen and Lucas, 2011). The competition has both a track to evaluate agents that control Ms. Pac-Man and a track to evaluate agents that control the NPCs in the game. It also offers an extra challenge for the AIs thanks to the introduction of partial observability, allowing each character to only have a first person limited view of the maze.

To evaluate research progress on *real-time strategy* (RTS) games many AI competitions based on the game of StarCraft have been organized, among which the *StarCraft AI Competition* is the most popular (Ontañón *et al.*, 2013). Due to real-time play, imperfect information, non-determinism, combinatorial action space and high dimensionality of the state space (estimated to have at least 10^{1685} states), this game has always been considered hard to tackle for AI, and has gained more and more popularity as a test bed for AI techniques. DeepMind and Blizzard Entertainment in 2017 released the *StarCraft II Learning Environment* (SC2LE), a reinforcement-learning environment based on the successor of StarCraft, to be used for research purposes. Moreover, in early 2019 DeepMind announced that their StarCraft II agent, AlphaStar, based on a deep neural network and trained by supervised learning and reinforcement learning, had defeated two top professional players (Vinyals *et al.*, 2019). This achievement shows the potential for a game-playing agent to reach the strategical skills of human players, although, during the matches, AlphaStar still had the advantage of being able to act faster and had a higher precision than the human opponents had.

Regarding *first-person shooter* games, a known competition is the *Visual Doom AI Competition* (Kempka *et al.*, 2016), that evaluates how agents can play a game based on Doom by only having access to the screen buffer. This competition is particularly interesting to test techniques to play 3D video games, where agents have to navigate the environment. Finally, for the category of *racing* games, since 2007 the *Simulated Car Racing Championship* was established, and since 2008 the framework has been based on the TORCS racing game. Participants can submit car-racing controllers that are evaluated on different racing tracks.

1.3 Games and Artificial General Intelligence

The research discussed in the previous section and all the mentioned game-playing programs have one thing in common: game specificity. Game-specific programs might perform extremely well on one game but might not be able to deal with any other game. For many years, the vast majority of game AI research and, more in general, of the whole AI research field focused on the so-called "narrow AI", which

consists of developing specialized programs that show intelligence only on specific tasks. Some of these programs are quite successful, but in general they incur the risk of overfitting to the problem. Often, they tend to be refined using domain-specific knowledge and engineering tricks, and usually most of the knowledge is coded by the programmer instead of being learned by the program. This leaves less room for actual AI and makes the programs not applicable to any other task unless major modifications are first performed. This picture recently led some researchers to concentrate on more general AI, defining a new research area called *Artificial General Intelligence* (AGI) (Goertzel and Pennachin, 2007). This research area tackles the creation of programs that are able to perform different complex tasks in different environments and learn how to adapt to perform each task. Similarly, game AI research has been focusing on creating programs that are able to play many different games, fostering research in the area of *general game playing* (GGP). Given that search and planning are among the main competences that AGI should be able to display, using GGP to improve the general applicability of search and planning methods could be considered as one of the first stepping stones to reach AGI.

Having gained popularity only recently, most of the work on GGP has been performed in the past ten to fifteen years. However, the concept of GGP programs was proposed already in the 1960s by Pitrat (1968). He described a program able to play several games on a bidimensional board by being given their rules, and stresses the need for such a program to use general heuristics and to be able to study the game rules on its own. Moreover, in 1992 Barney Pell highlighted the problem of evaluating general intelligence of a game-playing program when most of the analysis of the game is actually performed by humans (Pell, 1993). In his research he proposed a framework called *Metagame* meant to evaluate multiple agents against each other on a set of different games. Agents should be able to play each game by just receiving its rules and without any human intervention. Pell also presented a generator for symmetric chess-like games to be used within the *Metagame* framework and a game-playing agent called METAGAMER, able to analyze game rules and derive a representation and an evaluation function for the game.

A commercial GGP system was released a few years later, in 1998, and is still available nowadays: *Zillions of Games* (Mallett and Lefler, 1998). This system uses a "Universal Gaming Engine" technology that enables users to play many types of puzzles and board games. Other than offering a variety of games, the system enables users to define their own games by writing the rules in a functional language. The engine then parses these rules so that the built-in agent can play the games. This system provides a useful tool for board and abstract game design. By letting the engine play the game against itself designers can test different properties. For example, if the game can be solved, if it is a forced win for any of the players, etc. However, since its release the system has not been used much for research purposes. This might be due to some of its limitations, which include the possibility to only represent perfect-information games, no support for arithmetic, functions and variables, and no support to create connection games, such as Hex.

A concept similar to *Metagame* and *Zillions of Games*, and probably one of the more well-known efforts of game AI research in providing a standard for GGP in abstract games is the GGP project of the Logic Group of Stanford University (Stanford

GGP project) (Genesereth, Love, and Pell, 2005). This project aims at developing agents that can play any arbitrary (finite, deterministic and perfect-information) game by being given the rules at run-time, thus assuming that the agent is seeing each game for the first time and that no prior, game-specific knowledge can be exploited. This project also defines the *Game Description Language* (GDL) (Love, Hinrichs, and Genesereth, 2006) to represent game rules, and a communication protocol that agents have to follow to interact with a central server that acts as a mediator and manages the game matches. To foster research on the topic, since 2005 an annual competition has been established, that matches GGP agents against each other to evaluate their strength.

Recently, two alternatives to the Stanford GGP project have been presented to facilitate and promote research in GGP for abstract games. Kowalski *et al.* (2019) have proposed a new language to represent board games, called *Regular Boardgames*. This language has been developed with the intent of overcoming some limitations of other existing game description languages, which in some cases lack expressiveness or efficiency, or make the game structure difficult to understand for users. At the same time, a new general game system, called *Ludii*, has been proposed (Browne, 2018; Piette *et al.*, 2019). This system defines games as structures of ludemes, i.e. high-level, easily understandable game concepts. Ludii enables modeling and play of a large variety of strategic games, and provides a more efficient and clear representation of their rules with respect to GDL. This makes it a promising benchmark for game AI, although no official GGP competition yet exists based on this system.

As seen for research on game-specific programs, also research on general game AI started expanding its horizon to more complex and challenging domains: video games. In 2012 the *Arcade Learning Environment* (ALE) was designed (Bellemare *et al.*, 2013). This framework provides an interface with a set of video games for the Atari 2600 console and expects the agents to be able to play all the given games. It allows agents to send the movement of the joystick as input and returns the pixel representation of the screen as output. Thus, agents have to include also a mechanism to interpret this information. A notable achievement regarding the ALE framework is the program based on deep neural networks and Q-learning designed by Google DeepMind, which managed to perform better than or equal to professional human players in more than half of the tested games (Mnih *et al.*, 2015). One of the limitations of ALE, however, is that the number of available games is quite small and it is not trivial to add more games. This makes it easy to tune the agents for single games in advance. Most of the research on ALE has indeed been limited to the creation of agents that share the same structure (usually neural networks), but still have parameters (e.g. network weights) learned for each specific game, which seems to go against the purpose of GGP (Yannakakis and Togelius, 2018).

To foster research on GGP for video games, a Dagstuhl seminar was organized in 2012 to establish the new research area of General Video Game Playing (GVGP) (Levine *et al.*, 2013). Similarly to the Stanford GGP project, GVGP aims at creating agents that are able to play many different video games that they might have never seen before. A *Video Game Description Language* (VGDL) has also been formalized to specify rules and levels of the video games (Ebner *et al.*, 2013; Schaul, 2013). To promote research in the area of GVGP, since 2014 the General Video Game-AI

(GVG-AI) competition has been running (Perez-Liebana *et al.*, 2016) as part of the GVG-AI project. This competition focuses on 2D arcade-style video games. The first editions of the competition offered only a single-player planning track, where agents had to play games using a forward model to simulate the effect of the actions on the game and plan their next action. In subsequent editions more tracks have been added, including a 2-player planning track, a single-player learning track, a game generation track and a level-generation track.

Recently, more and more frameworks have started appearing to test general game AI approaches. In 2016, the Retro Learning Environment (RLE) was proposed (Bhonker, Rozenberg, and Hubara, 2016). It is similar to ALE, but is based on the Super Nintendo Entertainment System and provides more complex games. In the same year Google DeepMind released the *DeepMind Lab* platform (Beattie *et al.*, 2016). This platform provides a set of first-person 3D games to benchmark GGP agents in different, large and partially observable environments. Since 2016, Open AI also released two platforms to support AI research, *OpenAI Gym* (Brockman *et al.*, 2016) and *OpenAI Gym Retro* (OpenAI, 2018). The first acts as interface for a set of various benchmark games on which reinforcement-learning algorithms can be tested. The second is an extension of *OpenAI Gym* that includes a large selection of classic video games.

1.4 Search Techniques

Most of the research on Game AI sees playing a game as a search problem, and deciding which actions to play means exploring the game tree with a *tree-search algorithm*. As mentioned in Section 1.2, initially all research in Game AI focused mainly on two-player board games with perfect information, deterministic, turn-based and with discrete search and action space, like Chess and Checkers. The foundation of most of the tree-search algorithms that enabled computer programs to reach a good performance in such board games is the *minimax* algorithm (Neumann and Morgenstern, 1944). This algorithm explores the game tree up to a certain depth in a breadth-first manner, alternating between states where the root player, identified as MAX, has to move and states where the opponent, identified as MIN, has to move. The (estimated) payoff obtained by the MAX player in leaf nodes is propagated backward in the tree considering that on her turn MAX always tries to maximize her score, while MIN always tries to minimize MAX's score. The most popular variant of minimax search for deterministic two-player games is *$\alpha\beta$-search* (Knuth and Moore, 1975), which uses a pruning technique to speed up minimax by excluding from the search the branches of the tree that are proved to have no influence on the final result of the game.

There are many games, however, for which minimax and its variants are not successfully applicable. First of all, for games with a high branching factor and a large state space minimax, even when enhanced with pruning techniques, cannot visit a sufficient portion of the tree to choose an action with an informed decision. Moreover, minimax and its variants rely on game-specific *heuristic evaluation functions* to compute the payoff of game states and there are games, like Go, for which

designing such heuristic is difficult, and domains, like GGP, where no game-specific information is available in advance. To avoid the need for pre-coded game-specific knowledge in heuristic evaluation functions the use of *Monte-Carlo evaluations* was proposed (Abramson, 1990). These evaluations estimate the value of a state by randomly simulating the game until a terminal state is reached and can be directly used to evaluate states in the minimax algorithm.

Monte-Carlo evaluations have been used as sampling strategy to develop other Monte-Carlo Search (MCS) algorithms. The simplest one is *Flat Monte-Carlo Search* (Flat-MCS), which samples all actions in a state of the game uniformly at random using Monte-Carlo evaluations, and then considers as the best action the one that obtained the highest average payoff. Flat-MCS has been successfully applied to Bridge (Ginsberg, 2001) and Scrabble (Sheppard, 2002). However, the limitation of Flat-MCS and in general of the random Monte-Carlo evaluations of game states is that, even when the number of samples is large, there is no guarantee that the action with the highest estimated value corresponds to the game-theoretic optimum (Browne, 2010). More advanced sampling techniques have also been used together with MCS, like UCB1 (Auer, Cesa-Bianchi, and Fischer, 2002), that can balance exploitation of actions with a high estimated payoff and exploration of less visited actions that might turn out to be better in the future, once more samples are collected. Convergence to the game-theoretic optimum is not guaranteed with UCB1 either, but it enables MCS to converge faster to the action with the highest Monte-Carlo evaluation. The UCB1 strategy, together with Monte-Carlo evaluations and the idea of building a tree structure during the search has posed the basis for the creation of MCTS (Coulom, 2007a) and one of its most well-known action selection strategies, UCT, i.e. UCB applied to Trees (Kocsis and Szepesvári, 2006).

MCTS is a tree-search technique that incrementally builds a tree representation of the game space by iterating over the following four phases: selection, expansion, play-out and backpropagation. During selection a strategy is used to decide how to traverse the tree built so far. When an unvisited state is reached, a corresponding node is added to the tree during expansion. A Monte-Carlo evaluation is performed during the play-out to evaluate the new node, and the result is propagated in the tree during backpropagation to update the action statistics that will be used by the selection strategy in the next iteration. In its basic form MCTS is (i) *aheuristic*, i.e. it does not require any game-specific knowledge, (ii) *anytime*, i.e. it can choose the action to be played within any time budget, and (iii) *selective*, i.e. it favors regions of the search tree that have the most promising actions, growing the tree asymmetrically. Its selectivity makes it more suitable than minimax to tackle games with a high branching factor, like Amazons (Lorentz, 2008).

Due to its characteristics, MCTS has been successfully applied to many games. The most popular is the game of Go (Coulom, 2007a), for which MCTS represented a substantial step forward. Other examples are Hex (Arneson, Hayward, and Henderson, 2010), Havannah (Teytaud and Teytaud, 2010), Lines of Action (Winands, Björnsson, and Saito, 2010) and Ms. Pac-Man (Pepels, Winands, and Lanctot, 2014). MCTS has also been shown to be particularly suitable for GGP, where it has seen successful applications both on board games (Finnsson and Björnsson, 2010) and video games (Perez-Liebana *et al.*, 2016). Other than games, MCTS has been

applied to a wide range of domains, like robotics (Goldhoorn *et al.*, 2014), logistics (Edelkamp *et al.*, 2016), transportation (Trunda and Barták, 2013), medical planning (Zhu, Lizotte, and Hoey, 2014), chemistry (Segler, Preuss, and Waller, 2018) and space exploration (Hennes and Izzo, 2015).

1.5 Problem Statement and Research Questions

Previous sections discussed the relevance of games in the field of AI and AGI, and general game playing was identified as a suitable domain to test search techniques that support AGI. Moreover, MCTS was presented as a successful technique for domains like GGP, where no specific domain knowledge is available. This thesis focuses on enhancing MCTS for AGI in games, with application to general game playing. The following problem statement guides the research.

> **Problem statement**: *How can the performance of Monte-Carlo Tree Search for general game playing be improved?*

To answer the problem statement four research questions have been formulated. They deal with (1) speeding up the interpretation of game rules written in a declarative language, (2) evaluating the use of global or local information to enhance the selection strategy of MCTS, (3) on-line tuning search-control parameters for MCTS, and (4) investigating the effect of search-control parameter randomization in MCTS.

> **Research question 1**: *How can the process of interpreting on-line the game rules written in a declarative language be sped up?*

GGP requires to define a formal language to encode the rules of the games it considers. One possible approach to define game rules is the use of a declarative language, of which the *Game Description Language* (GDL) (Love *et al.*, 2006) is a well-known example. GDL represents game rules as a collection of logical rules. An advantage of using this approach is that it enables game-playing agents to extract knowledge from the game rules by logical deduction. When describing the game rules using a declarative language as GDL game-playing agents cannot directly get all the elements that are necessary to reason about a game (i.e. future game states, legal actions and goals for the players, etc.). They have to include a mechanism to interpret such rules. A limitation of using a declarative language to describe the game rules is that the interpretation process is usually very slow, and this might hinder the performance of MCTS. Although MCTS can choose an action at any time, the quality of its choice depends on how many simulations can be performed (Robilliard, Fonlupt, and Teytaud, 2014). More simulations mean more accurate estimates of the collected statistics on which the final action selection is based. How fast an agent can interpret the game rules to reason on the game directly influences the number of simulations that can be performed in a given amount of time. To increase the reasoning speed, game-specific agents usually exploit game-specific characteristics, and are therefore much faster than GGP agents that have to interpret a declarative language like GDL (Schiffel and Björnsson, 2014). This thesis deals with the problem of speeding up the interpretation process of game rules written in GDL, such

that MCTS could benefit from a higher number of simulations. To answer the first research question, an agent for the Stanford GGP project is considered, for which a game rule interpreter based on Propositional Networks (PropNets) (Schkufza, Love, and Genesereth, 2008; Cox *et al.*, 2009; Genesereth and Thielscher, 2014) is investigated. Moreover, four optimizations for the structure of such PropNets are investigated. Finally, an implementation of the PropNet structure on a Field Programmable Gate Array (FPGA), which has the potential of further increasing the simulation speed of an MCTS agent, is presented.

> **Research question 2**: *What is the effect of using locally or globally collected information to enhance the selection strategy for Monte-Carlo Tree Search?*

During the selection phase of MCTS the tree is traversed from the root to a leaf node. The selection strategy decides how the tree is traversed during this phase by selecting which action to visit in each node. Many selection strategies have been proposed for MCTS, among which UCT (Kocsis and Szepesvári, 2006) is one of the most popular. Previous research has shown that enhancing the UCT strategy by increasing the amount of information used to guide the search can consistently improve the overall performance of MCTS (Finnsson and Björnsson, 2010; Nijssen and Winands, 2011; Gelly and Silver, 2011; Cazenave, 2015). Rapid Action Value Estimation (RAVE) (Gelly and Silver, 2007) is among the strategies proposed to enhance the UCT selection. In each tree node, RAVE uses locally collected information about the general performance of the actions to bias the selection. RAVE has proved successful, other than in specific domains like Go (Gelly and Silver, 2011), also when applied to the Stanford GGP project (Finnsson and Björnsson, 2010). Recently, a generalization of RAVE has been proposed, Generalized Rapid Action Value Estimation (GRAVE) (Cazenave, 2015). When a tree node has only a small number of visits, GRAVE uses more global information than RAVE to bias action selection. This strategy has been shown to perform better than RAVE on some variants of Go and some other games. To answer the second research question, this thesis first proposes another variant of RAVE, History Rapid Action Value Estimation (HRAVE), which biases action selection always using global information about the actions. Subsequently, it compares the performance of RAVE, GRAVE and HRAVE against each other. These RAVE variants are tested in the framework of the Stanford GGP project.

> **Research question 3**: *How can search-control parameters for Monte-Carlo Tree Search be tuned effectively on-line?*

Many enhancements for the different phases of MCTS have been applied successfully in GGP (Finnsson and Björnsson, 2010; Tak, Winands, and Björnsson, 2012; Soemers *et al.*, 2016). Often, MCTS and its enhancements are controlled by multiple parameters that require extensive and time-consuming off-line optimization. Moreover, as the played games are unknown in advance, off-line optimization cannot tune parameters specifically for single games. It has to find values that perform overall well on a predefined set of games, with no guarantee that they will perform

successfully also on unseen games. To answer the third research question, this thesis proposes a Self-Adaptive MCTS strategy (SA-MCTS) that integrates within the search a method to automatically tune search-control parameters on-line per game. Seven different allocation strategies are presented, which decide how to allocate the available samples to evaluate the different values of all the tuned parameters. What these strategies have in common is that they are all designed to balance exploitation of parameter values that seem to perform well for the game at hand and exploration of parameter values that have been evaluated less. SA-MCTS is evaluated both in the framework of the Stanford GGP project and in the framework of the GVG-AI project.

> **Research question 4**: *What is the effect of randomizing search-control parameters for Monte-Carlo Tree Search?*

Previous research has shown that adding randomization to certain components of the search might increase its diversification and improve its performance. For example, Beal and Smith (1994) showed that the addition of a random term to the heuristic function of minimax is able to capture some aspects of the structure of the tree, biasing the search towards states where the player has more available actions (i.e. more mobility). Moreover, Bošanský *et al.* (2016) improved MCTS by modifying the action-selection strategy to randomly select an action among the ones with the highest values. This causes MCTS to diversify the strategies that are explored for both players. Finally, Chen (2012) added randomization to diversify how the selection strategy and play-out strategy of an MCTS agent sample the available actions. This is shown to be beneficial for the search for the game of Go. In a domain that tackles many games with different characteristics, like General Game Playing (GGP), trying to diversify the search adding some randomization might be a good strategy for some games. Furthermore, also the on-line search-control parameter tuning mechanism used by SA-MCTS is adding a random component to the search whenever it explores different parameter combinations. Therefore, it is interesting to verify what happens when parameters are always selected randomly instead of making informed choices. To answer the fourth research question, this thesis tests four different strategies to randomize search-control parameters in MCTS: randomization per game, per turn, per simulation and per state. Moreover, search-control parameter randomization is compared with fixed parameter settings and with on-line parameter tuning both in the framework of the Stanford GGP project and in the framework of the GVG-AI project.

1.6 Thesis Overview

This thesis is organized into eight chapters. Chapter 1 provides an introduction to games, discusses their relevance for AI and AGI and gives a brief description of the most popular search techniques used by game AI programs. Moreover, it presents the formulation of the problem statement and the four research questions that guide the research reported in this thesis.

Chapter 2 discusses tree-search techniques. First, a formal definition for the games used in this thesis is given. Subsequently, Monte-Carlo methods are described and linked to the development of MCTS, which is presented next. Finally, the chapter gives an overview of one of the most well-known MCTS selection strategies, UCT, and of a selection of enhancements for different parts of MCTS that are relevant for this thesis.

Chapter 3 introduces the test environments used for the experiments: the Stanford GGP project and the GVG-AI project. For each of them, the chapter presents the language used to describe games, the process to manage the execution of a game run, the set-up of the corresponding competition and the characteristics of the agent(s) that are used in subsequent chapters for the experiments. Finally, the chapter ends with a discussion of interesting research directions to improve the performance of MCTS in the presented environments.

Chapter 4 addresses the first research question. To speed up the interpretation process of GDL, an interpreter based on the representation of the game rule as a PropNet is investigated. First, four optimizations for the structure of the PropNet are evaluated, and their best combination is identified. Subsequently, an implementation of the optimized PropNet on an FPGA board is presented. Both the optimized PropNet and its implementation on an FPGA are compared with a custom-made rule interpreter, the Prover. All the interpreters are tested on a subset of games of the Stanford GGP environment.

The second research question is addressed in Chapter 5. Three enhancements for the selection strategy of MCTS are evaluated, RAVE, GRAVE and HRAVE, of which HRAVE is newly proposed. These enhancements share the same biasing mechanism for the selection of actions in a node, but differ in how information is collected. RAVE uses local information, GRAVE varies between using local and global information and HRAVE uses global information. The performance of these three enhancements is tested also in combination with a more informed play-out strategy. These strategy are compared on a selection of games from the Stanford GGP project to investigate how using local or global information affects the search.

Chapter 6 addresses the third research question. This chapter proposes the design of SA-MCTS by using a mechanism that tunes search-control parameters on-line. Seven strategies to allocate available samples to evaluate parameter value combinations on-line are proposed and compared. The allocation strategy that performs best is also tested against a successful GGP agent, CadiaPlayer. SA-MCTS is tested on a subset of games taken both from the Stanford GGP project and from the GVG-AI project.

The fourth research question is addressed in Chapter 7. Four strategies to randomize search-control parameters for MCTS are presented in this chapter and compared against each other. The strategy that performs best is further investigated and compared with on-line parameter tuning both directly and against different types of opponents. Moreover, the chapter analyzes how parameter randomization influences the structure of the tree built by MCTS. Experiments are performed both on games taken from the Stanford GGP project and games taken from the GVG-AI project.

Finally, Chapter 8 answers the four research questions and the problem statement. It also gives an overview of possible future research directions.

Additionally, Appendix A gives an example of a GDL game description for the Stanford GGP project and an example of a VGDL game description for the GVG-AI project. Moreover, the rules and the relevant characteristics of the games for the Stanford GGP project and the GVG-AI project used in the experiments are reported in Appendix B. Finally, additional results for Chapters 4, 5, 6 and 7 are presented in Appendices C, D, E and F, respectively.

Chapter 2

Search Techniques

Parts of this chapter are based on:

- Sironi, Chiara F., Liu, Jialin, and Winands, Mark H.M. (2019). Self-Adaptive Monte-Carlo Tree Search in General Game Playing. *IEEE Transactions on Games*, In press.
- Sironi, Chiara F. and Winands, Mark H.M. (2016). Comparison of rapid action value estimation variants for General Game Playing. *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pp. 309–316.
- Soemers, Dennis J.N.J., Sironi, Chiara F., Schuster, Torsten, and Winands, Mark H.M. (2016). Enhancements for Real-Time Monte-Carlo Tree Search in General Video Game Playing. *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pp. 436–443.

This chapter introduces the search techniques used in this thesis. It focuses on the description of MCTS and of the methods from which it originated. It also presents some of the enhancements for MCTS that have been proposed in the literature and that are necessary to understand subsequent chapters of this thesis.

The chapter is organized as follows. First, Section 2.1 gives a formal definition of the considered games and introduces the terminology used throughout this thesis. Next, Section 2.2 introduces Monte-Carlo methods and their link to MCTS, while the MCTS algorithm is described in Section 2.3. Subsequently, the UCT action selection strategy for MCTS is presented in Section 2.4, and relevant MCTS enhancements are presented in Section 2.5. Finally, Section 2.6 discusses the need for an appropriate environment to test search strategies and their enhancements.

2.1 Tree Search in Games

The games considered in this thesis can be formally defined as reported below. The given definition assumes that the games are discrete and in each turn all players have to move simultaneously by performing a *joint action*. In a game with r players,

a joint action is represented as a tuple $\vec{a} = \langle a_1, ..., a_r \rangle$, where each a_i is the action performed by player i. This definition can also model games where not all players execute an action at the same time. This is done by assuming that players that do not perform an action in a turn are actually performing a *null* action[1] that has no effect on the game. All the search techniques considered in this thesis are based on the given formal definition, expressed by the following components:

- $\boldsymbol{\mathcal{R} = \{1, ..., r\}}$: the set of r players. In this thesis we use increasing indices to represent the players.
- $\boldsymbol{\mathcal{S}}$: the set of all possible states in the game.
- $\boldsymbol{\{\mathcal{A}_1, ..., \mathcal{A}_r\}}$: for each player $i \in \mathcal{R}$, a set of actions that are legal for i in at least one state during the game.
- $\boldsymbol{s_0 \in \mathcal{S}}$: the initial state.
- $\boldsymbol{legal(s, i) : \mathcal{S} \rightarrow \mathscr{P}(\mathcal{A}_i)}$: a function that given a state $s \in \mathcal{S}$ and a player $i \in \mathcal{R}$ returns the set $\mathcal{A}_{(s,i)} \subseteq \mathscr{P}(\mathcal{A}_i)$ of actions that are legal in s for player i.
- $\boldsymbol{next(s, \vec{a}) : \mathcal{S} \times (legal(s, 1) \times ... \times legal(s, r)) \rightarrow \mathcal{S}}$: a *transition function* that given a state $s \in \mathcal{S}$ and a joint action $\vec{a}$ of all players returns the next state of the game s' reached by performing the joint action $\vec{a}$ in s. Note that for deterministic games this function always returns the same state, while for non-deterministic or stochastic games each time it will return one of the possible next states according to the probability distribution over next states.
- $\boldsymbol{terminal(s) : \mathcal{S} \rightarrow \{true, false\}}$: a function that given a state $s \in \mathcal{S}$ determines whether the state is terminal, i.e. the game is over in the state.
- $\boldsymbol{payoff(s) : \mathcal{S} \rightarrow \mathbb{R}}$: a payoff function that given a state $s \in \mathcal{S}$ returns a tuple $\vec{q} = \langle q_1, \ldots, q_r \rangle$ with the payoff obtained in s by each of the players.

The functions listed above to define a game are frequently used by the search techniques presented in this thesis. When reporting pseudocode, the methods that implement the functions $legal(s, i)$, $next(s, \vec{a})$, $terminal(s)$ and $payoff(s)$ are called GetLegalActions(s, i), GetNextState($s, \vec{a}$), Terminal(s) and Payoff(s), respectively. This thesis assumes that all presented search techniques have access to a *game model*, also called *forward model*, that offers the implementation of such methods for the game being played.

Knowing the game components and starting with the initial state s_0 we can build the *game tree* that represents all possible ways the game can be played. Figure 2.1 shows a representation of a game tree. Each *node* corresponds to a state in the game and each *edge* corresponds to a (joint) action. The initial state of the game, s_0 is represented by the *root* node of the tree. Starting from the root node, for each action available in the corresponding state we can add one edge to the tree, and for each state reached by applying one of such actions we can add a new node to the tree.

[1] A *null* action in the literature is also commonly referred to as *noop* or *pass*.

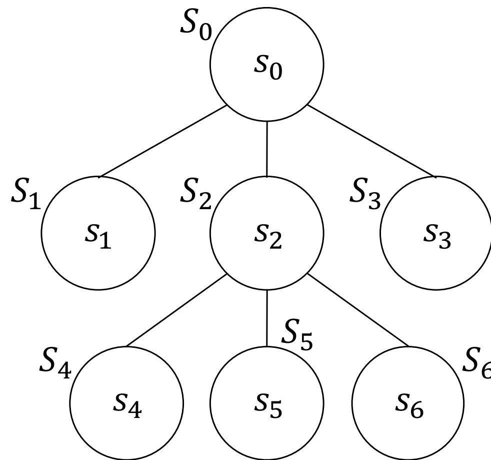


Figure 2.1: Representation of a game tree.

By repeating this process recursively on each new node the complete game tree can be built. Note that with this procedure, if a state can be reached from the initial state with different sequences of actions, multiple separate nodes will be created in the tree that represent the same state.

When talking about game trees it is useful to define the following commonly used terms:

- Consider a tree node S_i representing state s_i and a tree node S_j representing state s_j. If state s_j can be reached from state s_i with an action, then node S_i is called *parent* of node S_j, and node S_j is called *child* of node S_i. Each node in the tree has one and only one parent, except the *root* that has no parent. A node can have zero, one or multiple children. Nodes with the same parent are called *siblings*. For example, in Figure 2.1 node S_0 is the parent of node S_2, node S_2 is a child of node S_0, and node S_3 is a sibling of node S_2.

- A node can be identified as an *ancestor* of node S_i if it is S_i itself, its parent or an ancestor of its parent. Conversely, all nodes that have node S_i as ancestor are identified as its *descendants*. A node S_i, together with all its descendants is identified as the *subtree* rooted in S_i. In Figure 2.1, for example, node S_0 is ancestor of node S_5, which is its descendant. The subtree rooted in S_2 includes the nodes S_2, S_4, S_5 and S_6.

- All nodes that have at least one child are called *internal nodes*. Nodes with no children are identified as *leaf nodes*. In a game tree, leaf nodes correspond to terminal game states, thus are also called *terminal nodes*. In terminal nodes the game is over and the payoff of each player is known. In Figure 2.1, S_2 is one of the internal nodes, while S_1 is one of the leaf nodes.

Game trees can have a very large number of nodes, which makes it infeasible to visit the full tree. Game-playing programs usually use a *search technique* to visit only

a portion of the game tree called *search tree*. During each turn of the game, a search technique incrementally builds the search tree starting from the node representing the current game state as the root. Action and state statistics might be memorized in each node and used to guide future search. The search runs until a search budget has been consumed, for example until a certain amount of time has passed, or until the tree has been visited up to a certain depth. It is relevant to point out that search trees, as opposite to game trees, can have non-terminal leaf nodes. From such nodes the search algorithm can continue the search and expand the tree.

2.2 Monte-Carlo Methods

Monte-Carlo methods were first applied in physics to approximate intractable integrals (Metropolis and Ulam, 1949). Since then they have been applied in many other domains, including games. Abramson (1990) proposed to use them during tree search to evaluate game states in place of game-specific heuristics. The idea behind Monte-Carlo evaluations is to perform multiple simulations of the game starting from the state that we want to evaluate until a terminal state is reached. The payoffs obtained by the simulations are then averaged to get an estimate of the state value.

Monte-Carlo evaluations have been used to develop numerous search techniques, among which *Flat Monte-Carlo Search* (Flat-MCS) (Browne *et al.*, 2012) is one of the simplest. Given a game state, Flat-MCS performs a 1-ply search, sampling all legal actions uniformly at random with Monte-Carlo evaluations. It then picks the one with the highest average payoff. The performance of Flat-MCS can be improved by replacing the uniform sampling strategy with one that balances exploration of less sampled actions that might turn out to be more profitable and exploitation of actions that have obtained a high average payoff so far. Multi-Armed Bandit (MAB) problems (Auer *et al.*, 2002) present the same exploration vs exploitation dilemma. Algorithms designed to handle these problems can be integrated with MCS methods as well. This idea is behind the creation of the MCTS algorithm (Kocsis and Szepesvári, 2006; Coulom, 2007a).

Subsection 2.2.1 describes Flat-MCS, while Subsection 2.2.2 introduces the *Multi-Armed Bandit* algorithms and relates them to tree search, giving the basis to understand how MCTS originated.

2.2.1 Flat Monte-Carlo Search

Flat-MCS can be used to choose which actions to play in a game (Browne *et al.*, 2012). Algorithm 1 gives the pseudocode of Flat-MCS that can be applied to games with any number of players, with either simultaneous or sequential moves, and either deterministic, stochastic or non-deterministic. Given the game state s where an action has to be chosen and the player performing the search i, the procedure MonteCarloSearch(s, i) samples legal joint actions uniformly at random until the search budget expires. Examples of search budget could be a limited amount of time or a given number of samples. This procedure uses two types of variables to memorize in the root node statistics about the performance of the actions. For each action a_i of player i, the variable $qSum_{a_i}$ keeps track of the sum of payoffs

```
1: procedure MonteCarloSearch(s, i)
   Require: The game model gm.
   Input: Current root state s, player performing the search i ∈ R.
   Output: The action to play in state s.
2:     while search budget available do
3:         a⃗ ← random legal joint action of all the players
4:         s′ ← gm.GetNextState(s, a⃗)
5:         q⃗ ← PlayOut(s′)
6:         qSum_{a_i} ← qSum_{a_i} + q_i
7:         n_{a_i} ← n_{a_i} + 1
8:     A_{(s,i)} ← gm.GetLegalActions(s, i)
9:     return argmax_{a_i ∈ A_{(s,i)}} { q̄_{a_i} = qSum_{a_i} / n_{a_i} }

10: procedure PlayOut(s)
   Require: The game model gm.
   Input: The game state s to evaluate.
   Output: A tuple q⃗ = ⟨q_1, ..., q_r⟩ with the payoffs obtained at the end of the
   play-out by each player.
11:     if gm.Terminal(s) then
12:         return gm.Payoff(s)
13:     else
14:         a⃗ ← random legal joint action of all the players
15:         s′ ← gm.GetNextState(s, a⃗)
16:         return PlayOut(s′)
```

Algorithm 1: Pseudocode for Monte-Carlo Search.

obtained by all simulations in which action a_i was played in the initial state, while the variable n_{a_i} keeps track of the number of times action a_i was selected in the initial state. These two variables are used to compute the average payoff $\bar{q}_{a_i}$ of each action a_i of player i as in Formula 2.1.

$$\bar{q}_{a_i} = \frac{qSum_{a_i}}{n_{a_i}} \tag{2.1}$$

More precisely, for each iteration, Flat-MCS selects a random legal joint action $\vec{a}$ of the players, where player i plays one of its legal actions a_i. Then it computes the next state s' reached by performing the joint action $\vec{a}$ in state s. From state s' a play-out of the game is performed, using a Monte-Carlo evaluation and obtaining a tuple $\vec{q} = \langle q_1, \ldots, q_r \rangle$ of payoffs, one for each player. The payoff q_i of the player performing the search is added to $qSum_{a_i}$ and the counter n_{a_i} is incremented by 1. At the end of the search, the legal action of player i that has the highest $\bar{q}_{a_i}$ is returned to be played in the game.

The PlayOut(s) procedure, given a state s, performs a Monte-Carlo evaluation. If the state is terminal, this procedure returns the tuple with the payoffs of each player in the state. Otherwise it selects a random joint action, uses it to compute

the next state and recursively calls itself on the new state. Note that the presented Flat-MCS algorithm can be applied also to single-player games by considering that the joint action and payoff tuples will contain a single element, and to sequential move games by considering that whenever a player is not on its turn it will play a *null* action with no effect on the game.

2.2.2 Multi-Armed Bandit Algorithms

The MAB problem (Auer *et al.*, 2002) with m arms is defined as a set of m unknown independent real payoff distributions $\mathcal{Q} = \{Q_1, ..., Q_m\}$, each of which is associated to one of the arms. When one of the arms is played a payoff is obtained as a sample of the corresponding distribution.

The aim of a sampling algorithm for a MAB problem is to maximize the cumulative payoff obtained by successive plays of the arms as quickly as possible. To do so, for each iteration the algorithm can choose which arm to play depending on past played arms and obtained payoffs. This originates the *exploration vs exploitation dilemma*: the algorithm should try to exploit promising arms more often, while still trying to explore other arms that might have been unlucky so far, but actually have a high average payoff. Solving the exploration vs exploitation dilemma means finding an algorithm that minimizes the regret of the player over time, which is the expected payoff loss due to not playing the best arm.

Among the algorithms that have been proposed for the MAB problem, the family of the *Upper Confidence Bound* (UCB) algorithms includes some of the most popular ones. One of them, and also one of the most used, is UCB1 (Auer *et al.*, 2002), that in each iteration assigns a value, $UCB1(a)$ to each arm a, and selects which arm a^* to play as shown in Formula 2.2.

$$\begin{aligned} a^* &= \operatorname*{argmax}_{a \in A} \left\{ UCB1(a) \right\} \\ UCB1(a) &= \bar{q}_a + \sqrt{\frac{2 \ln n}{n_a}} \end{aligned} \tag{2.2}$$

In this formula, A is the set of available arms, $\bar{q}_a$ is the average payoff over all the plays of arm a, n is the number of samples taken so far for all arms, and n_a is the number of samples taken so far for arm a. The first term of the formula, $\bar{q}_a$, is the exploitation term that rewards arms with a high average payoff, while the second term is the exploration term that rewards arms that have been visited less. An interesting property of UCB1 is that, when the payoff distributions have values in the interval $[0, 1]$, the algorithm is guaranteed to achieve logarithmic regret uniformly over time. A logarithmic increase has been proved to be the best possible lower bound on the growth of the regret for MAB problems (Lai and Robbins, 1985).

The UCB1 algorithm can also be used as action selection strategy in MCS algorithms, instead of the uniformly random selection strategy. In a game, the problem of selecting an action in a state for a player can be seen as a MAB. The player has a number of available actions (i.e. the arms) and for each iteration of the search has to select the one that maximizes the payoff, while still making sure to explore less visited ones in case they turn out to be better. Once an action is selected by

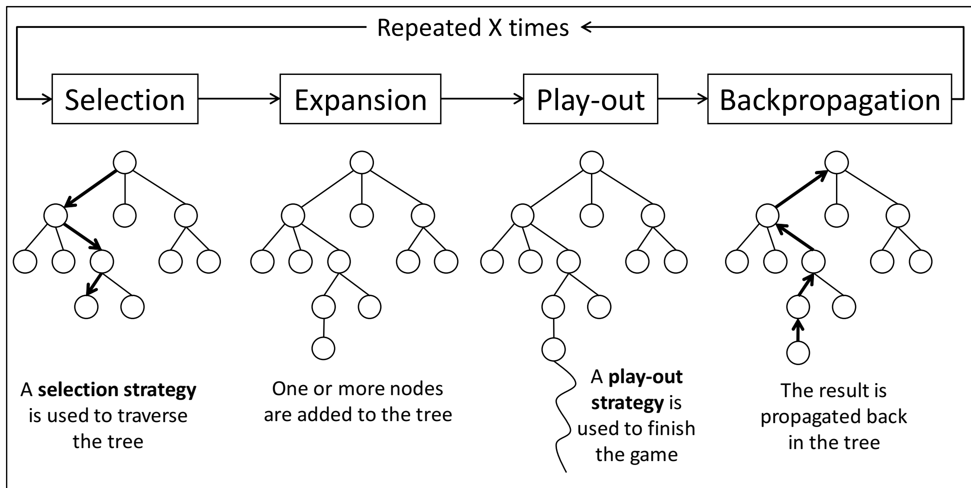


Figure 2.2: Outline of Monte-Carlo Tree Search (inspired by Chaslot *et al.*, 2008b).

the UCB1 algorithm, its payoff can be computed using Monte-Carlo evaluations on the state that results from the application of such action. In addition, UCB1 can be used to select the actions for the opponents in two- and multi-player games. This, together with the idea of building a tree that memorizes part of the visited states together with statistics about visited actions, is the basis of the MCTS algorithm (Coulom, 2007a) and of one of its most popular action selection strategies, UCT (Kocsis and Szepesvári, 2006).

2.3 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a simulation-based search method used to select which action to play in a given game state (Kocsis and Szepesvári, 2006; Coulom, 2007a; Chaslot *et al.*, 2008b). Starting from the given state, MCTS incrementally builds a tree representation of the search space of the game. Each iteration of the algorithm performs a complete simulation[2] of the game from the given root state to a terminal state, adding nodes to the tree after each simulation and collecting information about the game in every node. At the end of the search, the collected information is used to select which action to play. Each iteration of the MCTS algorithm consists of the four phases shown in Figure 2.2: *Selection*, *Expansion*, *Play-out* and *Backpropagation*. Below we give a detailed description of each phase:

Selection. During this phase the algorithm descends the tree built so far until it reaches a node that requires expansion. At each node it uses a *selection strategy* to determine which joint action to visit next. Multiple selection strategies

[2]Sometimes in the literature the term *simulation* is used as a synonym for *play-out*, which is one of the four phases of MCTS. In this thesis, instead, we use the terms *iteration* and *simulation* interchangeably to indicate a complete repetition of the four MCTS phases.

have been proposed in the literature, many of which derive from multi-armed bandit algorithms. We can distinguish deterministic selection strategies, which assign a value to each move of a player and pick the one with the best value, and stochastic selection strategies, which perform a probabilistic action selection. Among the most popular and successful deterministic selection strategies we can find the ones based on UCB algorithms, like UCB1, from which UCT originated, and its refinement, UCB1-TUNED (Auer *et al.*, 2002). One of the simplest stochastic selection strategies is *ϵ-greedy*, which, for each player, selects the action with highest estimated payoff with probability $(1 - \epsilon)$ and a random action otherwise (Sutton and Barto, 1998). More advanced stochastic selection strategies are Exp3 (Exploration-Exploitation with Exponential weights) (Auer *et al.*, 1995) and Regret Matching (Hart and Mas-Colell, 2000). They both select an action for each player using a probability distribution on the actions available for the player. The first strategy computes this probability distribution using the cumulative payoff obtained by each action weighted by the probability of selecting the action, while the second uses for each action the cumulative regret for not having selected such action in previous iterations.

Expansion. In this phase one or more nodes are added to the tree. A common strategy is the one that adds to the tree one new node per simulation. More precisely, a new node is added for the first state encountered in the simulation that has no corresponding node in the tree yet. Other strategies might add more or less than one node per simulation or they might use different strategies to select which node to add. Some alternative expansion strategies are discussed in (Yajima *et al.*, 2011).

Play-out. During a play-out (also called *roll-out* in the literature), starting from the last node added to the tree the algorithm plays the game until a terminal state or a certain depth is reached. In each state the algorithm uses a *play-out strategy* to choose the joint action to simulate. One of the basic play-out strategies consists in performing a Monte-Carlo evaluation, thus for each player in each considered state an action is selected uniformly at random among the legal actions. However, it has been shown in the literature that more informed play-out strategies can increase the level of play (Gelly *et al.*, 2006; Finnsson and Björnsson, 2010; Tak *et al.*, 2012).

Backpropagation. At the end of the simulation, the payoffs obtained by all players are propagated back as a tuple through all the nodes traversed in the tree. The payoffs are used to update the information memorized in each node. Usually, each node s memorizes for each of the players the expected payoff and the number of visits of each legal action in the state. Depending on which selection and play-out strategies are being used, other information might be memorized in the nodes. In this thesis we backpropagate a tuple $\vec{q}$ with the scores that the players achieve in the last state reached by the simulation.

The execution of MCTS terminates when a predefined budget, for example a given number of simulations or a finite amount of time, expires. At this point, a

final action selection strategy chooses the best action in the root to be played in the real game. The final action selection technique can be implemented in various ways. For instance, it could choose the action with the highest number of visits or the one with the highest average score.

The general pseudocode for MCTS is given in Algorithm 2. This algorithm can be applied to games with any number of players, either deterministic or not, and with either simultaneous or sequential moves. It starts with the procedure MonteCarloTreeSearch(S, i), that keeps performing MCTS simulations from the root node until the search budget is over. After the last iteration, the action to play in the real game is selected with the *final action selection strategy* and returned.

Procedure PerformMctsSimulation(S) shows the recursive implementation of a single MCTS simulation, starting from the given tree node S. First of all, the procedure retrieves the game state s corresponding to the given node S using the method S.GetState( ). If this state is terminal, the simulation ends and the payoffs of the players are returned. Otherwise a joint action $\vec{a}$ is chosen by the selection strategy (line 9). The child node S' of S that is reached by playing such action is retrieved by the method S.GetChild($\vec{a}$). If the node S' is not null (i.e. it has been already created) then the selection phase continues with the recursion at line 12, otherwise the expansion phase starts (lines 14 to 16). During this phase the next state s' reached by performing the selected joint action $\vec{a}$ in s is retrieved. The method CreateNewNode(s') takes care of creating the new node S' by setting s' as the corresponding state and by initializing the statistics needed by the selection strategy. The node S', is added to the tree as the child of S. Subsequently, a play-out is started from the new state s' (line 17). Finally, line 18 makes sure that the statistics collected so far for the selected action $\vec{a}$ in S are updated with the backpropagated payoffs.

The play-out phase is implemented by the procedure PlayOut(s). Starting from the given state s this procedure uses the *play-out strategy* to select the joint action to simulate $\vec{a}$, and recursively calls itself on the next state reached by playing such action. When a terminal state is encountered, the tuple of payoffs is returned.

2.3.1 "Open-Loop" MCTS for Non-Deterministic Games

As mentioned in Section 2.1, given a state s and an action $\vec{a}$, the transition function $next(s, \vec{a})$ for non-deterministic or stochastic games returns each time a different state according to a predefined probability distribution over all the possible next states. To model this type of games the game tree is usually extended with the addition of *chance nodes*. A chance node is reached once an action has been selected in a state by the players, and it is used to model the choice of a possible next state that does not depend on the players. Once a chance node is reached, the environment selects one of the available next states according to the underlying probability distribution. Figure 2.3 shows an example of how a stochastic game is represented using chance nodes. Round nodes represent game states where the players can perform a joint action. Each action is associated to one of the exiting edges and leads to a diamond-shaped chance node. Each edge exiting a chance node is associated to the probability of reaching the corresponding next state (in

```
1: procedure MonteCarloTreeSearch(S, i)
   Require: The game model gm.
   Input: Current node S corresponding to the current game state s, player performing the search i ∈ R.
   Output: The action to play in the root state s for player i.
2:     while search budget available do
3:         q⃗ ← PerformMctsSimulation(S)
4:     return finalActionSelectionStrategy.Select(S, i)

5: procedure PerformMctsSimulation(S)
   Require: The game model gm.
   Input: Tree node S from where to start the simulation.
   Output: A tuple q⃗ = ⟨q_1, ..., q_r⟩ with the payoff obtained at the end of the simulation by each player.
6:     s ← S.GetState( )
7:     if gm.Terminal(s) then
8:         return gm.Payoff(s)
9:     a⃗ ← selectionStrategy.Select(S)
10:    S′ ← S.GetChild(a⃗ )
11:    if not S′ = null then
12:        q⃗ ←PerformMctsSimulation(S′)
13:    else
14:        s′ ← gm.GetNextState(s, a⃗ )
15:        S′ ← CreateNewNode(s′)                ▷ State s′ memorized in node S′
16:        S.AddChild(a⃗, S′)
17:        q⃗ ←PlayOut(s′)
18:    S.Update(a⃗, q⃗ )
19:    return q⃗

20: procedure PlayOut(s)
   Require: The game model gm.
   Input: The game state s where to start the play-out.
   Output: A tuple q⃗ = ⟨q_1, ..., q_r⟩ with the payoff obtained at the end of the play-out by each player.
21:    if gm.Terminal(s) then
22:        return gm.Payoff(s)
23:    a⃗ ← playOutSrtategy.Select(s)
24:    s′ ← gm.GetNextState(s, a⃗ )
25:    return PlayOut(s′)
```

Algorithm 2: Pseudocode for Monte-Carlo Tree Search.

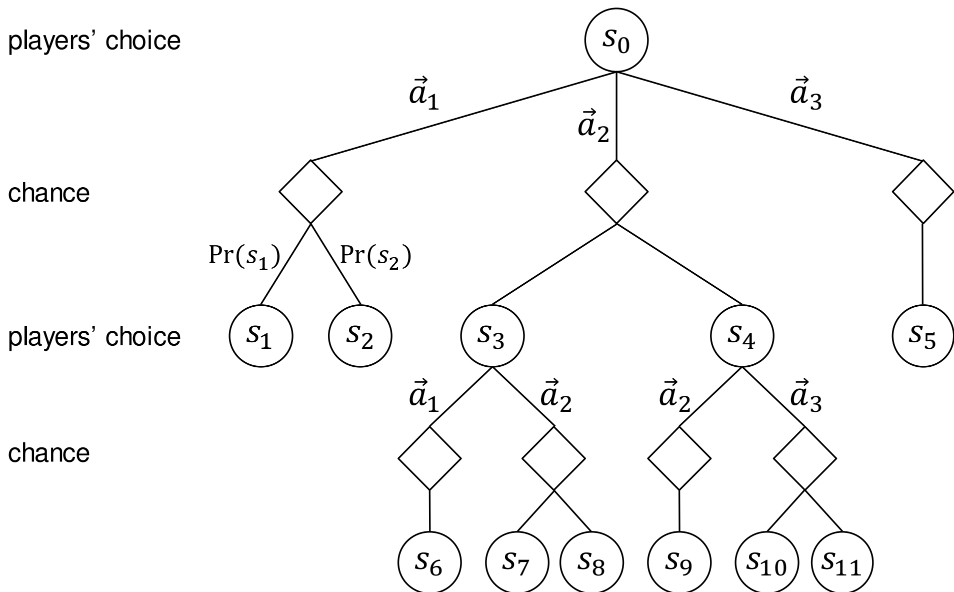


Figure 2.3: Representation of a game tree with chance nodes.

the figure probabilities are explicitly reported only for the chance node reached performing action a_1 in the root).

Such a model is easy to construct for stochastic games, where the players know the probability distribution of the next states. For non-deterministic games it is still applicable, however, the probability distribution is unknown and it is thus infeasible to create the exact model. The probability distribution over the next states could be estimated. However, a large number of samples for each action would be required in order to find all possible successor states and compute an accurate estimate. In addition, adding a node for each possible successor state of each action makes the three grow exponentially.

To avoid these problems, the "open-loop" representation of the game tree has been proposed as an alternative to the model with chance nodes for non-deterministic domains (Perez-Liebana *et al.*, 2015). This tree representation has seen successful applications particularly in the GVG-AI competition, where it is adopted by many of the participating agents to tackle non-deterministic video games (Perez-Liebana *et al.*, 2016; Gaina *et al.*, 2018; Perez-Liebana *et al.*, 2018). With an "open-loop" representation, each tree node except the root can correspond to multiple game states. More precisely, a node S in the game tree of a non-deterministic game corresponds to all the states that can be reached by performing the sequence of actions on the path from the root node to S. The root node of the game tree is the only node that is always corresponding to a single state, the current state of the game. Moreover, each edge exiting a tree node corresponds to one of the actions that are legal in any of the states that correspond to the node. Figure 2.4 shows the "open-loop" representation of the same game shown in Figure 2.3. Note that

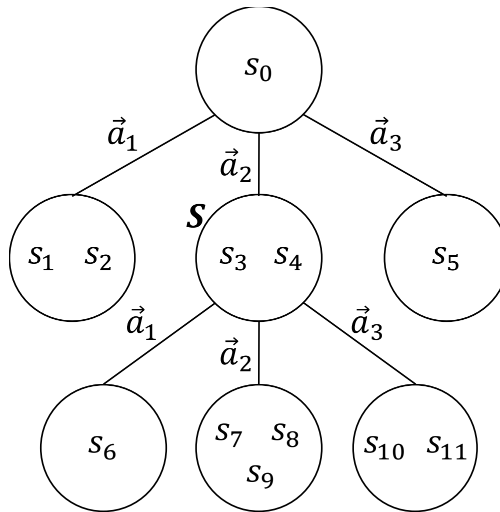


Figure 2.4: "Open-loop" representation of a game tree for a non-deterministic game.

this representation reduces the number of nodes in the tree, but at the same time it does not model some information about the game. For example, in node S we do not know that by playing action $\vec{a}_2$ we can reach states s_7 and s_8 only from state s_3, and state s_9 only from state s_4.

To exploit an "open-loop" representation the MCTS algorithm, Algorithm 2 presented in Section 2.3, has to be adapted. This algorithm used a "closed-loop" representation: each time a new node is added to the tree during a simulation, the state generated by the transition function is memorized at it (lines 14 and 15), and it is reused in subsequent simulations without being generated again (line 6). For a non-deterministic game this means that each node will be associated to only one of all the possible next states generated by sampling the corresponding distribution. When generating and reusing only one state we do not know how representative it is for the set of all possible states, and this might be detrimental for the performance of the search.

Algorithm 3 gives the pseudocode of "open-loop" MCTS, which does not memorize the states at the nodes, but uses the game model to generate the states every time it is required. The main difference with "closed-loop" MCTS in Algorithm 2 is that every time a node in the tree is visited the corresponding state is generated using the game model (lines 9 and 10), and whenever a new node is created the state is used to initialize the statistics but is not memorized (line 14). Not memorizing the state at the node also means that both the procedure MonteCarloTreeSearch(S, s, i) and the procedure PerformMctsSimulation(S, s) have to use the state s corresponding to the visited node S as a parameter.

The "open-loop" approach has the advantage of using less memory than the "closed-loop" approach, because states are not memorized at the nodes. At the same time, however, it requires the use of the game model to compute a state every time a node is visited in the tree during a simulation, while "closed-loop" MCTS only

```
1: procedure OpenLoopMonteCarloTreeSearch(S, s, i)
   Require: The game model gm.
   Input: Current node S, the corresponding current game state s, player per-
   forming the search i ∈ R.
   Output: The action to play in the root state s for player i.
2:     while search budget available do
3:         q⃗ ← PerformMctsSimulation(S, s)
4:     return finalActionSelectionStrategy.Select(S, i)

5: procedure PerformMctsSimulation(S, s)
   Require: The game model gm.
   Input: Tree node S from where to start the simulation and (one of) the corre-
   sponding game state(s) s.
   Output: A tuple q⃗ = ⟨q_1, . . . , q_r⟩ with the payoff obtained at the end of the
   simulation by each player.
6:     if gm.Terminal(s) then
7:         return gm.Payoff(s)
8:     a⃗ ← selectionStrategy.Select(S)
9:     S′ ← S.GetChild(a⃗)
10:    s′ ← gm.GetNextState(s, a⃗)
11:    if not S′ = null then
12:        q⃗ ←PerformMctsSimulation(S′, s′)
13:    else
14:        S′ ← CreateNewNode(s′)          ▷ State s′ not memorized in node S′
15:        S.AddChild(a⃗, S′)
16:        q⃗ ←PlayOut(s′)
17:    S.Update(a⃗, q⃗)
18:    return q⃗

19: procedure PlayOut(s)
   Require: The game model gm.
   Input: The game state s where to start the play-out.
   Output: A tuple q⃗ = ⟨q_1, . . . , q_r⟩ with the payoff obtained at the end of the
   play-out by each player.
20:    if gm.Terminal(s) then
21:        return gm.Payoff(s)
22:    a⃗ ← playOutSrtategy.Select(s)
23:    s′ ← gm.GetNextState(s, a⃗)
24:    return PlayOut(s′)
```

Algorithm 3: Pseudocode for “open-loop” Monte-Carlo Tree Search.

needs the game model to compute one state in the tree for each simulation, i.e. the one corresponding to the newly added node. If using the game model has a high computational cost, "open-loop" MCTS might be at a disadvantage compared to "closed-loop" MCTS. When deciding which of the two approaches to use these aspects should be taken into account.

2.4 The UCT Selection Strategy

A popular MCTS selection strategy is UCT (Kocsis and Szepesvári, 2006), which is based on the UCB1 algorithm discussed in Subsection 2.2.2. Given a game state s and the set $\mathcal{A}_{(s,i)}$ of all legal actions of player i in s, UCT assigns the value $UCT(s, a_i)$ to each action a_i of the player, and selects the action a_i^* according to Formula 2.9.

$$\begin{aligned} a_i^* &= \operatorname*{argmax}_{a_i \in \mathcal{A}_{(s,i)}} \left\{ UCT(s, a_i) \right\} \\ UCT(s, a_i) &= \bar{q}_{(s,a_i)} + C \times \sqrt{\frac{\ln n_s}{n_{(s,a_i)}}} \end{aligned} \tag{2.3}$$

Here, n_s is the number of times state s has been visited during the search and $n_{(s,a_i)}$ is the number of times action a_i has been selected for player i whenever node s was visited. The term $\bar{q}_{(s,a_i)}$ is the average payoff obtained for all the simulations in which action a_i was selected for player i in state s, and like for MCS it can be computed as $\frac{qSum_{(s,a_i)}}{n_{(s,a_i)}}$, where $qSum_{(s,a_i)}$ is the sum of all payoffs obtained so far by action a_i when selected in state s. C is a constant that is used to control the balance between the exploitation promoted by the first term of the formula, and the exploration promoted by the second term. As for UCB1, in UCT the values of the payoffs are expected to be in the interval $[0, 1]$. When this holds true and given sufficient time and memory, Kocsis and Szepesvári (2006) proved that in a two-player, sequential move game the probability of selecting a sub-optimal action in the root of the MCTS tree converges to zero. Therefore, the MCTS tree converges to the optimal value of the minimax tree.

Whenever the UCT selection strategy or one of its variants are used, MCTS has to memorize in the nodes the statistics that are necessary to compute the expected payoff of an action $\bar{q}_{(s,a_i)}$ that is used in Formula 2.9. These statistics can be memorized at the action level, i.e. for each legal action in each node, or at the node level, i.e. for each node in the tree. For example, when considering the tree of a sequential move game, memorizing statistics at the action level would mean memorizing in each node the values $qSum_{(s,a_i)}$ and $n_{(s,a_i)}$ for each action a_i that is legal for the player i that is on turn in the node. These values can then be used to compute $\bar{q}_{(s,a_i)} = \frac{qSum_{(s,a_i)}}{n_{(s,a_i)}}$ to be used in the UCT formula. Memorizing statistics at the node level, instead, would mean memorizing in each node the values $qSum_{(s,i)}$ and n_s for the corresponding state s and then computing the expected payoff of an action a_i as the expected payoff of the state s' reached by performing such action, $\bar{q}_{(s,a_i)} = \frac{qSum_{(s',i)}}{n_{(s')}}$. Here, $qSum_{(s,i)}$ is the sum of all the payoffs obtained by player

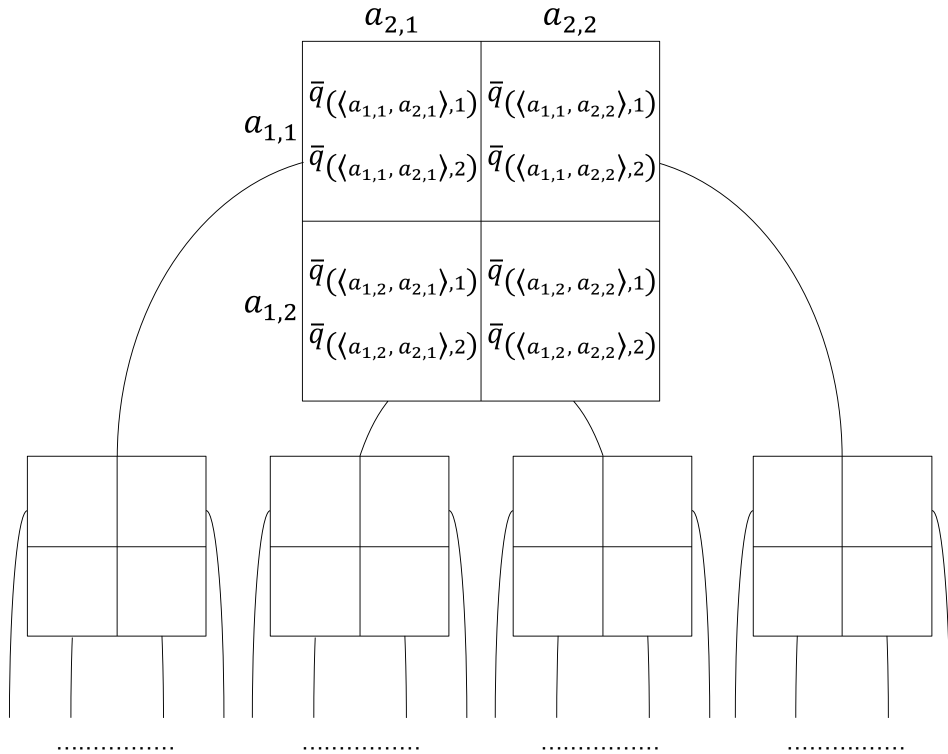


Figure 2.5: Representation of a game tree for a simultaneous move game with two players.

i in the simulations in which state s was visited, and n_s is the number of times state s was visited. Both implementations are equivalent for sequential move games, but in general one might have (dis)advantages over the other. In the remainder of this thesis it is assumed that statistics are memorized at the action level. These statistics are computed using as payoffs the scores obtained by the players at the end of each simulation. Because there is no guarantee that the scores will be in the interval $[0, 1]$, whenever used in the UCT formula they are first re-scaled for this interval.

2.4.1 Simultaneous Move UCT

The UCT selection strategy used by MCTS was originally designed for sequential move, turn-taking games. In such games the standard way of applying UCT consists in using it in each state to select the action for the player that has to move in the corresponding turn. For simultaneous move games, however, multiple players have to move in the same turn, and adapting UCT to such a situation is not trivial.

Figure 2.5 shows an example of how the search tree for a simultaneous move game can be represented. The considered game has two players, 1 and 2, each of which has two possible actions in each state. In the root, Player 1 can perform

actions $a_{1,1}$ and $a_{1,2}$, while Player 2 can perform actions $a_{2,1}$ and $a_{2,2}$ (given an action a, the first subscript index indicates the player that performs the action and the second subscript index is used to distinguish among the actions of the player). In each node, to represent all possible combinations of actions a matrix A can be used, where the rows corresponds to the actions of Player 1 and the columns to the actions of Player 2. Each entry $A_{j,k}$ of the matrix memorizes for each player $i \in \{1, 2\}$ the (estimated) payoff $\bar{q}_{(\langle a_{1,j}, a_{2,k}\rangle, i)}$ computed by selecting the given joint action in the node. Moreover, for each joint action there is an edge pointing to the next node in the tree, corresponding to the state reached by performing such joint action from the current node. The presented representation can be used for simultaneous move games with any number of players. For a game with r players moving simultaneously each node of the tree will correspond to an r-dimensional matrix.

It is easy to see the space complexity of simultaneous move games. First of all, the next game state depends on the actions of more than one player, therefore when adapting the UCT selection strategy to such games we have to decide if to consider this interdependency and how. Secondly, the number of joint actions increases exponentially with the number of players. Usually, to apply UCT each player has to memorize for each action a the sum of payoffs obtained so far by the action, $qSum_a$, and the visits count, n_a, so that it can compute the average payoff $\bar{q}_a$. If in a simultaneous move game these pairs of statistics for each joint action have to be memorized for each player, then the required space is also increasing exponentially. For example, if we consider a simultaneous move game with r players, each with x legal actions per node, we have x^r joint actions, with a total of rx^r pairs of statistics. This aspect might also be taken into account when designing a simultaneous move UCT selection strategy.

Two different approaches have been proposed to adapt UCT to simultaneous move games with two or more players (Tak, Lanctot, and Winands, 2014a): *Sequential UCT* (SUCT) and *Decoupled UCT* (DUCT). These two approaches differ in how they address the previously mentioned aspects. Below, SUCT is discussed first, and subsequently DUCT.

Sequential UCT

SUCT transforms the simultaneous move game into a sequential move game by serializing the game tree, therefore the selection of a joint action for a game state becomes a sequence of selections of single actions, one for each player. The standard UCT selection can thus be applied to select each of the single actions.

To better clarify how SUCT works with an example, Figure 2.6 shows part of the search tree built by serializing the game tree introduced in Figure 2.5. Each tree node (in gray) corresponds to two levels of the serialized game tree, one for each player. The figure also shows which statistics are memorized when using SUCT for the root state of the game. For Player 1, statistics are kept for each action, and an action is selected by applying standard UCT on these statistics. Whenever in a simulation one of the actions is chosen, the corresponding $qSum$ will be updated with the payoff obtained by Player 1 at the end of the simulation and the corresponding n will be incremented. For Player 2, a separate instance of statistics for the actions is kept

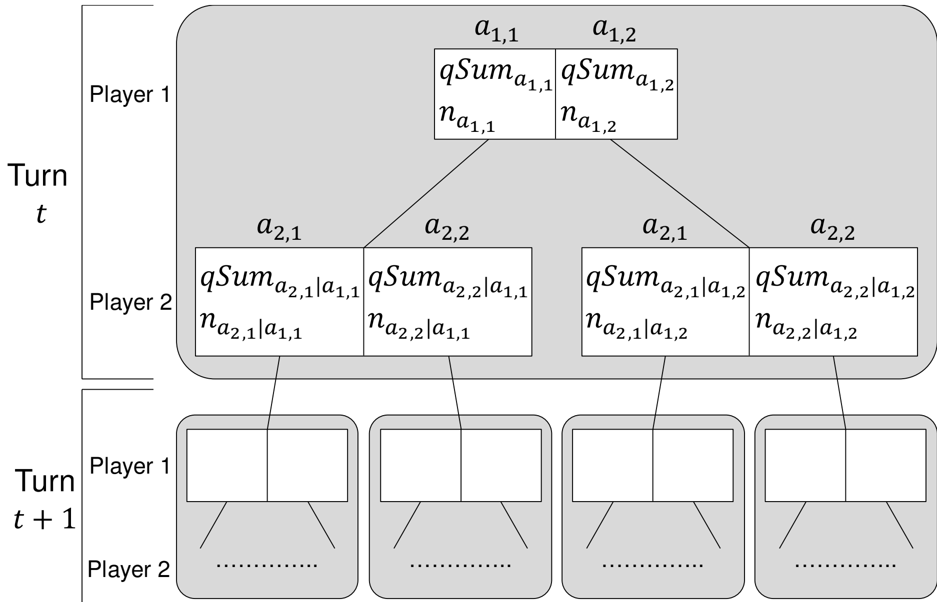


Figure 2.6: Sequential action selection in a turn of a two-player simultaneous move game.

for each of the actions of Player 1. The subscript $a_{2,k}|a_{1,j}$ for a statistic of Player 2 indicates that the statistic has been collected for action $a_{2,k}$ of the player, given that action $a_{1,j}$ has been selected for Player 1. An action for Player 2 is selected using standard UCT on the instance of statistics reached by the edge corresponding to the action selected for Player 1. At the end of a simulation, the statistics of the action are updated only for this instance. The payoff obtained by Player 2 at the end of the simulation will be used to update the corresponding $qSum$, and the corresponding n will be incremented. Only when the complete joint action is selected the next game state is computed and a new node is visited (or generated and added to the search tree if not yet present).

The same structure can be easily adapted to simultaneous move games with r players, by adding for each tree node a level of statistics for each player. Collecting statistics with this structure reduces the space usage with respect to the matrix structure discussed previously. When using SUCT, if we consider a simultaneous move game with r players, each with x legal actions per node, we have a total of $\sum_{i=1}^{r} x^i$ pairs of statistics, which is less than the rx^r pairs of statistics memorized by the matrix.

To be noted when using SUCT is that the order in which the players are considered influences the action selection, because different statistics will be collected depending on the order of the players. Bošanský *et al.* (2016) studied the effect of game serialization on the value of a two-player, zero-sum, perfect-information, simultaneous move game. They show that for the player that plays second the value computed applying minimax to the serialized game is an upper bound on the value that would be obtained by considering the game as a matrix. Letting the opponent

	$a_{2,1}$	$a_{2,2}$	
$a_{1,1}$	$qSum_{\langle a_{1,1}, a_{2,1}\rangle}$ $n_{\langle a_{1,1}, a_{2,1}\rangle}$	$qSum_{\langle a_{1,1}, a_{2,2}\rangle}$ $n_{\langle a_{1,1}, a_{2,2}\rangle}$	$qSum_{a_{1,1}} = qSum_{\langle a_{1,1}, a_{2,1}\rangle} + qSum_{\langle a_{1,1}, a_{2,2}\rangle}$ $n_{a_{1,1}} = n_{\langle a_{1,1}, a_{2,1}\rangle} + n_{\langle a_{1,1}, a_{2,2}\rangle}$
$a_{1,2}$	$qSum_{\langle a_{1,2}, a_{2,1}\rangle}$ $n_{\langle a_{1,2}, a_{2,1}\rangle}$	$qSum_{\langle a_{1,2}, a_{2,2}\rangle}$ $n_{\langle a_{1,2}, a_{2,2}\rangle}$	$qSum_{a_{1,2}} = qSum_{\langle a_{1,2}, a_{2,1}\rangle} + qSum_{\langle a_{1,2}, a_{2,2}\rangle}$ $n_{a_{1,2}} = n_{\langle a_{1,2}, a_{2,1}\rangle} + n_{\langle a_{1,2}, a_{2,2}\rangle}$

Figure 2.7: Computation of cumulative statistics for Player 1 in a state of a simultaneous move game with two players.

play second will then make us overestimate her game value and make the player performing the search be more cautious in the action selection. Usually, when using SUCT, regardless of the number of opponents, the player performing the search is the one that plays first. In this way, the opponents are simulated with the advantage of knowing which action was selected for the initial player when it is time to select their own actions. As a consequence the player performing the search learns to play the actions that have less risk of being penalized by the opponents, thus assuming a defensive behavior.

In general, SUCT is not guaranteed to converge to an optimal solution, nevertheless it has been tested in multiple domains. Tron was the first game in which it was applied (Samothrakis, Robles, and Lucas, 2010; Den Teuling and Winands, 2012). In Tron SUCT proved quite successful. In the work of Lanctot *et al.* (2013) different variants of SUCT performed in general better than the stochastic approaches Exp3 and Regret Matching. Tak *et al.* (2014a) also showed that SUCT has a good performance in Tron. Their work also tested SUCT on other games to verify how it performs in a GGP setup. It was shown that for almost all games SUCT performs better than Exp3 and Regret Matching.

Decoupled UCT

DUCT is so called because, when selecting a joint action in a tree node, it decouples the action statistics for each player and applies UCT to these statistics to select the action of each player independently. Figure 2.7 gives an example of which statistics are considered for a player when the DUCT selection strategy is applied. This example refers to the same simultaneous move game shown in Figure 2.5, but only shows statistics from the point of view of Player 1 (an example for Player 2 would be similar). The matrix in the figure refers to the root state of the simultaneous move game tree presented in Figure 2.5. Given this matrix, the statistics that would be considered for each action of Player 1 are shown on the right of the corresponding row of the matrix. For each of the actions a_1^j of Player 1, DUCT considers the cumulative

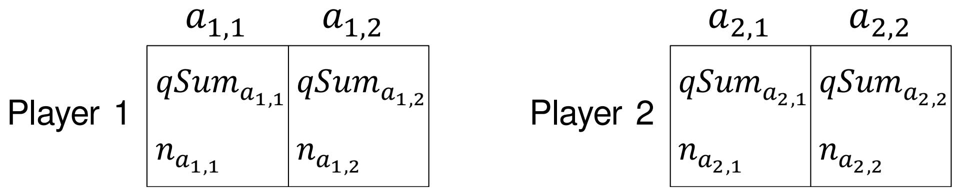


Figure 2.8: Statistics memorized by each player in a state, when DUCT is applied on a simultaneous move game with two players.

sum of payoffs and cumulative number of visits, $qSum_{a_1^j}$ and $n_{a_1^j}$ over all the other player's actions. These cumulative statistics are used to compute the estimated payoff $\bar{q}_{a_1^j}$ required to compute the value of action a_1^j with the UCT formula.

The fact that DUCT uses cumulative statistics means that each player only has to memorize such statistics for her own actions in each node, thus saving space with respect to the memorization of the whole matrix of statistics. Considering the same example introduced earlier, Figure 2.8 shows the statistics that each of the two players memorizes in the root node. The reduced space complexity can be seen as an advantage of DUCT. In a game with r players, where each player has x legal actions per state, the total number of statistics to be memorized is rx, which is much less than the previously mentioned rx^r statistics for the whole matrix of joint actions.

Despite DUCT has been proved to not converge in general to an optimal strategy (Shafiei, Sturtevant, and Schaeffer, 2009), it has been successfully applied in various domains. Different variants of DUCT have been successful in Tron, where they were shown to outperform stochastic selection strategies (Perick *et al.*, 2012). MCTS with the DUCT selection strategy was first applied to GGP by CadiaPlayer (Björnsson and Finnsson, 2009), an agent developed for the competition related to the Stanford GGP project (Genesereth *et al.*, 2005), where each abstract game is modeled as a simultaneous move game. The CadiaPlayer agent has always placed among the top players of the competition between 2007 and 2012. Tak *et al.* (2014a) confirmed this success in GGP by showing that DUCT is the best on most of the nine tested simultaneous move games when compared to SUCT, Exp3 and Regret Matching. DUCT has also been analyzed by Bošanský *et al.* (2016), which showed that its performance can be further improved by using a random tie-breaking rule when selecting among multiple actions of a player that have the same UCT value. More precisely, for each player an action is selected randomly among the ones that have a UCT value within a predefined small offset from the highest UCT value. Experiments show that this agent converges to a better approximation of the optimal strategy with respect to a DUCT agent that uses a deterministic tie-breaking rule (e.g. always selecting the first or the last among the actions with the same value). Given its overall better performance over SUCT, enhancements for the UCT strategy are presented in the rest of this thesis assuming the DUCT implementation.

2.5 MCTS Enhancements

This section discusses MCTS enhancements that are relevant for this thesis. First, Subsections 2.5.1 and 2.5.2 describe various enhancements for the selection and the play-out phase of MCTS, respectively. Subsequently, Subsection 2.5.3 introduces the use of transposition tables to memorize tree nodes for MCTS, and finally, Subsection 2.5.4 explains how the search tree built during a game turn can be reused in subsequent turns.

2.5.1 Selection Strategy Enhancements

Various enhancements for the MCTS selection strategy have been proposed in the literature (Browne *et al.*, 2012). Usually, a selection strategy assigns a value to each of the available actions in a state and selects an action according to these values. Enhancing the selection strategy by adding knowledge that modifies such values and guides the search towards certain actions has been shown to be beneficial in many domains. Below, the selection strategy enhancements that are used by the MCTS agents in this thesis are discussed: *First Play Urgency*, *Rapid Action Value Estimation* and *Progressive History*. A characteristic that all these enhancements have in common is that they are all domain-independent, which makes them suitable for domains like GGP.

First Play Urgency

In its standard implementation, the UCT selection strategy randomly visits all actions in a state at least once before using Formula 2.9 to select the best action. This is inefficient if there is a high number of actions in a state and not many simulations are available. Moreover, if an action is always returning a win or a high payoff whenever evaluated, it seems more reasonable to keep selecting such action rather than other unexplored ones.

Wang and Gelly (2007) propose to assign to unexplored actions a predefined value, called *First-Play Urgency* (FPU), that determines the urgency for these actions to be explored. When an action has to be selected in a state, the UCT value is computed for already explored actions, while the FPU value is assigned to unexplored ones. Subsequently, the action with the highest value is selected. In this way, it is possible to control how urgently unexplored actions should be selected. A value of $FPU = \infty$ corresponds to the standard strategy, i.e. all actions are explored once before considering their UCT value. On the contrary, a low value of FPU means that the exploitation of actions with a high expected payoff will be preferred to exploration of unvisited actions. The work of Wang and Gelly (2007) shows that using an FPU value that favors early exploitation has positive results in the game of Go.

The idea of FPU has also been successfully applied to GGP by Finnsson (2012a) using the enhancement that they call *Unexplored Action Urgency*. Like FPU, it assigns to the unvisited actions a predefined value that corresponds to the UCT value that would be computed for an action visited once and that resulted in a draw. In addition, a discount is added to the urgency value, which decreases the urgency of

exploring unvisited actions as the number of explored actions increases. The Unexplored Action Urgency enhancement is shown to improve the agent's performance in GGP when paired with an informed play-out strategy.

Rapid Action Value Estimation

The *Rapid Action Value Estimation* (RAVE) strategy (Gelly and Silver, 2007) is a domain-independent technique that has been proposed in order to speed up the learning process inside the MCTS tree. When there are only a few samples available to compute the UCT value of a player's action in a node, RAVE adds to the computation the action statistics collected for the whole subtree of the node. This results in a faster decrease of the variance of the estimated payoff of the actions. More details about the RAVE technique are given in Chapter 5.

RAVE has been successfully applied in Go (Gelly and Silver, 2007; Gelly and Silver, 2011), Hex (Cazenave and Abdallah, 2010), Havannah (Teytaud and Teytaud, 2010; Rimmel, Teytaud, and Teytaud, 2011) and GGP (Finnsson and Björnsson, 2010). Moreover, multiple variations of RAVE have been proposed. Lorentz (2011) presented *Killer RAVE*, a modification of RAVE that uses the subtree statistics to improve the UCT value only for the most important actions. Killer RAVE was shown to significantly improve over RAVE in Havannah. *RAVE-max* and its stochastic variant *δ-RAVE-max* were proposed by Tom and Müller (2011) to make RAVE action estimates more robust. These variant worked well for the game Sum of Switches, but not for Go. Another variant, *poolRAVE*, was proposed by Hoock *et al.* (2010). This variant keeps a pool of k-best moves according to the RAVE statistics. During selection for a player, a move from the pool is played with a certain probability, otherwise the default selection strategy is used. In the games of Havannah and Go poolRAVE was shown to be successful.

Progressive History

Progressive History (PH) has first been proposed by Nijssen and Winands (2011), which successfully applied it to multi-player games. Later, it has also been applied to GVGP showing good results in some of the tested video games (Soemers *et al.*, 2016). Similarly to RAVE, PH combines the UCT value of a player's action together with another quantity that is computed using knowledge acquired during the search. More precisely, PH biases action selection in a state towards actions that have performed well in other states, assuming that actions that are good in a state might be good in similar states as well. To compute the bias, PH keeps a global table where it memorizes for each action a_i of each player i the expected payoff $\bar{q}_{a_i}$ of all the simulations performed so far in which a_i was played at any point in the game. Formula 2.4 shows how the UCT value of an action a_i in a state s is computed with the PH enhancement from the point of view of player i. For each player, this value is computed independently.

$$UCT_{PH}(s, a_i) = \bar{q}_{(s,a_i)} + C \times \sqrt{\frac{\ln n_s}{n_{(s,a_i)}}} + \bar{q}_{a_i} \times \frac{W}{(1 - \bar{q}_{(s,a_i)})n_{(s,a_i)} + 1} \tag{2.4}$$

The first two terms are the same as in the UCT formula presented in Section 2.2. The last term represents the bias towards generally good actions. Here, $\bar{q}_{a_i}$ is the previously mentioned global average payoff of action a_i, and W is a constant that determines the influence of the PH bias on the selection. The denominator $(1 - \bar{q}_{(s,a_i)})n_{(s,a_i)} + 1$ is needed to decay the influence of the bias over time, as the actions get more visits and the UCT estimates become more reliable. Multiplying the number of visits $n_{(s,a_i)}$ by the inverse payoff of action a_i in s, $(1 - \bar{q}_{(s,a_i)})$, guarantees that actions that perform well in a state are biased longer than actions that perform poorly. Summing 1 at the denominator is used to avoid dividing by 0 if $\bar{q}_{(s,a_i)} = 1$.

The statistics that PH collects for each action are similar to the ones collected by RAVE, with the difference that RAVE collects such statistics for each node instead of globally. This means that PH has as an advantage over RAVE because it is using less memory. Moreover, the memory required to store statistics for PH is constant with respect to the size of the tree, while for RAVE it increases linearly as the tree grows. However, a possible disadvantage of PH with respect to RAVE is that the global information collected by PH might be less accurate for a certain state than the information collected by RAVE for the same state.

2.5.2 Play-out Strategy Enhancements

One of the basic play-out strategies for MCTS is the one that selects random actions in each state. However, random play-outs are not very realistic, because the opponent(s) are likely making decisions using a more rational strategy. Using some knowledge about the game to guide the play-out can increase the performance of an MCTS agent. Below we discuss the play-out strategy enhancements that are used by the MCTS agents in this thesis: *Move Average Sampling Technique* and *N-Grams Selection Technique*. Similarly to the selection strategy enhancements presented in Subsection 2.5.1, these play-out strategy enhancements are suitable for domains like GGP because they are all domain-independent.

Move Average Sampling Technique

The Move Average Sampling Technique (MAST) has been devised by Finnsson and Björnsson (2008), which successfully integrated it in CadiaPlayer, an MCTS-based agent developed for the Stanford GGP competition. The main idea behind MAST is the same as the PH enhancement for the selection strategy, i.e. an action that is good in one state is likely to be good also in other states. Like PH, during the search MAST memorizes for each action a_i of each player i the expected payoff $\bar{q}_{a_i}$ based on the results of all the simulations in which action a_i was played so far. The difference is that MAST uses this information to guide the search outside of the tree, during the play-out. In this thesis, when using MAST, duplicate actions in a simulation are not detected, thus the MAST strategy will update their expected value each time they appear in the simulation.

The original implementation of MAST guides the play-out by selecting actions in a state according to the probabilities computed using the Gibbs measure. Formula

2.5 shows how the probability $Prob(a_i, s)$ of selecting action a_i of player i in state s is computed with the Gibbs measure.

$$Prob(a_i, s) = \frac{e^{(\bar{q}_{a_i}/\tau)}}{\sum_{j=1}^{|\mathcal{A}_{(s,i)}|} e^{(\bar{q}_{a_{i,j}}/\tau)}} \tag{2.5}$$

Here, $\mathcal{A}_{(s,i)}$ is the set of all the actions that are legal for player i in the current state s, and τ is a parameter that controls the shape of the distribution. Higher values of τ make the distribution more uniform making it favor more exploration of actions, while lower values stretch it, making it favor more exploitation. With the Gibbs measure, actions with a high global average, and thus more likely to be good in general, are also more likely to be selected during the play-out.

Later research on MAST has shown that the use of an ϵ-greedy strategy to select actions gives a significantly better performance than the use of the Gibbs measure in most of the tested games (Tak *et al.*, 2012; Powley, Whitehouse, and Cowling, 2013). The ϵ-greedy strategy chooses the action with highest expected payoff $\bar{q}_{a_i}$ with probability $(1-\epsilon_{\text{MAST}})$ and a random action with probability ϵ_{MAST}. Tak *et al.* (2012) motivate the success of the ϵ-greedy strategy by considering that it always guarantees a fixed probability (i.e. $1 - \epsilon_{\text{MAST}}$) of selecting the action with highest expected payoff. With the Gibbs measure, on the contrary, this probability varies depending on the expected payoff of other actions, and the action with the highest payoff might get swamped by other actions.

A *first play urgency* parameter, fpu_{MAST} can be specified for the MAST strategy, both when using the Gibbs measure and when using an ϵ-greedy strategy. This parameter specifies a default value that is assigned to actions that have not been visited yet during the search, and thus no expected payoff is memorized for them in the statistics collected by MAST. A high value of fpu_{MAST} motivates exploration of not yet visited actions.

Further research by Tak, Winands, and Björnsson (2014b) has also shown that decaying the global statistics used by MAST is beneficial. MAST keeps the information collected during the search for a game step and reuses it in subsequent steps, so that at the start of a new search it is already known which the best actions are in general. However, as the game progresses, old statistics might not be as reliable as they were before. For example, statistics collected in an early state of the game might be based on parts of the tree that are not reachable anymore at a later stage of the game. Decaying the statistics over time helps addressing this issue. Three different decay methods have been proposed, *move decay*, *batch decay* and *simulation decay*, which decay MAST statistics after each action, after a batch of simulations and after each simulation, respectively. Move and batch decay discount the MAST statistics of all actions, while simulation decay discounts only the statistics of the actions visited in the simulation after which it has been applied. Formula 2.6 shows how the decay with a factor of $\omega \in [0, 1]$ updates the global statistics of an action a_i of player i.

$$\begin{aligned} qSum_{a_i} &\leftarrow \omega \times qSum_{a_i} \\ n_{a_i} &\leftarrow \omega \times n_{a_i} \end{aligned} \tag{2.6}$$

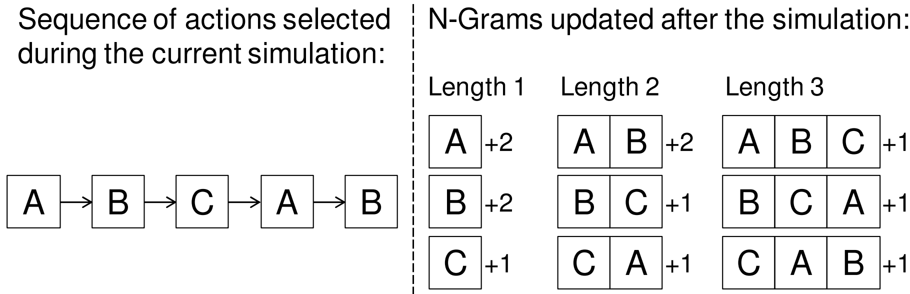


Figure 2.9: Extraction and update of N-Grams after an MCTS simulation.

The statistics $qSum_{a_i}$ and n_{a_i} have the same meaning as in Formula 2.1 in Subsection 2.2.1 and are used to compute the average payoff of action a_i. Among the tree proposed decay strategies, Tak *et al.* (2014b) show that move and simulation decay appear to be comparable in performance on the tested set of games. Both of them seem better than batch decay. However, there are a few games for which move decay shows a better performance than simulation decay and vice versa.

N-Grams Selection Technique

The *N-Grams Selection Technique* (NST) (Tak *et al.*, 2012) is a play-out strategy similar to MAST. Like MAST, NST biases the play-outs towards actions that have performed overall well so far. The difference is that NST does not only exploit information about single actions, but also about sequences (*N-Grams*) of actions. This makes NST more suitable for games where a good action might depend on the actions of the opponents, or where there are certain actions that perform generally well when applied in sequence. NST has been shown to be successful as domain-independent strategy when applied to GGP both for board games (Tak *et al.*, 2012) as well as for video games (Soemers *et al.*, 2016). Moreover, Powley *et al.* (2013) and Powley, Cowling, and Whitehouse (2014) used the same idea of NST (i.e. using statistics of N-Grams of actions to guide the play-outs) to implemented their strategy, the N-Gram-Average Sampling Technique (NAST). This technique has been shown to be successful also for imperfect-information games.

In this thesis, NST is applied only to single-player games, therefore the examples below are given for this category of games. However, NST is applicable to two- and multi-player games as well, with either sequential or simultaneous move. For a better explanation of how NST is applied to these categories of games we refer to Tak *et al.* (2012). When implemented in MCTS, after each simulation NST extracts from the simulated path all N-Grams of actions up to a certain length L and uses the payoff obtained by the simulation to update their expected payoff. Like for MAST, duplicates in the same simulation are not detected, therefore the payoff of an N-Gram will be updated for each of its occurrences. Figure 2.9 shows which N-Grams NST will extract from the given simulation for a single-player game with maximum N-Gram length $L = 3$. Next to each N-Gram the number of updates is shown.

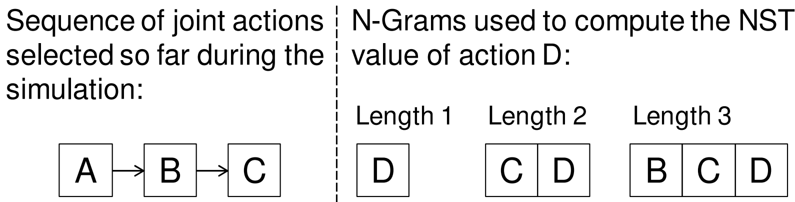


Figure 2.10: Use of N-Grams to compute the NST value of an action.

With NST, the selection of an action in a state during the play-out works as follows. Like for MAST, with probability ϵ_{NST} an action is chosen randomly. Otherwise, with probability $1 - \epsilon_{\text{NST}}$ the one with the highest NST value is chosen. In a single-player game, when considering N-Grams up to length L, the NST value of an action a is computed averaging the payoff of all the N-Grams of length $l \in [1, L]$ that end with action a preceded by the last $l - 1$ actions selected so far in the simulation. In this computation, N-Grams with length greater than 1 are taken into account only if they have been visited at least N times. For a single-player game with maximum N-Gram length $L = 3$, Figure 2.10 shows which N-Grams are considered to compute the NST value of the given action.

Like for MAST, also for NST a *first play urgency* parameter, fpu_{NST} can be specified. Exactly like fpu_{MAST}, this parameter specifies a default value that is assigned to actions that have never been encountered before during the search. Note that fpu_{NST} is not applied to unvisited N-Grams with length greater than 1, because such N-Grams are taken into account only if they have been visited at least N times. For NST it is also beneficial to keep the collected N-Grams statistics, so that they can be reused to guide the search in subsequent game steps. Once again, Tak *et al.* (2014b) showed that decaying such statistics over time has an overall positive effect on the search.

2.5.3 Transposition Tables

Oftentimes a search algorithm might visit identical states in different parts of the game tree. Such states are called *transpositions* and are reached through different sequences of actions, therefore in a search tree they will be associated to distinct nodes. This also means that relevant information about the same state is kept separately in each of these nodes. For games where transpositions are present, *transposition tables* (Greenblatt, Eastlake III, and Crocker, 1967) enable us to represent the search tree as a *graph*, where each state corresponds to a single node. In this way, the information memorized for a state in a node can be reused for each of its transposition, reducing search effort.

When using transposition tables, however, there are multiple aspects that have to be taken into consideration. First of all, using a graph representation of the search space might give rise to the Graph-History Interaction problem (GHI). This problem is caused by the fact that the information related to a node might depend on the history of actions that led to it. Moreover, transposition tables usually require

a large amount of space to store all the visited nodes, therefore an appropriate representation is necessary. Finally, when using transposition tables in MCTS it is necessary to adapt the UCT selection strategy, because there are multiple ways to update action statistics and compute UCT values. These issues are discussed below.

The Graph-History Interaction Problem

The graph represented by transposition tables could be of two types: a Directed Acyclic Graph (DAG), if a state cannot be visited more than once in the same line of play, and a Directed Cyclic Graph (DCG) otherwise. An example of game that can be represented with a DAG is Tic Tac Toe, because once a cell is marked cannot be unmarked to go back to a previous state. On the contrary, Chess is an example of game that can be represented with a DCG, because players are allowed to move their pieces back to the position where they came from. When transposition tables are used to transform a game tree into a DCG there is an important issue to take into consideration. In a DCG the same node might correspond to multiple identical states that, however, have a different history. The history of played moves might influence the information related to such states, causing two types of problems: the *move-generation problem* and the *evaluation problem*. The first problem is caused when the sequence of actions that leads to a state plays a role in determining which actions are legal in it. In Chess for example, this problem happens with castling moves, which can be performed in a given state only if the King and the Rook have not previously been moved. The second problem arises when the history of a node influences its value. For example, this problem in Chess happens because of the rule stating that the game should be declared a draw if a board position repeats three times with the same player to move. Together, the move-generation problem and the evaluation problem are known as the *Graph-History Interaction* (GHI) problem (Palay, 1983; Campbell, 1985). The most trivial solution for the GHI problem would be to include in each state all relevant history information, thus the previously mentioned situations could be discriminated and associated to different nodes. However, memorizing all the relevant history might require too much memory. Moreover, the frequency of transpositions might be reduced, defying the purpose of using a transposition table. Different other solutions for the GHI problem have been studied (Breuker, 1998; Kishimoto and Müller, 2004). In general, they propose to memorize only part of the relevant history, while at the same time providing different mechanisms to detect nodes for which values are wrongly computed, so that they can be fixed.

Representation

Another issue to discuss regrading transposition tables is how to represent them. To reduce space usage, a finite *hash table* is a commonly used representation (Breuker, 1998). A *hash value* is associated to each state and a *hash function* is used to compute from it the *hash index* of the corresponding entry in the table. Optionally, each state can be associated also to a *hash key*, which is used to distinguish two states with the same hash index.

In game playing, one of the most used hashing techniques for transposition tables is Zobrist hashing (Zobrist, 1970). This type of hashing associates a random integer value to each element that composes the state (e.g. a piece-position pair in chess) and computes a 64-bits hash value of the state by performing a XOR of the values of these single elements. If the hash table has 2^n entries, the first n bits of the hash value are used to compute the hash index, while the remaining $64 - n$ bits are used as hash key. Although Zobrist hashing was proposed as a game specific solution, it has been used in GGP as well. The GGP agent CadiaPlayer used it to compute the hash value of game states represented in GDL. In GDL a state is represented as a set of predicates that are true in it (more details in Chapter 3). CadiaPlayer computes the hash value of a state with Zobrist hashing by associating an integer value to each symbol that forms the true predicates and then performing a XOR on all of them.

The use of a finite hash table can cause two types of errors: *type-1 errors* and *type-2 errors*. Type-1 errors occur when two distinct states have the same hash value. This type of error is difficult to detect, because the two states also have the same hash key. A possible solution to this problem could be using a *hash map* instead of a *hash table*. With a hash map the entire state can be stored as the key, such that if the states have the exact same hash value they can always be distinguished. Type-2 errors occur when two states have the same hash index and therefore they are associated to the same table entry. This event is referred to as *collision*. Type-2 errors are easier to detect by simply checking the hash keys of the states, which will be different. However, when a type-2 error occurs a mechanism has to be implemented to decide how to deal with the collision. One solution consists in storing at each table entry a linked list that contains an element for each state with the same hash index.

Transposition Tables in MCTS

Transposition tables are a general enhancement that has been applied to different search algorithms, like $\alpha\beta$-search and MCTS. In this thesis they are specifically applied to MCTS. It is relevant to mention that the use of transposition tables with MCTS gives different options to implement the UCT selection strategy. In this section, four different implementations of UCT, identified as UCT0, UCT1, UCT2, and UCT3 are discussed. The first one, UCT0, is the standard implementation of UCT without transposition tables, which is presented here for comparison. UCT1, UCT2 and UCT3 have been proposed by Childs, Brodeur, and Kocsis (2008) for MCTS with transposition tables. These UCT variants use the same UCT formula (i.e. Formula 2.9), but differ in how they compute the terms $\bar{q}_{(s,a_i)}$, $n_{(s,a_i)}$ and n_s, and in how they update the collected statistics. Below, a one-player game is used to give examples of how these UCT variants are applied. The examples consider that statistics are stored at the action level. Figure 2.11 represents the search tree of the one-player game that would be built when transposition tables are not used, and on which UCT0 can be applied. Figure 2.12 represents the search graph of the one-player game that would be built using transposition tables, and on which all other UCT versions can be applied. Note that this is a one-player game, therefore

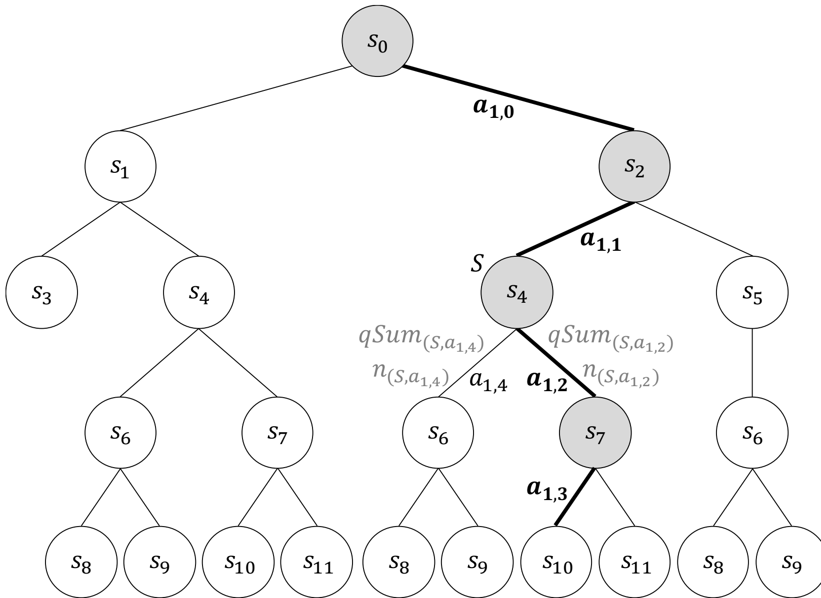


Figure 2.11: Application of UCT on a search tree built without using a transposition table.

the action subscript indicating the player is always 1. The second action subscript is used to discriminate among Player 1's actions.

UCT0. Without transposition tables, this variant computes the UCT value of an action of a player by considering only the statistics collected in the node reached by performing the exact sequence of actions selected so far. If the same state can be reached with a different sequence of actions, the statistics collected by performing such sequence are memorized in a different node, which is not considered by UCT0. Moreover, at the end of a simulation, UCT0 only updates the statistics of the actions in the nodes visited during the simulation. UCT0 can be applied to the tree in Figure 2.11. For example, to compute the UCT value of action $a_{1,2}$ in node S, UCT0 would compute the terms in the UCT formula as follows:

$$\begin{aligned} \bar{q}_{(s_4,a_{1,2})} &= \frac{qSum_{(S,a_{1,2})}}{n_{(S,a_{1,2})}} \\ n_{(s_4,a_{1,2})} &= n_{(S,a_{1,2})} \\ n_{s_4} &= n_{(S,a_{1,2})} + n_{(S,a_{1,4})} \end{aligned} \tag{2.7}$$

Note that here the notation $qSum_{(S,a_{1,j})}$ and $n_{(S,a_{1,j})}$ is used to specify that only the statistics of action $a_{1,j}$ in the particular node S are considered, even if there might be other nodes corresponding to the same state. The figure also gives an example of how backpropagation is implemented for UCT0. If the sequence of actions $a_{1,0}, a_{1,1}, a_{1,2}, a_{1,3}$ (i.e. the edges represented in bold) has

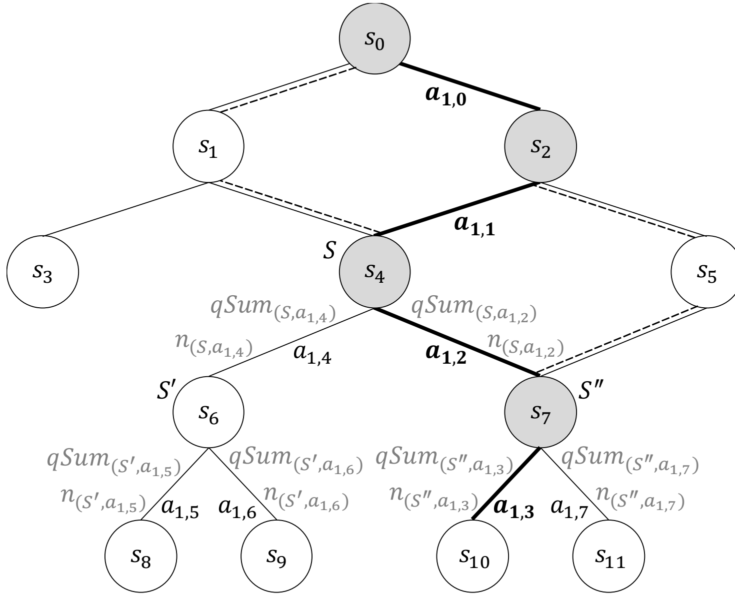


Figure 2.12: Application of UCT1, UCT2, UCT3 and UCT4 on a search graph built using a transposition table.

been visited during the simulation, only the action statistics in the gray nodes are updated.

UCT1. This variant is the same as UCT0, except that transpositions are taken into account. This means that the computation of the UCT value of an action in a state considers all the statistics accumulated for such state, even the ones obtained by reaching it from a different path. The backpropagation phase is the same as in UCT0, i.e. the payoff obtained at the end of a simulation is used to update statistics only for the nodes on the traversed path.[3] As an example, UCT1 can be applied to the search graph in Figure 2.12. With UCT1, if $a_{1,0}, a_{1,1}, a_{1,2}, a_{1,3}$ is the sequence of simulated actions, during backpropagation only the statistics in the gray nodes are updated. Moreover, the terms $\bar{q}_{(s_4,a_{1,2})}$, $n_{(s_4,a_{1,2})}$ and n_{s_4} needed to compute the UCT value of action $a_{1,2}$ in node S are obtained in the exact same way as for UCT0, the difference being that node S now contains all the statistics ever collected for the actions in state s_4.

UCT2. For this variant, the statistics update during backpropagation is the same as for UCT1. The difference is that, when computing the estimated value of an action a_i for player i in state s UCT2 uses the expected value of player i for the state s' reached by performing a_i in s. The node corresponding to state s' contains at least the same samples as the node associated to s, but

[3]Note that with transposition tables a node might have more than one parent, therefore it would be possible to backpropagate the payoff of the simulation also through the paths of all other parents.

it might contain even more samples, because it might have been visited from other paths as well. Therefore, with more samples, the value estimate of an action would be more accurate. As an example, if UCT2 is applied to the search graph in Figure 2.12, the computation of the UCT value of action $a_{1,2}$ in node S would use the following terms:

$$\begin{aligned}
\bar{q}_{(s_4,a_{1,2})} &= \frac{qSum_{(S'',a_{1,3})} + qSum_{(S'',a_{1,7})}}{n_{(S'',a_{1,3})} + n_{(S'',a_{1,7})}} \\
n_{(s_4,a_{1,2})} &= n_{(S,a_{1,2})} \\
n_{s_4} &= n_{(S,a_{1,2})} + n_{(S,a_{1,4})}
\end{aligned} \tag{2.8}$$

Note that node S'' might have been visited from the node with state s_5 as well, therefore its statistics might be based on more samples than the ones in S. However, also note that for the visits of state s_4 and action $a_{1,2}$ UCT2 still uses the statistics memorized in node S instead of the ones in S''. An alternative would consist in using the statistics of S'' to also compute the visits, as shown below:

$$\begin{aligned}
n_{(s_4,a_{1,2})} &= n_{(S'',a_{1,3})} + n_{(S'',a_{1,7})} \\
n_{s_4} &= n_{(S'',a_{1,3})} + n_{(S'',a_{1,7})} + n_{(S'',a_{1,5})} + n_{(S'',a_{1,6})}
\end{aligned} \tag{2.9}$$

This implementation was used by Kocsis, Szepesvári, and Willemson (2006a) when performing MC search with transposition tables. However, Childs *et al.* (2008) note that in MCTS this choice might lead the estimate for the value of $a_{1,2}$ in S to converge to the wrong value. This happens especially if state S'' has been visited mostly from other paths that did not include S as its parent.

UCT3. This variant is similar to UCT2, as it follows the same idea of computing the value of an action in a state s using the expected value of the state reached by performing such action in s. However, UCT3 further refines the estimated values of the actions by exploiting for each node the information of all its children recursively. This approach can be implemented by modifying the backpropagation phase with respect to the UCT approaches presented so far. During backpropagation, UCT3 updates each action a_i visited in the simulation using the expected value of the node reached by the action, instead of the actual simulation result. Moreover, whenever the statistics of an action are updated in a node, the update is recursively propagated backwards through all the actions that lead to the updated node. To give an example, looking at Figure 2.12, after a simulation that visited actions $a_{1,0}, a_{1,1}, a_{1,2}, a_{1,3}$, not only the statistics of the actions represented with a bold edge will be updated, but also the ones indicated by a dashed edge. With this particular update during backpropagation, the computation of the UCT value of an action can be performed in the same way as for UCT0 and UCT1, thus using the action statistics memorized in the node where the action is being evaluated. The difference will be that such statistics have been updated considering the statistics of all descendants of the current node.

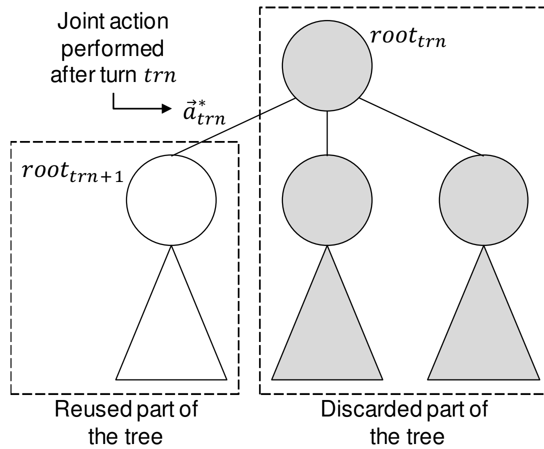


Figure 2.13: Tree reuse in MCTS.

Childs *et al.* (2008) compared all the presented variants on an artificial game tree, showing that the ones implemented for transposition tables (UCT1, UCT2 and UCT3) outperform the basic UCT0 algorithm that does not consider transpositions. Which variant among UCT1 UCT2 and UCT3 is more suitable seems to depend on the game. On an artificial game tree UCT2 performs slightly better than UCT1 and UCT3 performs slightly better than both UCT1 and UCT2. However, UCT3 requires a time consuming procedure to update statistics, therefore its usage is recommended for games where the time required to update the nodes is negligible when compared to the time necessary to simulate the game.

2.5.4 Tree Reuse

MCTS is used to select which action to play in each turn of a game. Whenever the search for a new turn starts, some implementations of MCTS simply discard the tree built in the previous turn and start building a new one from scratch. However, part of the information stored in the tree might still be relevant for the new turn as well, and it can be used to guide the search for the new turn instead of starting it without any knowledge. *Tree reuse* consist in keeping part of the tree and the corresponding memorized statistics in-between game turns. This search enhancement has been successfully applied in Lines of Action (Winands *et al.*, 2010), TwixT (Steinhauer, 2010), Ms. Pac-Man (Pepels *et al.*, 2014) and in GVGP (Soemers *et al.*, 2016).

Figure 2.13 shows how tree reuse works. The depicted tree has been created during the search at turn trn. When the joint action $\vec{a}^*_{trn}$ is performed at the end of turn trn the tree node that can be reached by performing such action is already part of the search tree. Therefore, when starting the search for turn $trn + 1$ this node can be used as the new root and its subtree, together with all the memorized statistics, can be reused during the new search. The old root, the siblings of the new root and their subtrees, which are not relevant anymore, can be discarded.

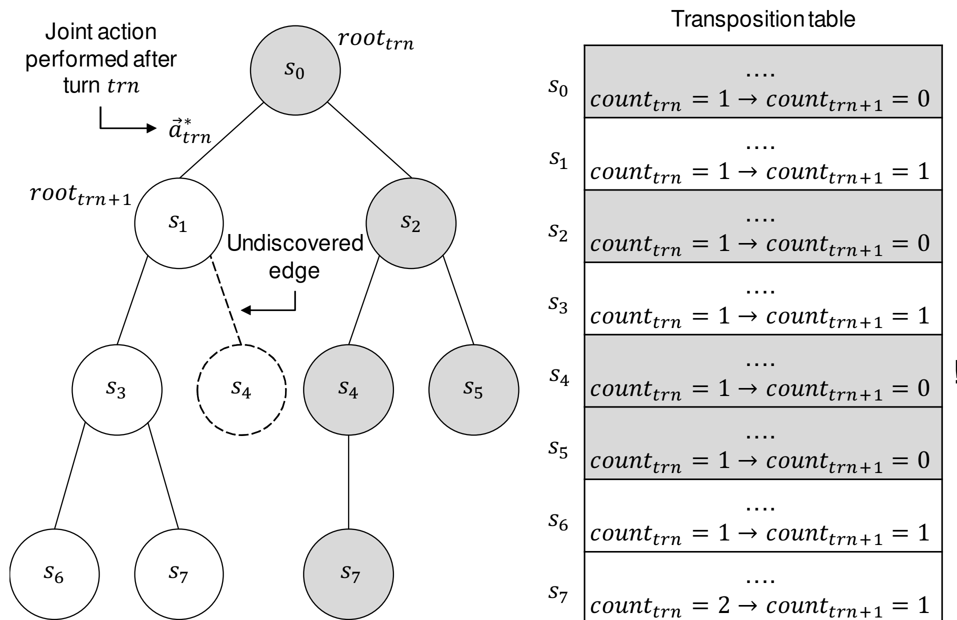


Figure 2.14: Example of graph reuse in MCTS with transposition tables.

In non-deterministic games, however, keeping part of the tree without any modification might have some negative effects. As mentioned in Subsection 2.3.1, the nodes in the search tree of a non-deterministic game might correspond to more than one possible state. When the root is initialized in turn $trn+1$ with the node created in the previous turn the exact state it corresponds to is known, because an action has been played in the real game. However, when the node was created in turn trn it might have corresponded to multiple states, therefore the information memorized in it and in its subtree might not be fully representative of the new root state at turn $trn+1$. A solution to this problem was proposed by Pepels *et al.* (2014), and it consists in decaying the statistics in the reused part of the tree before starting the search for the new turn. In this way, old statistics will have less impact on the new search, while at the same time providing it some guidance. The simplest way to decay statistics consists in multiplying them with a decay factor $\gamma \in [0,1]$. For example, if for each action a_i of each player i in a state s we are keeping the sum of payoffs obtained so far by playing the action, $qSum_{s,a_i}$, and the number of times the action has been visited, n_{s,a_i}, Formula 2.10 shows how these statistics are updated in order to decay them.

$$\begin{aligned} qSum_{(s,a_i)} &\leftarrow \gamma \times qSum_{(s,a_i)} \\ n_{(s,a_i)} &\leftarrow \gamma \times n_{(s,a_i)} \end{aligned} \tag{2.10}$$

The idea of tree reuse can be applied also to transposition tables. In this case it would be more appropriate to call it *graph reuse*, because transposition tables

represent the search tree as a graph. As we can do for the search tree, we can reuse information in a transposition table from one turn to the next. The difference is that for a transposition table it is not immediate to recognize which entries (i.e. nodes in the game graph) will be relevant for the subsequent turn.

An example of implementation of graph reuse for MCTS is shown in Figure 2.14. This is the implementation used by CadiaPlayer (Finnsson, 2012b), the agent developed for the Stanford GGP competition. This agent builds a game tree where each node contains a reference to the corresponding entry in the transposition table. Therefore, more nodes can point to the same table entry. An entry in the transposition table records all information about the corresponding state and keeps a counter *count* of all the nodes in the tree that reference to it. Like for tree reuse, at the end of a turn trn, the root node for the search in turn $trn + 1$ is initialized with the node reached by performing the selected joint action $\vec{a}^{*}_{trn}$. At this point the old root, the siblings of the new root and all their descendants (i.e. the gray nodes in the figure) are removed from the tree and for each removed node the counter of the corresponding table entry is decremented by 1. The figure shows for each table entry how the counter at the end of turn trn, $count_{trn}$, is updated before the start of turn $trn + 1$, $count_{trn+1}$. Table entries that are not referenced in the tree anymore, therefore with $count_{trn+1} = 0$, are deleted from the transposition table. These entries are colored in gray in the figure.

An issue with this implementation is that in-between turns some nodes might be removed from the transposition table even though they are still relevant for the search. This happens when a state that is still reachable in the new turn has been discovered only from the part of the tree that is being deleted at the end of the previous turn. For example, in Figure 2.14 the search has not discovered the dashed edge yet. Therefore, the table entry for state s_4 is erroneously identified as useless and removed from the transposition table. A new table entry will be created whenever the edge will be discovered. However, all the useful statistics accumulated in previous turns are lost.

An alternative solution to implement graph reuse is to record for each table entry the last turn in which it was visited. Before the start of a new turn, the only entries that are deleted from the transposition table are the ones that have not been visited in the last b turns. This solution does not completely avoid the risk of deleting table entries of states that are still reachable in later turns. However, it allows to control this risk by tuning the parameter b. Low values for b decrease the probability of keeping useless entries in the table, reducing space occupation, but at the same time they increase the probability of discarding entries that are still relevant. The opposite situation is caused by high threshold values.

2.6 Discussion

This chapter introduced relevant search techniques for this thesis, with focus on MCTS and its enhancements. The aim of the thesis is to investigate how MCTS can be used to support AGI in games, and GGP has been chosen as a suitable test domain for this task. The next step requires to identify environments that facilitate

the testing of multiple search techniques over possibly large and heterogeneous sets of games.

An option could be implementing from scratch various game environments that present different characteristics. However, this would require a considerable programming effort. Another option could be taking publicly available implementations of different game environments from different sources. Although, this would require effort in adapting the search agent to the characteristics of each domain implementation.

A solution to these problems comes for the effort of the research community in providing ready-to-use test environments for GGP. An ideal GGP test environment provides a large set of games that present different characteristics, and are represented with a common language. In this way, only the effort of implementing a single agent that is able to interpret this language is necessary.

Examples of such environments have been discussed in Chapter 1. Among them, two have been chosen for this thesis. The first is the environment of the Stanford GGP project, which provides a selection of abstract perfect-information games. The second is the framework of the GVG-AI project, which provides a selection of arcade-style video games. These two environments, together with their characteristics, and the language they use to describe games are presented in Chapter 3.

Chapter 3

Test Environments

Parts of this chapter are based on:

- Sironi, Chiara F., and Winands, Mark H.M. (2017). Optimizing Propositional Networks. *Computer Games*, Vol. 705 of *CCIS*, pp. 133–151.
- Soemers, Dennis J.N.J. and Sironi, Chiara F. and Schuster, Torsten and Winands, Mark H.M. (2016). Enhancements for Real-time Monte-Carlo Tree Search in General Video Game Playing. *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pp. 436–443.

This chapter describes the test environments used in this thesis: the *Stanford General Game Playing* (Stanford GGP) project and the *General Video Game AI* (GVG-AI) project. The first one focuses on GGP for abstract games, while the second on GGP for arcade-style video games. For each of the two environments, this chapter introduces the language used to describe the corresponding games, describes how the execution of a game run is managed, presents the rules of the associated competition and summarizes the common implementation details of the agents that are tested in the subsequent chapters. The Stanford GGP project is presented in Section 3.1, while the GVG-AI project is presented in Section 3.2. Finally, Section 3.3 discusses research directions worth investigating to improve the performance of MCTS for these environments.

3.1 Stanford General Game Playing

The Stanford GGP project (Genesereth and Thielscher, 2014) focuses on the creation of game-playing agents that are able to tackle a wide variety of abstract games assuming no-prior knowledge about them. Agents are provided with the game rules specified in the *Game Description Language* (GDL) (Love *et al.*, 2006), and have a limited amount of time (usually a few seconds) for each turn to select a move. To promote research on the topic, since 2005 an annual GGP competition has been established (Genesereth *et al.*, 2005) as part of the project. Due to this competition

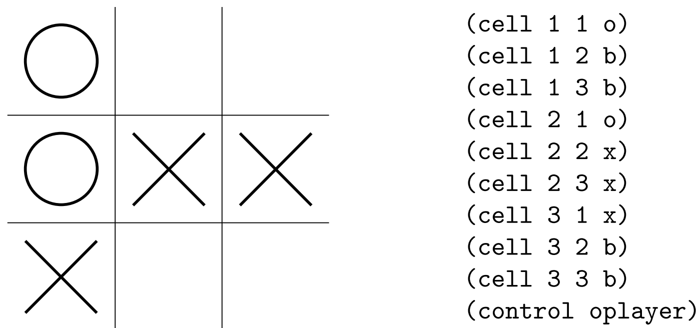


Figure 3.1: Representation of a state of Tic Tac Toe with GDL.

format GGP agents can be directly matched against each other to compare the performance of the approaches they use.

More details about the Stanford GGP project are given in the following subsections. GDL is presented in Subsection 3.1.1. Next, the game management procedure adopted to match agents against each other is described in Subsection 3.1.2. Subsection 3.1.3 introduces the Stanford GGP competition. Finally, Subsection 3.1.4 summarizes the main characteristics of the game-playing agent developed and tested in this thesis.

3.1.1 Game Description Language

GDL is a Prolog-based language proposed to represent game rules in a compact and modular format (Love *et al.*, 2006). It can be used to represent any finite, deterministic, perfect-information, turn-based, simultaneous-move game with any number of players. GDL can also model sequential-move games by considering them as simultaneous-move games where players that do not have to move in a given turn return a "null" action with no effect on the game. This "null" action in GDL is represented with a special keyword, *noop*. There exist also multiple extensions of GDL, which enable game authors to specify larger sets of games. GDL-II (Thielscher, 2011), which stands for *GDL with Imperfect Information*, gives the possibility to represent any finite game with randomness and imperfect information. GDL-III (i.e. *GDL-II with Introspection*) (Thielscher, 2017) further extends GDL-II adding the possibility to represent epistemic games, i.e. games where the rules depend on the knowledge of the players. Finally, rtGDL (*Real-Time Game Description Language*) (Kowalski, 2016) builds upon GDL-II adding the possibility to represent real-time games, where time-based events can be modeled. This thesis focuses on the part of the Stanford GGP project that deals with perfect information, turn-based games, therefore using only GDL and none of its extensions.

A state in GDL is represented as a set of true propositions. Figure 3.1 shows a state of Tic Tac Toe with the corresponding representation in GDL. To give an example, the GDL proposition `(cell 1 1 o)` means that the cell at position $(1, 1)$ in the grid is marked by the player that uses the symbol `o`, while the proposition `(cell 1 1 b)` means that the cell at position $(1, 2)$ in the grid is blank. The proposition

`(control oplayer)` means that in the state the player that should move next is the one that uses the symbol `o`.

When the game rules are specified in GDL, variables are used to reduce the length of game descriptions. A variable in GDL always starts with a question mark, for instance $?x$. GDL also provides 101 game-independent constants that represent the integers from 0 to 100. Rules in GDL are represented with the *prefix* notation. For example, the rule (`?f`$_1$ `<= ?f`$_2$`, ..., ?f`$_n$) is the equivalent of the logic implication $f_1 \Leftarrow f_2 \wedge ... \wedge f_n$, where `?f` and f represent propositions. Special relations, which are game independent, are used to define different game elements and the game dynamics (i.e. the goals, the legal actions, the transition function,...). Moreover, a finite arbitrary number of other relations can be defined by the game author to specify game-dependent concepts that support the definition of the game-independent relations. The game-independent relations, identified by particular keywords, are listed below. Parts of the GDL game description of Tic Tac Toe are used as examples. The complete game description can be found in Appendix A.1.

- **(role ?r):** specifies that `?r` is a role in the game. This relation is used to determine which players are participating in the game. For example, the GDL rules for Tic Tac Toe include the following definitions of roles:

  ```
  (role xplayer)
  (role oplayer)
  ```

- **(input ?r ?a):** specifies that `?a` is a feasible action for player `?r` in the game. In Tic Tac Toe, for example, a player `?r` can mark any cell or perform the *noop* action. This is specified by the following relations:

  ```
  (<= (input ?r (mark ?x ?y)) (index ?x) (index ?y) (role ?r))
  (<= (input ?r noop) (role ?r))
  ```

  In Tic Tac Toe, the game-dependent `index` relation is used to indicate that `?x` and `?y` have to be feasible indices for the game. An integer `?x` is a legal index for Tic Tac Toe if the game description contains the proposition `(index ?x)` (e.g. the game description of Tic Tac Toe specifies `(index 1)`, `(index 2)`, and `(index 3)`, because the grid has three rows and three columns.).

- **(base ?f):** specifies that `?f` is one of the propositions that are used to represent the state of the game. The base relation in Tic Tac Toe is specified as follows:

  ```
  (<= (base (cell ?x ?y b)) (index ?x) (index ?y))
  (<= (base (cell ?x ?y x)) (index ?x) (index ?y))
  (<= (base (cell ?x ?y o)) (index ?x) (index ?y))
  (<= (base (control ?r)) (role ?r))
  ```

  This means that a GDL state of Tic Tac Toe is represented by propositions of the form `(cell ?x ?y b)`, `(cell ?x ?y x)` and `(cell ?x ?y o)`, where `?x` and `?y` have to be a pair of feasible indices, and propositions of the form

`(control ?r)`, where `?r` has to be a feasible role in the game. The symbols `b`, `x` and `o` mean that a cell is blank, marked by `xplayer` and marked by `oplayer`, respectively.

- **(init ?f):** specifies which propositions `?f` are true in the initial state of the game. For example, in Tic Tac Toe we have the following relations:

```
(init (cell 1 1 b))
(init (cell 1 2 b))
...
(init (cell 3 3 b))
(init (control xplayer))
```

These relations are stating that in the initial state of Tic Tac Toe each cell in the grid is blank and that the first player to move is `xplayer`.

- **(true ?f):** specifies that proposition `?f` is true. In GDL, all the propositions `?f` that can be an argument for the `true` relation are the ones identified by the *base* relation, therefore propositions of the form `(true ?f)` are referred to as *base propositions*. The `true` relation usually appears in the body of GDL rules and is used to define which proposition have to be true in the current state in order for the relations in the head of the rule to be true. See below the examples for the *legal* and *next* relations.

- **(legal ?r ?a)** specifies that an action `?a` is legal for player `?r`. This relation usually appears as the head of the rules that specifies which proposition have to be true in a state in order for the move to be legal for the player. For example, in Tic Tac Toe the *legal* relation is used in the following rules:

```
(<= (legal ?r (mark ?x ?y))
(true (cell ?x ?y b))
(true (control ?r)))

(<= (legal xplayer noop)
(true (control oplayer)))

(<= (legal oplayer noop)
(true (control xplayer)))
```

The first rule specifies that the action `(mark ?x ?y)` is legal for player `?r` if both the propositions `(true (cell ?x ?y b))` and `(true (control ?r))` are true in the current state. This means that a player `?r` can mark the cell at position `(?x, ?y)` only if the cell is blank and it is her turn to move. The second and third rule are specifying that a player can perform the *noop* action only in states where it is the turn of the other player to move.

- **(does ?r ?a):** specifies that player `?r` performs action `?a`. In GDL, all the possible pairs of arguments `(?r ?a)` for the `does` relation are the ones identified

by the `input` relation, therefore propositions of the form `(does ?r ?a)` are referred to as *input propositions*. The `does` relation usually appears in the body of GDL rules with the *next* relation in the head and is used to define which action the player has to perform in the current state in order for the proposition in the *next* relation to be true in the next state. See below the example for the *next* relations.

- **(next ?f):** specifies that proposition `?f` is true in the next state. This relation usually appears as the head of the rules that specify which propositions have to be true in the current state in order for a proposition to be true in the next state. Two examples of rules with the *next* relation in the Tic Tac Toe game description are the following:

```
((<= (next (cell ?x ?y x))
(does xplayer (mark ?x ?y))
(true (cell ?x ?y b)))

(<= (next (control xplayer))
(true (control oplayer)))
```

  The first rule is stating that the cell at position `(?x ?y)` will be marked with the symbol `x` in the next state if in the current state the cell is blank and `xplayer` marks it. The second rule is stating that in the next state it will be the turn of `xplayer` if in the current state it is the turn of `oplayer`. Similar rules can be defined for `oplayer`, and other rules can be defined to specify which conditions are necessary in the current state for a cell to remain blank or remain marked in the next state.

- **terminal:** specifies that the current state is terminal. This relation usually appears as the head of the rules that define for which conditions a state is terminal. For example, in Tic Tac Toe we have the following rules:

```
(<= terminal (line x))

(<= terminal (line o))

(<= terminal (not open))
```

  These rules are stating that the game will end in states where there is either a line of three `o`s or a line of three `x`s, or no blank cell for the players to mark. The relations *line* and *open* are game-specific relations defined by the game author.

- **(goal ?r ?s):** specifies that player `?r` gets score `?s` in the current state. This relation appears as the head of the rules that define for which conditions a player gets a certain score. In GDL the *goal* relation must always be defined for terminal states, while it is not compulsory to define it for non-terminal

states. The values defined for the players' scores in GDL are always included in the interval $[0, 100]$. Below are some of the rules that specify the *goal* relation in Tic Tac Toe:

```
(<= (goal xplayer 100)
(line x))

(<= (goal xplayer 50)
(not (line x))
(not (line o))
(not open))

(<= (goal xplayer 0)
(line o))
```

These rules are specifying the scores for `xplayer` in a terminal state. The player will get a score of 100 if there is a line of `x` marks, a score of 0 if there is a line of `o` marks, and a score of 50 if there is no blank cell left and none of the players formed a line with her marks. Similar rules can be defined to specify the scores for `oplayer`.

By processing game rules written in GDL, which constitute a logic program, a player is able to reconstruct the dynamics of a finite state machine for the game. This state machine can be seen as the game model that can be used to simulate the game. The initial state can be directly computed using the `init` relation. Given the current state, identified by the base propositions that are true, the truth value of the propositions identified with the `terminal`, `goal` and `legal` relations can be computed. If also the truth value of the input propositions is given (i.e. the action that each player is performing is known), the truth value of the propositions identified with the `next` relation can be computed as well. This will give the set of base propositions that are true in the next state. This procedure can be repeated until a terminal state is reached. As an advantage over memorizing and using the state machine directly, GDL reduces both the space required to memorize game rules and the computational cost of reasoning on them. Agents implemented for the Stanford GGP competition have to include a component that interprets the GDL rules to build a state machine (i.e. the game model) and reasons on them in the way described above. This component is often referred to as the *reasoner*.

Note that an alternative to providing the GDL game rules to the agents would be to give them the game model directly, like described in Subsection 3.2.2 for the GVG-AI framework. However, having access to the GDL rules enables agents to extract knowledge from them and use it to enhance the search. An example of work in this direction is the one of Schiffel (2011), which investigates a knowledge-based approach for GGP. He proposes to use heuristic search with an evaluation function for non-terminal states that is generated automatically by analyzing the GDL game rules. This analysis determines how likely it is for a terminal or goal proposition to become true from the current state being evaluated. Another example is the work of Finnsson (2012b), which proposes the *Predicate-Average Sampling technique* (PAST)

and the Feature-to-Action sampling Technique (FAST). PAST uses the description of the state as a set of GDL propositions to bias the play-out in a way similar to MAST, but considering statistics of proposition-action pairs instead of actions only. FAST, instead, extracts board game features (i.e. pieces types and cells) from the rules and uses them to evaluate game states.

3.1.2 Game Management

To enable GGP agents to compete against each other there is the need for a mediator (or *game manager*) that takes care of managing the match (i.e. an instance of a game). Such game manager should communicate the rules of the game to the agents participating in the match, keep track of the state of the game, update it with the moves played by the agents, make sure that agents play legal moves and abide to the game rules, and finally determine the winner of the game. The Stanford GGP project defines a communication language that the game manager and the agents can use to set up and perform game matches (Genesereth and Thielscher, 2014). The communication takes place through HTTP messages and each agent must be connected to the internet, listening on a particular port. Before starting a match, the game manager must know the IP address and port of each of the participating agents and must have access to a database where to retrieve the GDL description of the game to be played.

When performing a match, the game manager uses the following types of messages to communicate with each agent that should take part in the match:

- **info():** this message is used to check if an agent is ready to play a new match. If the agent is ready it should respond with *available*, otherwise with *busy*.
- **start(*id, role, description, start-clock, play-clock*)**: this message communicates to the agent that a new match is starting, and gives information about the match. More precisely, it gives the agent the unique *id* of the match that is starting, the *role* that the agent must play in the match (taken from the game description) and the game *description* in GDL. In addition, a *start-clock*, which defines the amount of time in seconds that the agent has available to prepare to play the game, and a *play-clock*, which defines the amount of time in seconds that the agent can spend on selecting an action in each turn are given. After receiving this message, and before the start-clock is over, the agent should respond to the game manager with a message saying that it is ready to play the match.
- **play(*id, move*):** before each turn, the game manager sends this message to each agent to request an action. The message specifies the *id* of the match it refers to, and the joint action (*move*) performed by all agents in the previous turn. After receiving this message, each agent can use the joint action to update its representation of the state. Subsequently, before the play-clock is over, each agent has to reply to the game manager with the action it wants to play. If the agent does not respond in time, the game manager will substitute its action with a random legal one. Note that in all messages the actions of each player are represented as GDL proposition.

- **stop(*id, move*):** this message is used to notify the agents that a match is over. The message contains the *id* of the match it refers to and the last played joint action (*move*). Agents should respond to this message with *done*.
- **abort(*id*):** this message is used by the game manager to notify the agents that the match with the given *id* is terminating abnormally. After receiving this message, agents can stop their search and return to a *ready* state, returning to the game manager a message saying that they are *done*.

By setting up a game manager and using this protocol, agents can be matched and compared against each other. It is interesting to mention that this protocol has a possible drawback. If agents are running on different machines during a match, their performance might be influenced by the hardware specifications. For agents that use search-based techniques, for example, better hardware would mean that they can visit a bigger portion of the search space, and possibly improve the quality of the selected actions.

3.1.3 Competition

To give a common test bed to GGP agent developers and also to promote research on the topic, the Stanford GGP competition initiated in 2005 (Genesereth *et al.*, 2005). This competition tests all participating agents on a series of games with different levels of difficulty and different characteristics (e.g. single- and multi-player, competitive and cooperative, etc...). Matches are run following the protocol described in Subsection 3.1.2. Usually, during the competitions agents are not only tested for their ability to win games, but also for their ability to play legal moves consistently and for being able to respect the time constraints (i.e. the start- and play-clock settings). In addition, because one of the aims of GGP is to foster on-line learning ad on-line knowledge acquisition about the game that is being played, the start- and play-clock are usually set to a few seconds for each played game.

Over the years, many agents have taken part in the competition and various different approaches have been tested. The first editions of the competition were dominated by agents that were using traditional tree search approaches. The winners of 2005 and 2006, CLUNEPLAYER (Clune, 2007) and FLUXPLAYER (Schiffel and Thielscher, 2007) were using minimax search combined with heuristic functions learned automatically. Since 2007 the top agents have all been based on some variants of MCTS. CADIAPLAYER (Björnsson and Finnsson, 2009) was the winner in 2007, 2008 and 2012. ARY (Méhat and Cazenave, 2010) won in 2009 and 2010. TURBO TURTLE, the agent developed by Sam Schreiber, won in 2011 and 2013. The champion of 2014 was SANCHO (Draper and Rose, 2014) and the one of 2015 was GALVANISE, developed by Richard Emslie. An exception was WOODSTOCK (Piette, 2016), the winner of the last competition ran so far, in 2016. This agent was not using MCTS, but was transforming the GDL game description into a Stochastic Constraint Satisfaction Problem (SCSP). The obtained SCSP was then decomposed into one-step SCSPs (also known as μSCSPs), one for each game turn. A stochastic constraint solver was applied on each μSCSP to find feasible solutions, which were then evaluated with UCT sampling to find the ones with highest payoff.

3.1.4 GGP Base Agent

Agents for the Stanford GGP project can be implemented in any programming language as long as the communication protocol is respected. However, to facilitate the development of the agents, a code base is provided, the *General Game Playing Base Package* (GGP Base Package) (Schreiber and Landau, 2016). The GGP Base Package is implemented in Java and offers various functionalities, among which the implementation of a game server that manages game matches and the implementation of game-playing agents. The code provided for the agents already implements the communication protocol defined in Subsection 3.1.2 and a method to reason on the GDL rules and interpret them as a state machine. This code can be easily extended to implement and test different search methods.

The MCTS agent used in the experiments presented in the next chapters has been developed extending the code in the GGP Base Package. Given that GDL represents each game as being simultaneous move, this agent performs the search in every turn of the game. Therefore, when the game has actually sequential moves the agent keeps pondering in the opponent's turn as well. The main characteristics of this agent are summarized below. First, details about the implementation of UCT are given. Subsequently, the representation of the tree as a transposition table is discussed, together with all the implementation aspects of MCTS that are influenced by this representation.

Implementation of UCT

Given that the games played by the agent are expressed in GDL, which represents them as simultaneous move games, the agent implements the DUCT selection strategy (i.e. Decoupled UCT for simultaneous move games, Chapter 2, Subsection 2.4.1). The successful application of DUCT to GGP (Björnsson and Finnsson, 2009; Tak *et al.*, 2014a), together with the reduced space usage with respect to SUCT, is what motivates its use as selection strategy. When applied to games that have in fact sequential moves, DUCT behaves exactly like the standard implementation of UCT. Note that enhancements of the UCT selection strategy that modify the UCT value computed for the actions (e.g. RAVE and PH) can be easily implemented keeping the same decoupled structure of DUCT.

The selection strategy of the MCTS agent is enhanced with the definition of the FPU of yet unexplored moves (see Subsection 2.5.1), and with the random tie-breaking rule to choose among actions that have the same value during selection (see Subsection 2.4.1). More precisely, to implement the random tie-breaking rule the agent defines a *value offset* parameter VO. When selecting an action in a state, it will select a random action among all the actions that have the value less than VO away from the maximum computed action value. For example, when selecting an action a_i in state s for player i using UCT, a random action will be selected in the following set of actions: $\{a_i \in A_{(s,i)} : UCT(s, a_i) \in [UCT_{max} - VO, UCT_{max}]\}$, where $UCT_{max} = \max_{a_i \in A_{(s,i)}}(UCT(s, a_i))$. For all the experiments, the default value of FPU is set to 1, and the default value of VO is set to 0.01. Note that these settings consider that the payoffs of the agent are rescaled to $[0, 1]$ when using UCT.

Implementation of MCTS and Transposition Tables

The agent represents the search tree as a transposition table using a hash map. The entire state is used as hash key and the hash index is computed directly on such state. Depending on the type of reasoner that the agent is using, a state is represented differently and its representation influences how the hash function is implemented. Two types of state representations can be distinguished:

Set of GDL propositions. A state is represented as a set of GDL propositions that are true in it. The hash value of such state is computed as the sum of the hash values of each proposition that is true in the state. The hash value of a proposition is computed with the default *Object.hashCode()* method, called on the class that represents a GDL proposition in the GGP Base Package.

Sequence of bits. A state is represented as a sequence of bits, where each bit corresponds to a GDL base proposition. A bit is set to 1 if the corresponding proposition is true in the state and to 0 otherwise. The hash value of such a state is computed with the default method provided by the class that is used to represent the sequence of bits, namely the *org.apache.lucene.util.OpenBitSet* class of the Apache Lucene library.[1]

Using the entire state as hash key ensures that the transposition table does not suffer from type-1 errors. The risk of type-2 errors is still present, although easy to detect by simply comparing the two states that have the same hash index. When such an error occurs, the involved states are memorized at the same index of the hash map using a linked list. To be also noted is that, because GDL is used to define the game model, the agent does not have to deal with the GHI problem directly. If the history of moves that led to a certain state is relevant, GDL already represents it as part of the state. The agent can thus assume that states that have the same GDL representation are indeed identical states, with the same legal actions and the same value.

The use of transposition tables influences various aspects of the implementation of MCTS. First of all, because of transposition tables, the agent in this thesis adopts "open-loop" MCTS even when they are playing deterministic games. Given that the state representation is used as key for the hash map to retrieve the corresponding tree node, it would be necessary to memorize all next states in the parent node so that they could be used as keys to retrieve the child nodes. Therefore, to reduce the amount of memory used by the transposition table, whenever an action is chosen in a node the next state is generated with the game model, and the corresponding tree node is retrieved from the transposition table using such state as key.

Also the implementation of UCT is affected by the transposition table. Among the UCT variants for transposition tables presented in Subsection 2.5.3 the agent in this thesis uses UCT1. Although UCT1 seems to perform slightly worse than UCT2 and UCT3, the performance of all three variants is still close (Childs *et al.*, 2008). Moreover, UCT1 it is the one with the most efficient computation of the UCT value when games are all modeled with simultaneous moves and DUCT is implemented.

[1] `https://lucene.apache.org/core/4_8_1/`

To compute the value of an action a_i for player i in a node, UCT1 uses the statistics memorized for the action itself, which are already recorded as a single value. UCT2 and UCT3, instead, use the statistics of the actions available in the nodes reached by performing a_i (UCT3 even does this recursively). In a simultaneous move game represented with DUCT there might be multiple next states reached by performing the action a_i in a node, depending on the moves that other players perform. This would mean that to obtain the expected value of this action it is necessary to sum over all these nodes, which is time consuming.

The final aspect that is influenced by the use of a transposition table is the implementation of graph reuse. The agent in this thesis implements graph reuse as discussed in Chapter 2, at the end of Subsection 2.5.4. Each table entry is stamped with the last turn in which it was visited. At the beginning of a new turn it is eliminated if it has not been visited in the past b turns. For the agent, b is set to 2. Other than eliminating entries that have not been visited in the last 2 turns, no further decay of statistics memorized in the transposition table is performed between game turns, because all the games the agents deals with are deterministic.

3.2 General Video Game AI

The GVG-AI project was born to foster research on GVGP (Levine *et al.*, 2013). It focuses on the creation of gents that are able to play many different 2D arcade-style real-time video games, which are defined using the Video Game Description Language (VGDL) (Ebner *et al.*, 2013; Schaul, 2013). As part of the GVG-AI project, the GVG-AI framework has been developed, which provides researchers with a common platform for testing their agents. Since 2014, the GVG-AI competition has been running using this framework. For the first two years, the GVG-AI competition consisted only of a single-player planning track, while later a 2-player planning track, a single-player learning track, a game generation track and a level-generation track have been added (Perez-Liebana *et al.*, 2018).

This thesis focuses on the single-player planning track of the competition. In the next subsections more details about the GVG-AI project are given, mainly describing the characteristics that are relevant for this competition track. First, Subsection 3.2.1 introduces VGDL. Subsequently, Subsection 3.2.2 explains how games are managed in the GVG-AI framework, while 3.2.3 describes the structure of the competition. Finally, the agents used in this thesis are presented in Subsection 3.2.4.

3.2.1 Video Game Description Language

VGDL is a high-level language that describes 2D arcade-style video games in a concise manner (Ebner *et al.*, 2013; Schaul, 2013). At first, it was used only to define single-player games. Subsequently, it has been extended for two-player video games (Gaina, Perez-Liebana, and Lucas, 2016) and for games with real-world physics (Perez-Liebana *et al.*, 2017). In the rest of this subsection, the focus is on the aspects of VGDL that are relevant for single-player video games, which are the video games this thesis considers.

Although inspired by GDL, in that it aims at representing a large number of heterogeneous video games, VGDL presents substantial differences. As opposed to the abstract games considered by the Stanford GGP project, video games include non-deterministic behavior, either due to the presence of NPCs or random events. Moreover, state changes might not only be influenced by the players' action but also by NPCs' actions or by time-based events, which might happen at any moment and asynchronously. Therefore, a turn-based model it is not sufficient anymore, and a continuous representation is necessary. In addition, the dynamics of the game might be defined also by interaction and collisions between the various game elements and the players, and a video game description language should be able to model this as well. All these considerations were taken into account when designing VGDL.

VGDL is used to model real-time, non-deterministic games. A game in VGDL is described as a set of objects that are placed in a two-dimensional space and are characterized by certain properties and/or behaviors. Objects can move, spawn, disappear, spawn new objects, transform or change properties. They can also interact with each other by means of collisions, and the consequences of a collision are given in the VGDL rules. In VGDL two main components are used to specify a game: the *game description* and the *level description*.

The game description defines the game dynamics and the interactions among game objects. Below, the four main parts in which a game description is divided are explained. Parts of the game description of the game of Zelda are used as example. The complete game description for the game can be found in Appendix A.2.

- **SpriteSet:** defines the *objects* used in the game, their *type* and their *properties*. A property is defined with the syntax `propertyName=propertyValue` and can specify how the object is visualized, how it is affected by the physics of the game, what its resources are and which parameters control its behavior. For example, in Zelda this is the definition of an object identified as `key`:

  ```
  key   > Immovable color=ORANGE img=oryx/key2
  ```

  Here, `Immovable` specifies the type of the object, while `color=ORANGE` and `img=oryx/key2` specify two of its properties. More precisely, the color of the key is orange and the image used to represent the key in the game is "oryx/key2.png". The objects are organized in a tree hierarchy, which is defined by the indentation of each line. Each object inherits the properties of its ancestors. For example, the following is part of the objects definition in Zelda:

  ```
  avatar  > ShootAvatar stype=sword frameRate=8
    nokey   > img=oryx/swordman1
    withkey > color=ORANGE img=oryx/swordmankey1
  ```

  Looking at the indentation, it is possible to deduce that `avatar` is the parent of `nokey` and `withkey`, therefore both of them will be avatars of type `ShootAvatar`, will be able to use a sword (defined in a separate line of the description) and have the frame rate set to 8.

- **LevelMapping:** associates (a subset of) the defined objects to symbols that can be used to define the initial state of the game in the level description. For example, in Zelda the following lines are part of the *LevelMapping* block:

```
A > floor nokey
w > wall
. > floor
```

  Here, the symbol `A` represents the agent without a key, `nokey`, on a `floor` tile, `w` represents the wall object and `.` represents a floor tile with no other object on it.

- **InteractionSet:** defines what happens when there is a collision between two objects. More precisely, each line in the InteractionSet maps two objects to an *effect* that takes place when they collide and can specify optional properties for this effect. As part of the properties of an effect, a change in score can also be specified, both positive, for an increase, and negative, for a decrease. Note that there are no limitations on the value of a score change. Therefore, the range of score values that can be obtained in GVG-AI is unbounded, unlike the score values in the Stanford GGP project. As an example, below is part of the InteractionSet of Zelda:

```
movable wall > stepBack
enemy sword > killSprite scoreChange=2
avatar enemy > killSprite scoreChange=-1
nokey key > transformTo stype=withkey scoreChange=1 killSecond=True
```

  The first lines means that no `movable` object can pass through a `wall` (i.e. it has to step back). The second line means that whenever an `enemy` and the `sword` collide the enemy is killed and the score of the player is increased by 2 points. The third line means that whenever the `avatar` collides with an `enemy` the avatar is killed and the player loses 1 point. Finally, the fourth line means that if the avatar without a key (i.e. `nokey`) collides with the `key`, then it transforms into an avatar with the key (i.e. `withkey`), the player gets 1 point and the `key` object disappears.

- **TerminationSet:** defines the *termination conditions* for the game. Each line corresponds to a separate condition, and properties can be optionally specified for each condition. For example, in Zelda the following two termination conditions are specified:

```
SpriteCounter stype=goal   win=True
SpriteCounter stype=avatar win=False
```

  The first condition means that whenever the number of `goal` sprites reaches 0, i.e. the avatar with the key collided with the goal and the goal disappeared, the player wins the game. The second condition, instead, means that the player loses the game whenever it is killed, i.e. the number of avatars in the game reaches 0.

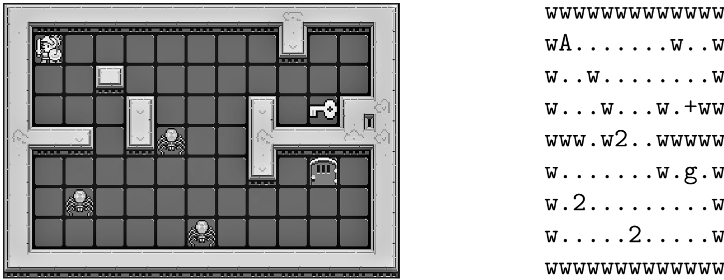


Figure 3.2: Representation of a level of Zelda with VGDL.

The level description describes the layout of the two-dimensional space of the game and the initial position of all the objects. It consists in a text file with a matrix of symbols, each of which represents a different type of object in the game. The symbols used in the level description are (a subset of) the ones defined in the *LevelMapping* block of the game description. Note that an arbitrary number of different levels can be designed for the same game. As an example, Figure 3.2 gives a level description for the game of Zelda and the corresponding graphical visualization.

VGDL is always coupled with software that can parse the game rules and create the corresponding implementation of the game. Each *object* defined in the *SpriteSet* of the game description corresponds to one of the *classes* implemented in the software. The class of an object is identified by the *type* specified for it, and the *class parameters* correspond to the *properties* that are defined in the game description for the object. The *effects* of collisions specified in the *InteractionSet* of the game description are implemented in *methods* provided in the software. If specific *properties* are specified for the effect of a collision, they correspond to *parameters* of the classes that implement the effects. The software also contains the implementation of predefined *methods* that test the *termination conditions* for the games. Once the game model has been built, the software can parse the level description and create the initial state of the corresponding game level by placing the objects at their position in the grid that represents the space of the game.

3.2.2 Game Management

As opposed to the Stanford GGP project, for which agents can be implemented independently from the game manager as long as the communication protocol is respected, the GVG-AI project requires agents to be directly integrated in the framework that manages the games. This framework is known as the *GVG-AI framework* (Perez-Liebana *et al.*, 2016; Perez-Liebana, 2018), is implemented in Java and offers various functionalities that are necessary to run video games. The fact that the GVG-AI project requires all agents to be integrated in the GVG-AI framework means that they all have to run on the same hardware. If compared to game management in the Stanford GGP project, this set up makes sure that the hardware characteristics do not influence the relative performance of the agents.

A single-player video game in the GVG-AI framework is managed as follows. The

game flow is discretized into game *ticks* with a duration of a few milliseconds in order to simulate the real-time nature of the games. First of all, the framework retrieves the VGDL game and level descriptions and parses them to build a game model, called forward model in the framework. This game model contains information about the initial state of the game and can be advanced to future states using the player's actions. With this game model the framework keeps track of the flow of the game, of the position of the objects and their collisions. It also can detect when the game is over and how the score of the player changes. At this point, the agent that has to play the game is created, is assigned to play the avatar role and is given a short amount of time to prepare to play the game. Once the initialization time for the agent is over, the framework starts running the game. At the start of each game tick, the agent is given a copy of the game model and is queried for the next action to play. Depending on the game, the agent can select an action among a subset of the following actions: $\{UP, LEFT, DOWN, RIGHT, USE, NIL\}$. A game tick usually lasts a few milliseconds, therefore the agent has a short amount of time to select the next action. Before the end of the game tick, the agent can use its copy of the game model to simulate the effect of its actions on the game. However, games might be non-deterministic, therefore there is no guarantee that the state returned by the game model of the agent will be the same as the one that would be reached by playing the same action in the actual game. The game continues until a terminal state is reached or until 2000 ticks have been performed.

Although the concept is similar to the one of the Stanford GGP project, analyzing the way game management is implemented in the GVG-AI framework highlights substantial differences between the two projects. First of all, in the GVG-AI framework agents have access only to the game model and not to the VGDL description of the game. Stanford GGP agents, instead, can analyze the GDL game rules directly. Moreover, the real-time nature of the GVG-AI games poses an extra challenge. Stanford GGP agents have a few seconds to prepare to play the game and then a few seconds per turn to select an action. In the GVG-AI framework the initialization time is usually around 1s, while the time to select a move for each tick is just a few milliseconds. This means that in GVG-AI agents that are based on tree search algorithms will be able to visit a smaller portion of the game tree. Finally, another difference is the branching factor. In GVG-AI the set of available actions for the player contains at most 6 actions, while games developed for the Stanford GGP project have varying branching factors.

3.2.3 Single-Player Planning Competition

To foster research on GVGP and to provide a common test environment for GVGP agents the GVG-AI competition has been running since 2014 (Perez-Liebana *et al.*, 2016). The first editions of the competition only featured the single-player planning track. This track evaluates the participating agents on a set of singe-player games, which are not revealed until the competition starts. Each game has 5 different levels, and the agents have to play each of them multiple times. For the competition, the initialization time for the agents is set to 1s, while the tick duration is set to 40ms. If the agent takes more than 40ms to return an action, but less than 50ms, then its

action is substituted by the *NIL* action as a penalty and the game continues. If the agent takes more than 50ms to return an action, it will be disqualified.

For each played game, the participating agents are ranked according to three different criteria, listed below in order of importance:

- **Wins:** number of wins achieved by the agent over all the played runs of the game.
- **Score:** average score obtained by the agent over all the played runs of the game.
- **Time:** total number of ticks that the agent spent playing all the runs of the game.

Players are awarded points for each game in the set according to their rank, following the Formula 1 scoring system. Therefore, players will get 25, 18, 15, 12, 10, 8, 6, 4, 2 or 1 points if they placed first, second, third, etc., respectively. The final winner is decided by summing for each agent the points gained for each game.

So far, five editions of the single-player planning track of the GVG-AI competition have taken place, some of which were divided into multiple legs. Many of the agents that participated in the first edition were based on tree search methods, including the winner, OLETS (Perez-Liebana *et al.*, 2016), which was using an open-loop version of MCTS. In later editions of the competition the participating agents showed a wider variety of approaches. Other than using tree search and MCTS, like MaastCTS2 (Soemers *et al.*, 2016), the winner of the 2016 edition of the competition, agents started using techniques based on evolutionary approaches and on hyper-heuristics. An example of the latter category is YOLOBOT (Joppen *et al.*, 2018), which won several editions of the competition. This agent implements heuristic Best-First Search for deterministic games and MCTS for stochastic games.

3.2.4 GVG-AI Agents

The agents used in this thesis have all been implemented in the GVG-AI framework, therefore in Java. To play single-player video games, agents have to extend the *AbstractPlayer* class in the framework and provide the implementation of the following two public methods:

- **Agent(StateObservation so, ElapsedCpuTimer elapsedTimer)** a constructor for the agent that takes as input the observation of the initial state containing the game model, and a timer that tells the agent how much time is available to complete the execution of the constructor. The agent can use this time to prepare to play the game.
- **Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer):** a method in which the agent has to select an action for the controlled avatar, given the observation of the current state and the timer that enables the agent to know how much time it has available to complete the execution of this method. This method is called once per tick to request to the agent the next move that the avatar has to play.

Using the StateObservation the agent can query the game model and retrieve information about the game. For example, it can request the current game tick, the current score and whether the state is terminal, and if so, who the winner is. Moreover, the agent can request information about the avatar it is controlling (e.g. its legal actions, its position, its speed, etc.), about other objects in the game and about all the collisions that have happened so far.

For the experiments performed in this thesis on the single-player planning track of the GVG-AI competition, the following two agents are considered: sampleMCTS and MaastCTS2. Experiments focus mainly on MaastCTS2, while sampleMCTS has been used for an experiment that supports the discussion on on-line parameter tuning presented in Subsection 6.4.9. The characteristics of these agents are reported below.

sampleMCTS. This agent is already provided in the GVG-AI framework. It implements "open-loop" MCTS with the UCT selection strategy and a random play-out strategy. This agent uses the heuristic in Formula 3.1 to evaluate a game state s, and update the statistics in the nodes.

$$Value(s) = \begin{cases} score(s) - 10\,000\,000 & \text{if the agent loses in } s \\ score(s) + 10\,000\,000 & \text{if the agent wins in } s \\ score(s) & \text{otherwise.} \end{cases} \quad (3.1)$$

When used to compute the UCT value of an action in a node, the value computed by the heuristic is rescaled in the interval $[0, 1]$.

MaastCTS2. This is the winning agent of the Single-Player Planning track of the 2016 GVG-AI Competition (Soemers *et al.*, 2016). This agent uses an "open-loop" implementation of MCTS extended with the following enhancements:

- Tree Reuse in between game ticks, for which tree statistics are decayed with a factor $\gamma = 0.6$.
- PH as selection strategy enhancement, with exploration constant $C = 0.6$ and bias $W = 1$.
- NST to enhance the play-out strategy, with $\epsilon = 0.5$, $fpu_{NST} = max$, where max is the maximum score obtained so far in the game, maximum N-Gram length $L = 3$ and the minimum number of times that an N-Gram has to be visited in order to be included in the computation of the value of an action $N = 7$. When using NST, MaastCTS2 takes the position of the avatar into account as well, because the performance of an action in GVG-AI usually depends on the position of the avatar when the action is played. Therefore, NST statistics are updated for N-Grams of $(action, avatar_position)$ pairs instead of N-Grams of actions only.
- *Breadth-First Tree Initialization* before the start of MCTS, which consists in a repeated 1-ply Breadth-First Search (BFS) to estimate the value of all the root successors. Moreover, *Safety Prepruning* is performed, which means that only the successors of the root that had the least number of losses when visited with BFS are kept.

- *Loss Avoidance*, which tries to ignore losses by immediately searching for a better alternative whenever a loss is encountered the first time a node is visited. This enhancement improves the performance on games where many losses are causing MCTS to underestimate the values of nodes, even if they could be easily avoided.
- *Novelty-Based Pruning*, an enhancement that uses the idea of *novelty tests* introduced by Geffner and Geffner (2015) for the Iterated Width algorithm. This idea is applied to MCTS to only keep novel nodes and prune the ones that lead to redundant paths in the search. Note that, whether a node is novel or not might depend on the particular state that has been generated by the non-deterministic game model. Therefore, when reusing the tree in between game ticks, nodes that were pruned in the previous step are unpruned and they are checked for novelty again.
- *Knowledge-Based Evaluations* for the states. This enhancement is inspired by the work of Perez-Liebana, Samothrakis, and Lucas (2014) and consists in using domain knowledge to generate a heuristic evaluation function for non-terminal states. This function evaluates a state depending on the position of certain objects in it, rewarding the agent for exploring the effects of interacting with the objects and subsequently moving closer to objects that influence the score positively and away from objects that influence the score negatively.
- *Deterministic Game Detection*, a mechanism that is implemented by many GVG-AI successful agents, like RETURN42 (Perez-Liebana *et al.*, 2018) and YOLOBOT (Joppen *et al.*, 2018). When this mechanism detects a game that is likely to be deterministic, some aspects of the search are modified to better deal with this type of games. More precisely, in MAASTCTS2 the term $\bar{q}_{(s,a)}$ in the PH formula (Formula 2.4) is substituted by $\frac{3}{4} \times \bar{q}_{(s,a)} + \frac{1}{4} \times qMax_{(s,a)}$, where $qMax_{(s,a)}$ is the maximum score observed so far in the subtree rooted in the node reached by performing a in s. Moreover, tree statistics are not decayed anymore when reusing the tree and nodes pruned after testing their novelty are not unpruned when the game tick changes.

3.3 Discussion

This chapter introduced the environments used in this thesis to test the application of MCTS on general game playing: the Stanford GGP project and the GVG-AI project. Although these environments share some common characteristics, like the assumption of no prior or game-specific knowledge for the agents, the necessity of on-line learning and a limited amount of time to select a move for each game turn, they offer different challenges to MCTS. The Stanford GGP project focuses on turn-based, perfect information games, while GVG-AI introduces the challenge of non-determinism and real-time decisions.

MCTS is suitable to tackle all these challenges because of its characteristic of being aheuristic, anytime and selective (see Section 1.4), and it has already been

shown to be promising for GGP. However, to further improve its performance in these environments, the following three aspects of the search are worth considering:

Search speed. The performance of MCTS is directly influenced by the number of simulations that it is able to perform to collect its samples (Robilliard *et al.*, 2014). The higher the number of performed simulations, the more accurate the collected statistics are. Given that the choices of MCTS are based on such statistics, more accurate estimates lead the agent to play better actions in the game. This holds true both when MCTS is applied to the games provided by the Stanford GGP project and the games provided by the GVG-AI project. In both environments, the number of simulations that an agent can perform in a fixed amount of time depends on how fast the agent can reason on the game. Reasoning on the game is achieved by constructing a game model that implements the computation of the players' legal moves and goals in a state, the computation of state terminality and of state transitions. In GVG-AI the implementation of the game model is provided by the framework, therefore improving the reasoning speed of the agent is out of the control of the programmer. For the Stanford GGP competition, instead, it is up to the agent to construct the game model from the GDL game rules and implement a component, the reasoner, with a mechanism to reason on them. It is worth investigating how the reasoner can be improved because an efficient representation of the game model can substantially speed up the search. Chapter 4 of this thesis deals with this issue.

Search strategies. The performance of MCTS also depends on how the visited actions are chosen during the simulation, both in the selection and in the play-out phase of the search. Enhancing the selection and play-out strategies using further information acquired on-line improves the accuracy with which actions are selected during a simulation. Therefore, MCTS can converge to optimal values within a smaller number of simulations. Given the short amount of time they have available to perform the search, it is worth investigating more informed strategies for the MCTS agents, and analyzing how the information should be collected by such strategies. Chapter 5 of this thesis investigates how collecting information globally or locally with respect to the state being visited influences the selection strategy of MCTS in the Stanford GGP environment.

Search adaptation. Equally important is to detect if and how the acquired information should be exploited during the game. In GGP, agents have to face games with different characteristics, for which the information exploited by a certain search strategy might be more or less relevant, also depending on the current point of the search. Moreover, different search strategies might perform differently on different games. It is thus worth investigating how the search can be adapted to each new game being played, modifying the way information is exploited during the search, or even deciding which search enhancements to (de)activate depending on the game. In both the environments considered in this thesis, agents have to deal with heterogeneous sets of games, without knowing them in advance. Therefore, it might be worth devising techniques

to adapt the search to each new game on-line. This is investigated in Chapter 6 of this thesis.

The discussed aspects can be seen as interdependent. On the one hand, incrementing the simulation speed can improve the performance of the search strategies or give more time to adapt the search strategy to the game. On the other hand, a more informed search strategy that possibly adapts to the played game can compensate for a slower reasoning process.

Chapter 4

Optimizing Propositional Networks

This chapter is based on:

- Sironi, Chiara F., and Winands, Mark H.M. (2017). Optimizing Propositional Networks. *Computer Games*, Vol. 705 of *CCIS*, pp. 133–151. Springer.
- Siwek, Cezary and Kowalski, Jakub and Sironi, Chiara F. and Winands, Mark H.M. (2018). Implementing Propositional Networks on FPGA. *AI 2018: Advances in Artificial Intelligence* (eds. T. Mitrovic, B. Xue, and X. Li), Vol.11320 of *LNCS*, pp. 133–145, Springer.

General game-playing agents that deal with games written in GDL require a reasoner to be able to interpret the game rules and search for the best actions to play in the game. One method for interpreting the game rules consists of translating the GDL game description into an alternative representation that the player can use to reason more efficiently on the game. The Propositional Network (PropNet) (Cox *et al.*, 2009; Schkufza *et al.*, 2008; Genesereth and Thielscher, 2014) is an example of such method. The use of PropNets for GDL games can speed up the reasoning process by several orders of magnitude compared to custom-made or Prolog-based GDL reasoners, improving the quality of the search for the best actions. This holds true also for MCTS, which, with a faster GDL reasoner, could perform a higher number of simulations and improve its performance (Robilliard *et al.*, 2014). This chapter answers the first research question by analyzing the performance of a PropNet-based reasoner and evaluates four different optimizations for the PropNet structure that can help further increase its reasoning speed in terms of visited game states per second. The chapter also discusses research on implementing a reasoner that encodes the PropNet on a Field-Programmable Gate Array (FPGA), which has the potential of achieving an even higher speed up of the reasoning process.

This chapter is structured as follows. First, Section 4.1 introduces the PropNets. Next, Section 4.2 gives details about the implementation of the PropNet and of the

reasoner based on it that is tested in this chapter. Subsequently, Section 4.3 presents the empirical evaluation of the PropNet reasoner and Section 4.4 discusses research on the implementation of a reasoner that encodes the PropNet on an FPGA. Finally, Section 4.5 concludes the chapter and discusses possible future research.

4.1 Background

Many different approaches have been proposed to parse the GDL game rules. Three main methods to interpret GDL can be identified (Schiffel and Björnsson, 2014): (1) Prolog-based interpreters, that translate the game rules from GDL into Prolog and then use a Prolog engine to reason on them, (2) custom-made interpreters written for the sole purpose of interpreting GDL rules, and (3) reasoners that translate the GDL description into an alternative representation that the player can use to efficiently reason on the game. A description and performance evaluation of available GDL reasoners is given by Schiffel and Björnsson (2014).

A faster GDL reasoner, which in a given amount of time can analyze a higher number of game states than other reasoners, can positively influence Monte-Carlo based search. PropNets (Cox *et al.*, 2009; Schkufza *et al.*, 2008; Genesereth and Thielscher, 2014) have become popular in GGP because they can speed up the reasoning process by several orders of magnitude compared to custom-made or Prolog-based GDL reasoners. Since 2011, most of the best agents that participated in the Stanford GGP competition used a PropNet-based reasoner (Draper and Rose, 2014; Schreiber and Landau, 2016).

In the next subsection the structure of a PropNet is described, and an example on how a PropNet can be built from a GDL game description is given.

4.1.1 Propositional Networks

A PropNet (Schkufza *et al.*, 2008; Cox *et al.*, 2009; Genesereth and Thielscher, 2014) can be seen as a graph representation of GDL. A PropNet is a directed graph where each component represents either a GDL proposition or a connective, and can assume a truth value, *true* or *false*). Each component has incoming arcs from its input components and outgoing arcs to its output components. The truth value of a component depends on the truth value of its inputs and is propagated to its outputs. There are four types of connectives: *and*, *or* and *not* logic gates, and *transitions*, identity gates that output their input value with one step delay. Propositions in the PropNet can be divided into three categories: *input*, that have no input components, *base*, that have one single *transition* as input, and all other propositions, identified as *view*. The truth values of *base* propositions represent the state of the game. Their input, the *transition*, controls their value for the next state. Having no inputs, *input* proposition have their value set by the game-playing agent, which sets to true the ones corresponding to the actions it decides to simulate for each player in the game. *View* propositions express players' goals and legal moves, and terminality of game states. A unique truth assignment to *base* propositions in the PropNet determines the unique truth values of *view* propositions. The combination of truth assignments

```
(role player)
(light p) (light q)
(<= (base (on ?x)) (light ?x))
(<= (input player (turnOn ?x)) (light ?x))
(<= (legal player (turnOn ?x)) (not (true (on ?x))) (light ?x))
(<= (next (on ?x)) (does player (turnOn ?x)))
(<= (next (on ?x)) (true (on ?x)))
(<= terminal (true (on p)) (true (on q)))
(<= (goal player 100) (true (on p)) (true (on q)))
```

Figure 4.1: GDL game description for a simple game.

```
(role player)
(light p) (light q)
(base (on p))
(base (on q))
(input player (turnOn p))
(input player (turnOn q))
(<= (legal player (turnOn p)) (not (true (on p))))
(<= (legal player (turnOn q)) (not (true (on q))))
(<= (next (on p)) (does player (turnOn p)))
(<= (next (on p)) (true (on p)))
(<= (next (on q)) (does player (turnOn q)))
(<= (next (on q)) (true (on q)))
(<= terminal (true (on p)) (true (on q)))
(<= (goal player 100) (true (on p)) (true (on q)))
```

Figure 4.2: Grounded GDL game description for a simple game.

to *base* and *input* proposition uniquely determines the truth assignment for the next state.

Each GDL game description can be translated into a PropNet with the same dynamics. In order to do so, the game description has to be *grounded* first, which means being transformed into an equivalent description that does not contain any variable. For example, Figure 4.1 shows the GDL description of a simple game, where a player can independently turn on two lights (p and q). After being turned on, each light will remain on. The game ends when both lights are on and in this state the player achieves a goal with score 100. Figure 4.2 shows the grounded version of this game description. More details about grounding game descriptions are given by Schiffel (2017). Once the grounded description has been computed, the PropNet propositions can be created and linked using connectives as specified by the GDL rules. Figure 4.3 shows the PropNet that can be built using the given grounded description. Base propositions, which are used to represent the game state, are identified from the *base* relation, while input propositions, which are used to represent that a player is performing an action, are identified from the *input* relation. All other propositions

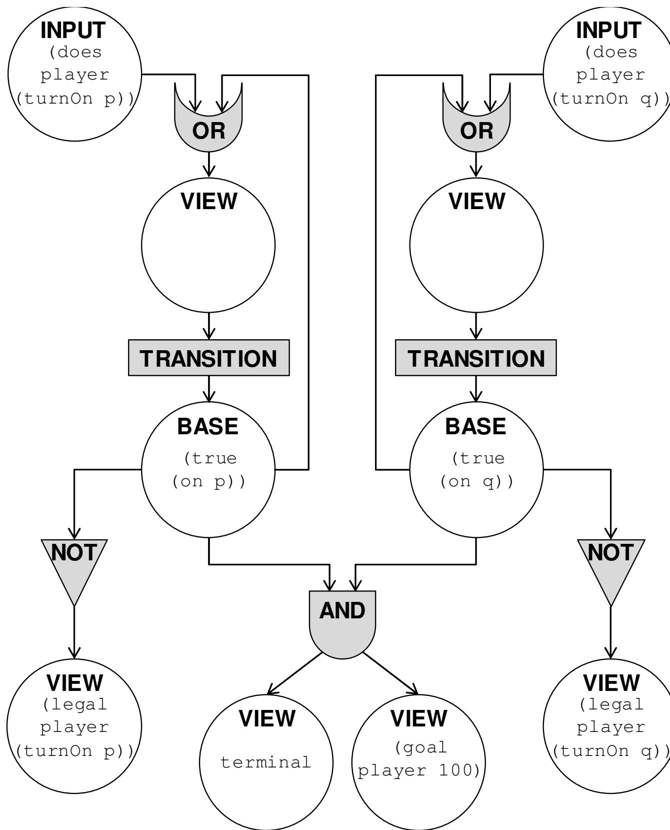


Figure 4.3: PropNet structure example.

are identified from the rules in the grounded game description and are integrated in the PropNet accordingly. For example, from the rule `(<= (legal player (turnOn p)) (not (true (on p))))` the proposition `(legal player (turnOn p))` is identified. Its truth value depends on the negation of the base proposition `(true (on p))`, therefore in the PropNet it is connected to such proposition with a *not* gate in-between. Rules with the `next` relation are treated differently. A proposition identified by the `next` relation is not added to the PropNet and the conditions that make it true are modeled in the PropNet and connected to the transition that leads to the corresponding `base` proposition. For example, the proposition `(next (on p)))` is true if either `(does player (turnOn p))` or `(true (on p))` are true, therefore the outputs of such propositions are connected to an *or* gate, which is connected to the transition that leads to the base proposition `(true (on p))`. The empty view proposition in the figure is created to collect the output of the *or* gate.

4.2 PropNet Implementation

In this section, the implementation of the tested PropNet is discussed. First, Subsection 4.2.1 describes how the PropNet built from the GDL game description is initialized. How the PropNet is optimized is described in Subsection 4.2.2, and Subsection 4.2.3 explains how the PropNet is used by the reasoner.

4.2.1 Initialization

The PropNet tested in this chapter is implemented for the agents developed in the GGP Base Package (Schreiber and Landau, 2016). Given a GDL game description, to create the PropNet the algorithm provided in the GGP Base Package is used.[1] This algorithm is implemented in the CREATE(List<Gdl> description) method of the *OptimizingPropNetFactory* class and builds the PropNet according to the rules in the given GDL description.

The final product of the algorithm is a set of all the components in the PropNet, each of which has been connected to its input and output components. This set can then be used to initialize a PropNet object. The algorithm distinguishes six different types of components: *constants* (TRUE and FALSE), *propositions*, *transitions* and three different *gates* (AND, OR, NOT).

The GGP Base Package also provides a PropNet class that can be initialized using the created set of components. We use this class as a starting point and implement some changes to the initialization process to ensure that the PropNet respects certain constraints that are needed for the optimizations algorithms to work consistently. The first step of the initialization iterates over all the components in the PropNet and inserts them in different lists according to their type. While iterating over all the components, the following are the main actions that the initialization algorithm performs:

- Identify a single TRUE and a single FALSE constant, creating them if they do not exist, or removing the redundant ones.
- Identify the type of each proposition. Each proposition must be associated to one type only. A proposition that has a *transition* as input is identified as BASE type and a proposition that corresponds to a GDL relation with the *does* keyword is identified as INPUT type. The propositions corresponding to GDL relations with the *legal*, *goal* or *terminal* keyword are identified as LEGAL, GOAL and TERMINAL type, respectively. To all other propositions the type OTHER is assigned.
- Ensure that all the INPUT and LEGAL propositions are in a 1-to-1 relation. If a proposition is detected as being an INPUT but there is no corresponding LEGAL in the PropNet, then it can be removed since we are sure that the corresponding move will never be chosen by the player. On the contrary, if there is a LEGAL proposition with no corresponding INPUT, the INPUT

[1] The version used in this thesis is more recent and improved with respect to the one tested by Schiffel and Björnsson (2014).

1: **procedure** OPT$_0$(*propnet*)
Input: The structure of the PropNet, *propnet*.
2: $\mathcal{O}_T \leftarrow propnet.TRUE.outputs$
3: $\mathcal{O}_F \leftarrow propnet.FALSE.outputs$
4: **while** $\mathcal{O}_T \neq \varnothing$ **or** $\mathcal{O}_F \neq \varnothing$ **do**
5: REMOVEFROMTRUE($propnet, \mathcal{O}_T, \mathcal{O}_F$)
6: REMOVEFROMFALSE($propnet, \mathcal{O}_T, \mathcal{O}_F$)

Algorithm 4: Remove constant-value components.

proposition is added to the PropNet, since the LEGAL proposition might become true at a certain point of the game and the player might choose to play the corresponding move.

- Ensure that only constants (i.e. TRUE and FALSE) and INPUT propositions have no input components. If a different component is detected as having no inputs, set one of the two constants as its input. This action is needed because as a by-product of the PropNet creation some OR gates and non-INPUT propositions might have no inputs. The behavior of the PropNet has been empirically tested to be consistent when such components are connected to the FALSE constant.

4.2.2 Optimizations

The PropNets built by the algorithm given in the GGP Base Package contain usually many components that are not strictly necessary to reason about the game. This subsection presents four optimizations (Opt0, Opt1, Opt2 and Opt3) that can be performed on the PropNet structure to reduce the number of these components. Opt0 removes components that are known to have a constant truth value, Opt1 removes propositions that do not have a particular meaning for the game, Opt2 detects more components with a constant truth value and removes them, and Opt3 removes components that have no output and are not influential. The algorithms that perform such optimizations have been suggested by Landau (Schreiber and Landau, 2016). However, they have never been thoroughly investigated. The PropNet optimization algorithms described in this thesis contain some minor modifications with respect to the original GGP Base Package version in order to adapt them to the changes that were performed on the PropNet class structure.

Opt0: Remove Constant-value Components

This optimization removes from the PropNet the components that are known to be always *true* or always *false* and at the same time do not have a particular meaning for the game. For example, an AND gate that has an input that is always *false* will also always output *false*, thus the gate can be removed and all its outputs can be connected directly to the FALSE constant of the PropNet.

Algorithm 4 shows the main steps of Opt0. The sets $\mathcal{O}_T$ and $\mathcal{O}_F$, at any moment, contain, respectively, the outputs of the TRUE and the outputs of the FALSE

```
 1: procedure RemoveFromTrue(propnet, O_T, O_F)
    Input: The structure of the PropNet, propnet, the set of output components
    of the TRUE constant that still have to be checked for removal, O_T, the set
    of output components of the FALSE constant that still have to be checked for
    removal, O_F.
 2:   while O_T ≠ ∅ do
 3:     c ← O_T.RemoveElement()
 4:     switch c.compType do
 5:       case TRANSITION
 6:         if |c.outputs| = 0 then
 7:           propnet.Remove(c)
 8:       case NOT
 9:         connect c.outputs to FALSE
10:         O_F ← O_F ∪ c.outputs
11:         propnet.Remove(c)
12:       case AND
13:         if |c.inputs| = 1 then                       ▷ Only TRUE as input
14:           connect c.outputs to TRUE
15:           O_T ← O_T ∪ c.outputs
16:           propnet.Remove(c)
17:         else if |c.inputs| = 2 then       ▷ Only 2 inputs, one is TRUE
18:           connect c.outputs to other input
19:           propnet.Remove(c)
20:         else                        ▷ More than 2 inputs, one is TRUE
21:           disconnect c from TRUE
22:       case OR
23:         connect c.outputs to TRUE
24:         O_T ← O_T ∪ c.outputs
25:         propnet.Remove(c)
26:       case PROPOSITION
27:         connect c.outputs to TRUE
28:         O_T ← O_T ∪ c.outputs
29:         if c.propType ∈{OTHER, BASE} then
30:           propnet.Remove(c)
31:     end switch
```

Algorithm 5: Remove true components.

constant that still have to be checked for removal. At the beginning $\mathcal{O}_T$ contains all the outputs of the TRUE constant and $\mathcal{O}_F$ contains all the outputs of the FALSE constant (Lines 2 and 3).

The procedure RemoveFromTrue($propnet, \mathcal{O}_T, \mathcal{O}_F$) (Line 5) and the procedure RemoveFromFalse($propnet, \mathcal{O}_T, \mathcal{O}_F$) (Lines 6) check the outputs of the TRUE and of the FALSE constant, respectively. Algorithm 5 shows exactly which components the first procedure removes. The algorithm for the second procedure removes

```
1: procedure OPT1(propnet)
   Input: The structure of the PropNet, propnet.
2:     for all c ∈ propnet.propositions do
3:         if c.propType = OTHER then
4:             connect c.input with c.outputs
5:             propnet.REMOVE(c)
```

Algorithm 6: Remove anonymous propositions.

the outputs of the FALSE constant in a similar way. In the case of the FALSE constant, also always false GOAL and LEGAL propositions are removed since they will never be used. Moreover, whenever a LEGAL proposition is removed also the corresponding INPUT proposition is removed, since it is certain that the corresponding move will never be played. Note that whenever a component is removed or detected as having always a constant value, it means that also its output is constant, thus its output components are connected directly to one of the two constants. In this case each output component will be added to the appropriate set (either $\mathcal{O}_T$ or $\mathcal{O}_F$) to be checked in the next steps.

Algorithm 4 alternates between the two procedures mentioned above until both sets, $\mathcal{O}_T$ and $\mathcal{O}_F$, are empty. This repetition is needed because of the NOT gate. Whenever this gate is removed from the outputs of a constant, its outputs are connected to the other constant, thus the set of outputs to be checked for that constant will still have at least one element.

Examples of how Opt0 changes the structure of the PropNet are given in Figure 4.4. Double circles are used to represent the TRUE and FALSE constants. The first six examples clearly show the advantage of this optimization, as it removes components from the PropNet. In the last two examples no components are removed from the PropNet. However, after applying Opt0, two components become direct outputs of a constant. Thus, they might be removed in the next iteration of the algorithm or their outputs (if any) will be also analyzed by Opt0.

Opt1: Remove Anonymous Propositions

This optimization is trivial, nevertheless useful as it removes many useless components from the PropNet. When the PropNet is created, many proposition without a special meaning for the game are created to collect the outputs of various combinations of components. Once the PropNet has been created, these propositions can be removed. During initialization such propositions are identified as being of type OTHER, therefore the algorithm for Opt1 (Algorithm 6) simply iterates over all the propositions in the PropNet and removes the ones with this type, connecting their input directly to each of their outputs. For example, Figure 4.5 shows the PropNet presented in Figure 4.3 with the type of all the propositions identified. The figure presents the structure of the PropNet before and after performing Opt1.

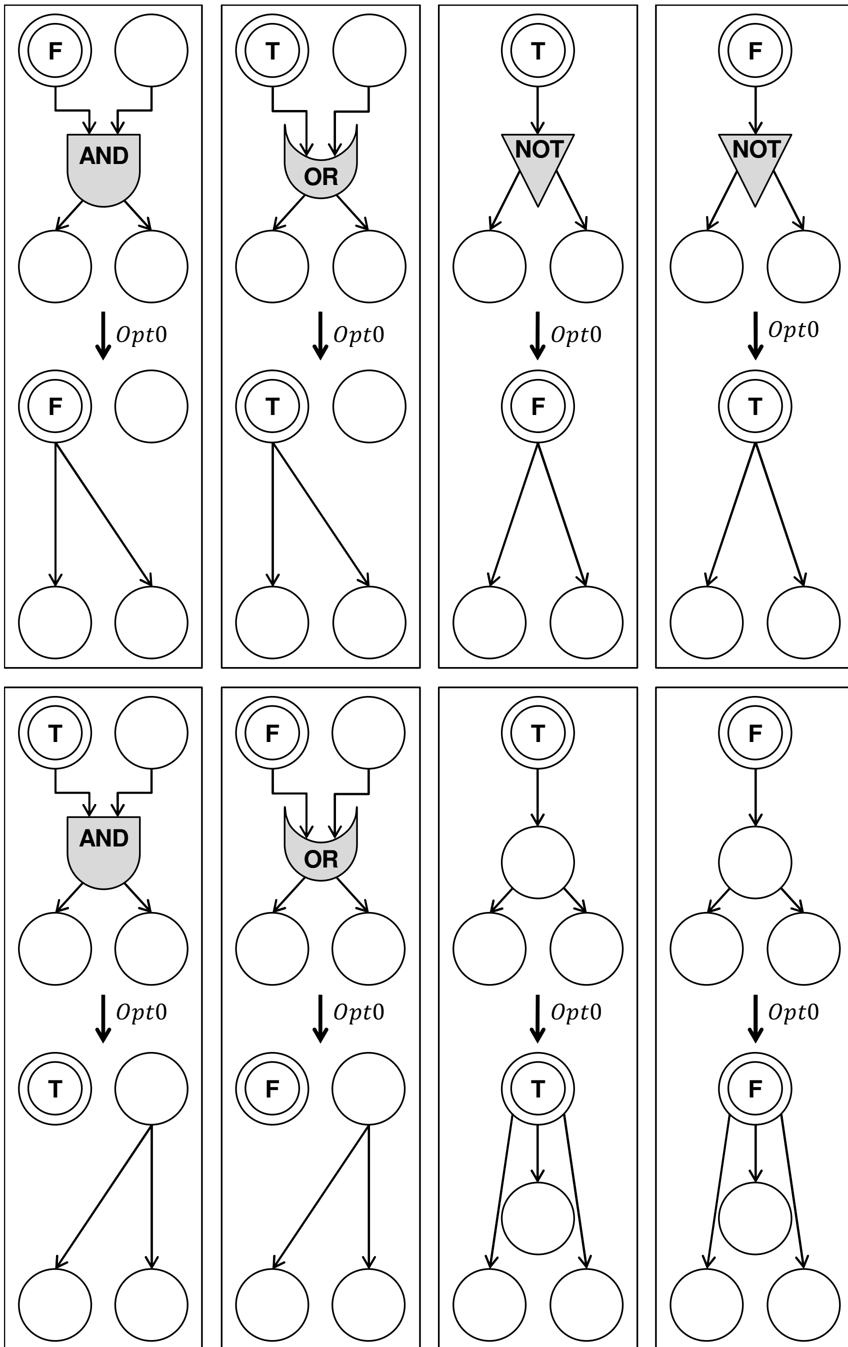


Figure 4.4: Examples of changes to the PropNet structure after applying Opt0.

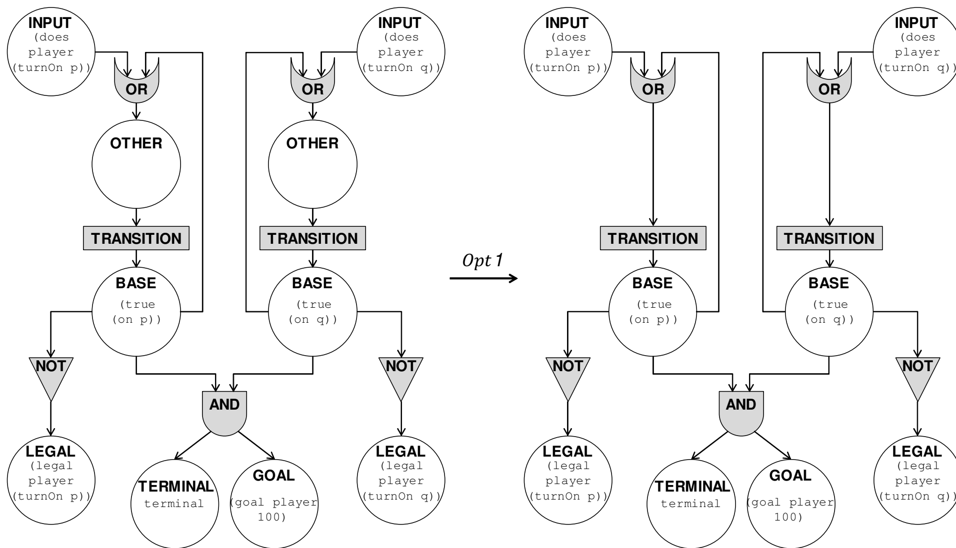


Figure 4.5: Changes to the PropNet structure after applying Opt1.

Opt2: Detect and Remove Constant-value Components

This optimization can be seen as an extension of Opt0 where, before removing from the PropNet the constant value components directly connected to the *TRUE* and *FALSE* constant, the algorithm detects if there are other constant value components that have not been discovered yet.

The pseudocode for this optimization is given in Algorithm 7, while Figure 4.6 gives an example of how this optimization works for the PropNet introduced in Figure 4.3. Note that, in order to have a visible effect of Opt2 on the PropNet, the example assumes that the proposition `(on p)` is true in the initial state.

Opt2 associates to each component c in the PropNet a set $\mathcal{W}_c$ that contains all the truth values that such component can assume during the whole game. There are only four possible sets of truth values, namely:

- $\mathcal{N} = \varnothing$: if the corresponding component can assume *neither* of the truth values.
- $\mathcal{T} = \{true\}$: if the corresponding component can only be *true* during all the game.
- $\mathcal{F} = \{false\}$: if the corresponding component can only be *false* during all the game.
- $\mathcal{B} = \{true, false\}$: if the corresponding component can assume *both* values during the game.

The idea behind the algorithm is to start from the components for which the truth value in the initial state of the game is known. Then, for each of these components c,

```
 1: procedure Opt2(propnet)
    Input: The structure of the PropNet, propnet.
 2:     Initialize all the parameters and the stack
 3:     while stack ≠ ∅ do
 4:         (c, 𝒳_{c_in}) ← stack.Pop()
 5:         𝒴_c ← ToOutputValueSet(c, 𝒳_{c_in})
 6:         𝒳_c ← 𝒴_c \ 𝒲_c
 7:         if 𝒳_c ≠ 𝒩 then
 8:             𝒲_c ← 𝒲_c ∪ 𝒳_c
 9:             for all c_out ∈ c.outputs do
10:                 stack.Push(c_out, 𝒳_c)
11:             if c.compType = PROPOSITION and c.propType = LEGAL then
12:                 i ← c.correspondingInputProposition
13:                 stack.Push(i, 𝒳_c)
14:     for all c ∈ propnet.components do
15:         if 𝒲_c = 𝒯 or 𝒲_c = ℱ then
16:             Connect c to the appropriate constant
17:     Opt0(propnet)
```

Algorithm 7: Detect and remove constant-value components.

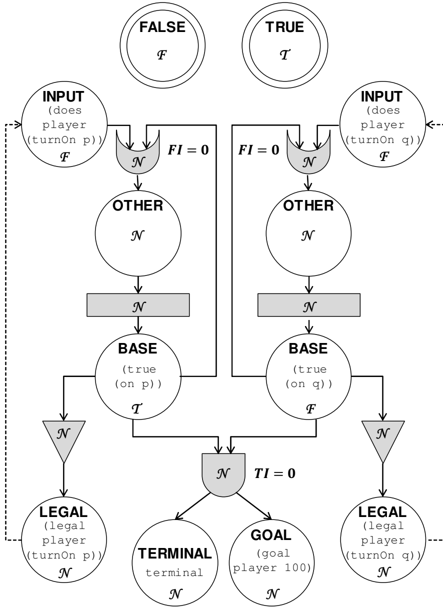
the algorithm propagates the initial value to each of their outputs c_{out}, updating the corresponding truth value set $\mathcal{W}_{c_{out}}$. Whenever the truth value set of a component is updated, the algorithm propagates such changes on to its output components. This process will eventually end when the truth values sets of all components stop changing. Termination is guaranteed since only the truth values just added to the truth value set of a component are propagated to its outputs and the number of possible truth values is finite.

When the algorithm starts, the set $\mathcal{W}_c$ of each component c is set to $\mathcal{N}$, since it is not known yet which values the component can assume. For each AND gate a the algorithm keeps track of TI_a, i.e. the number of inputs of a that can assume the *true* value. Similarly, for each OR gate o the algorithm keeps track of FI_o, i.e. the number of inputs of o that can assume the *false* value. This parameters are used to detect when an AND gate and an OR gate can assume respectively the *true* (if $TI_a = |a.inputs|$) and the *false* (if $FI_o = |o.inputs|$) value. These values are initialized to 0 for all the gates.

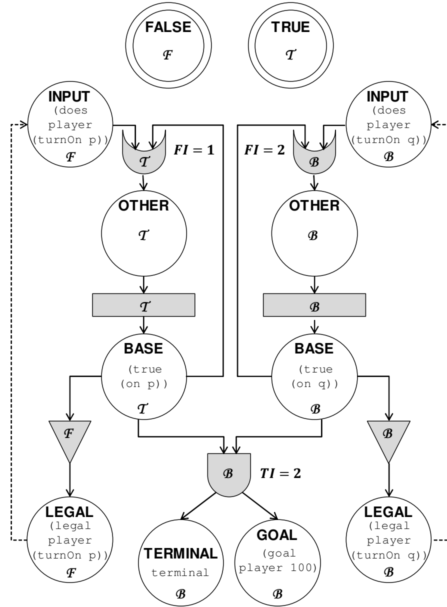
The algorithm exploits a stack structure *stack* to keep track of the components for which the set of truth values that their input(s) can assume is changed. A pair $(c, \mathcal{X}_{c_{in}})$ is added to the stack when the algorithm detects that an input c_{in} of the component c can also assume the values in the set $\mathcal{X}_{c_{in}} \subseteq \mathcal{W}_{c_{in}}$, and such values must be propagated to the component c. At the beginning the stack is filled with the following pairs:

- $(\text{TRUE}, \mathcal{T})$. The TRUE constant can assume value *true*.
- $(\text{FALSE}, \mathcal{F})$. The FALSE constant can assume value *false*.

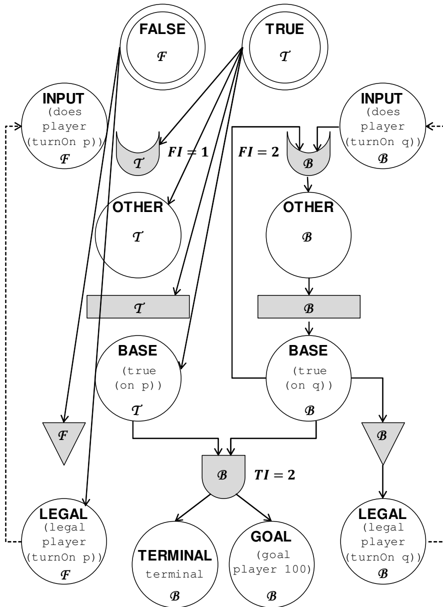
1. Initialization:



2. Result of value propagation:



3. Connection to constants



4. Result after performing $Opt1$:

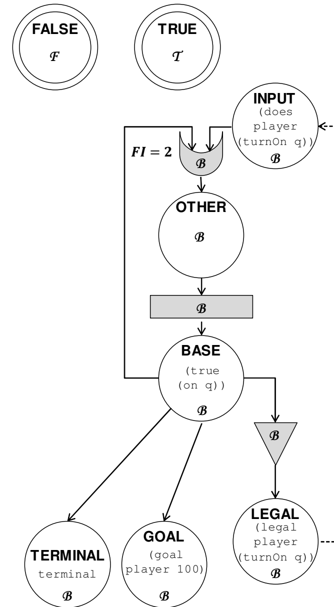


Figure 4.6: Examples of changes to the PropNet structure after applying Opt2.

- $(i, \mathcal{F})$, for each INPUT proposition i in the PropNet. Each INPUT proposition can be *false* since we assume that no game exists where one player can only play a single move for the whole game.

- $(b, \mathcal{T})$, for each BASE proposition b in the PropNet that is *true* in the initial state.

- $(b, \mathcal{F})$, for each BASE proposition b in the PropNet that is *false* in the initial state.

The upper left part of Figure 4.6 shows the initialization of the sets of truth values of each component and of the counters for the AND and OR gates for the considered PropNet.

During each iteration, the algorithm pops a pair $(c, \mathcal{X}_{c_{in}})$ from the stack (Line 4) and checks if, given the new truth values $\mathcal{X}_{c_{in}}$ that the input c_{in} can assume, also the truth values $\mathcal{W}_c$ of its output c will change. Note that not for each type of component the set of truth values that its input can assume corresponds to the set of truth values that the component itself can output. The NOT component n, for example, has $\mathcal{W}_n = \mathcal{T}$ if its input n_{in} has $\mathcal{W}_{n_{in}} = \mathcal{F}$. Moreover, for an AND gate a, $(true \in \mathcal{W}_a \Leftrightarrow \forall a_{in} \in a.inputs, true \in \mathcal{W}_{a_{in}})$. The same holds for the $false$ value for an OR gate. This means that the algorithm hast to change first the values in $\mathcal{X}_{c_{in}}$ according to the type of the component c, obtaining the new set of truth values $\mathcal{Y}_c$ that c can output. This is performed at Line 5 by the function ToOutputValueSet$(c, \mathcal{X}_{c_{in}})$. Subsequently, the algorithm checks if in $\mathcal{Y}_c$ there are some values $\mathcal{X}_c$ that were not in $\mathcal{W}_c$ yet (Line 6), and if so, it adds them to the set $\mathcal{W}_c$ (Line 8) and records on the stack that they have to be propagated to all the outputs c_{out} of c (Lines 9-10). Here the algorithm treats each LEGAL propositions as if it was a direct input of the corresponding INPUT proposition, thus whenever the truth values set of a LEGAL proposition changes, the values are propagated to the corresponding INPUT proposition (Lines 11-13). The second part of Figure 4.6 shows the result of applying the just described propagation method on the PropNet used in the example.

When no more changes are detected in the truth values sets (Line 3), the process terminates. At this point, the truth values set of each component is checked (Line 15) and if it equals the set $\mathcal{T}$ or $\mathcal{F}$ it is certain that the component will always be respectively *true* or *false*. It can then be disconnected from its input(s) and connected to the correct constant (Line 16). This is shown in the third part of Figure 4.6 for the PropNet considered in the example. The last step that the algorithm performs consists in running the same algorithm that was proposed as Opt0 to remove all the newly detected constant components (Line 17). The result of this algorithm on the PropNet considered in the example is shown in the last part of Figure 4.6.

Opt3: Remove Components with no Outputs

This optimization is quite straightforward, but helps remove more useless components. For some games, the algorithm that creates the PropNet leaves some gates

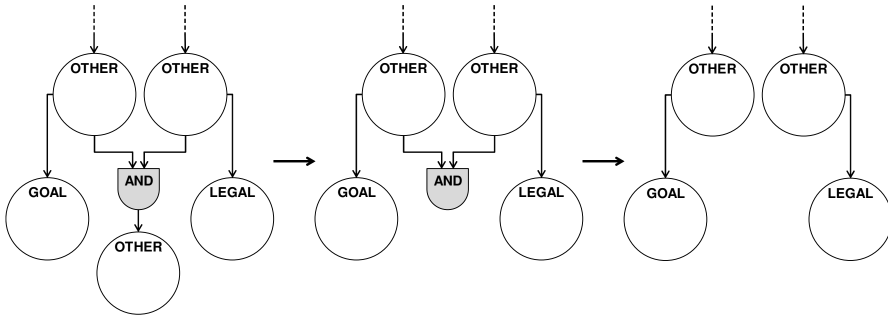


Figure 4.7: Changes to the PropNet structure when applying Opt3.

```
1: procedure Opt3(propnet)
   Input: The structure of the PropNet, propnet.
2:     C ← propnet.components
3:     while C ≠ ∅ do
4:         c ← C.RemoveElement()
5:         if ((c.compType = PROPOSITION and c.propType = OTHER)
               or c.compType ∈ {AND, OR, NOT}) and |c.outputs| = 0 then
6:             C ← C ∪ c.inputs
7:             propnet.remove(c)
```

Algorithm 8: Remove components with no outputs.

that have no outputs in the structure. These gates can be safely removed by Opt3, because they do not influence the process of reasoning on the game. Other than gates, Opt3 removes also propositions that have no output and are of type OTHER, and therefore have no influence on the game.

Algorithm 8 shows this procedure: all the components in the PropNet are checked, if they are gates or propositions of type OTHER and they have no output they are removed from the PropNet. Every time a component is removed, its inputs are added again to the set of components to be checked, since they might have been left without any output. Figure 4.7 gives an example of how multiple iterations of the loop in Algorithm 8 change the structure of the PropNet. First the OTHER proposition is removed, leaving the AND gate without outputs. Then, this gate is removed in the following iteration.

Note that the propositions of type OTHER could be ignored by Opt3, assuming that Opt1 takes care of removing them. However, in this case Opt3 would become dependent on Opt1, meaning that there would be components that Opt3 could remove only if Opt1 has been performed first. For example, Opt3 would have no effect on the portion of PropNet showed in Figure 4.7 and it could remove the AND gate only after Opt1 has been performed. The choice of removing OTHER propositions as well has been made in order for Opt3 to be self-contained (i.e. the same components are removed by Opt3 independently of when it is performed with respect to other optimization).

4.2.3 PropNet-based Reasoner

The reasoner is the component of the game-playing agent that, given a GDL game description, takes care of parsing it to build the game model. When the PropNet is used as representation of the game model, its creation and optimization is taken care of by the reasoner. Moreover, all the methods that a search algorithm needs to call on the game model in order to reason on the game are implemented as part of this component. The methods provided by the PropNet-based reasoner tested in this thesis are listed below, together with a brief explanation of how they operate on the PropNet to compute the corresponding game information. All of them use the following encoding for the components of the state machine of a game:

- **State:** the state is represented as a sequence of bits, each corresponding to one of the *base* proposition in the PropNet. A bit is set to 1 if the corresponding proposition in the PropNet is true, to 0 otherwise.
- **Action:** an action is represented as the index of the corresponding PropNet proposition in the list of *input* propositions.
- **Role:** a role is represented as its index in the list of roles extracted from the GDL game description.

The following are the methods that the reasoner offers to the agent to reason on the game:

- **s GetInitialState():** this method returns the initial state s of the game in the PropNet encoding. To identify the *base* propositions that are true in the initial state the *init* relation is used. The propositions that are identified by the *init* relation are extracted from the GDL game description and the corresponding *base* propositions have their bit set to true in the state.
- **$\vec{a}$ GetLegalActions(s, i):** given a state s and a player i, this method returns all the actions $\vec{a}$ that are legal for the player in the state. To compute the legal actions for a player, this method first sets the truth values of the *base* propositions according to the values specified by the state s, and then updates the values of all the PropNet components that are influenced by the *base* propositions. Finally, it returns the actions of the player for which the value in the PropNet is set to true.
- **s' GetNextState($s, \vec{a}$):** given a state s and a joint action $\vec{a}$, this method returns the state s' reached by performing such joint action in s. To compute the next state, this method sets the truth values of the base propositions according to the values specified by the state s. Moreover, it sets to true the *input* propositions that correspond to the actions specified in $\vec{a}$ and to false all other *input* propositions. Subsequently, it updates the values of all the components that are influenced by the *base* and input propositions. Finally, it creates a new state where the value of each bit is set to the value of the transition component that inputs in the base proposition corresponding to such bit. Note that at this point the values of the *base* propositions represent the current state, while the values of the *transitions* represent the next state.

- ***boolean*** **TERMINAL(s):** given a state s, this method returns *true* if the state is terminal and *false* otherwise. The procedure to compute state terminality with the PropNet is the same as the one to compute legal actions, except that the method returns the value of the *terminal* proposition after updating the truth values of the PropNet components.

- **$\vec{q}$ PAYOFF(s):** given a state s, this method returns the payoff obtained by each player. The same procedure that is used to compute legal actions and state terminality is applied, but in the end the score identified by the true *goal* propositions is returned for each player.

4.3 Experiments

In this section an empirical evaluation of the performance of the PropNet and its optimizations is presented. Subsection 4.3.1 describes the setup of the performed experiments. Subsections 4.3.2 and 4.3.3 discuss the results of the experiments that compare the performance of the reasoners based on the PropNet with single optimizations and combinations of them respectively. The reasoner based on the combination of PropNet optimizations that performs overall best is then compared with the Prover reasoner provided in the GGP Base Package. Subsection 4.3.4 presents a comparison of the PropNet and Prover reasoners in terms of their speed, while Subsection 4.3.5 presents a comparison in terms of their game-playing performance.

4.3.1 Setup

To evaluate the performance of the PropNet reasoner multiple series of experiments are performed. Each of them tests the performance of the reasoner with different PropNet optimizations and combinations of them. Each series of experiments poses the bases to decide which other combinations of optimizations to check.

To test the different PropNet optimizations and their combinations, the PropNet reasoner is tested performing Flat-MCS on a set of heterogeneous games. For each optimized PropNet the search is run from the initial state of the game with a time limit of 20s. This experiment is repeated 100 times for each of the chosen games. Such games are the following: Amazons, Battle, Breakthrough, Chinese Checkers with 1, 2, 3, 4 and 6 players, Connect Four, Othello, Pentago, Skirmish and Tic Tac Toe. They are the same used by Schiffel and Björnsson (2014) to compare different GDL reasoners, and have been selected for having different complexities and being representative of the type of games commonly used in the Stanford GGP competition. A description of the rules and the relevant properties of these games can be found in Appendix B.1, while their GDL descriptions can be found on the GGP Base repository (Schreiber, 2016).[2]

One of the reasons behind the choice of repeating each experiment multiple times for each game is that for every repetition of the game a different seed is used for the random number generator that controls the random exploration of the search

[2]The GDL descriptions used for the experiments were downloaded from the repository on 03/02/2016.

tree with the MCS algorithm. Thus, for different seeds different results might be obtained and different parts of the search space explored.

Another reason is that the number of components that the PropNet of a game has when created by the basic algorithm (i.e. without optimizations) is not always constant. This variance in the number of components is caused by the non-determinism of the order in which game rules are translated into PropNet components for different runs of the algorithm. This can cause a different grounding order of the GDL description, originating more or fewer propositions and can also cause gates and propositions to be connected in different equivalent orders. The non-determinism within the creation of the PropNet is also the reason why in the experiments performed in this and subsequent chapters, whenever the instances of the agent are using the PropNet to play a game, a new PropNet is generated for each game run. Moreover, the same optimized PropNet structure is used by all the agents taking part in the game run and each of them keeps its own copy of the state of the PropNet (i.e. which truth values are assigned to each component). This guarantees that in a game run none of the agent instances is penalized by a PropNet with a less efficient structure.

The reasoner with the PropNet optimizations that showed the best overall performance in the first series of experiments is compared with the Prover. The Prover is a custom-made interpreter for the GDL rules, which is provided as the standard GDL reasoner in the GGP Base Package. Both reasoners are also tested with the addition of a cache that memorizes previously computed information about the states. More precisely, the cache is represented as a map where each entry corresponds to a game state and maps to an object containing information about such state. This information includes the legal actions and the goals of each player in the state, the reachable next states, and whether the state is terminal. The first time the reasoner is queried to compute any of this information on a state, the result is saved in the corresponding cache entry. Whenever a query is performed the cache is checked first, and the reasoner is queried only if no result was found in the cache.

The series of experiments that compares the PropNet with the Prover matches two Flat-MCS players that use the Prover, one with cache and one without, against each other, and two Flat-MCS players that use the reasoner with the best optimized PropNet, one with cache and one without, against each other. The same 13 games that were used for the other experiments are used. Each player has 10s per move to perform the search. A new PropNet is built for each game run in advance, before the game playing starts. When used by an agent, the cache is empty at the beginning of each new game run, and at the end of each turn the entries that have not been accessed in the last two turns are removed. For each game, if r is the number of roles in the game, there are 2^r different ways in which two types of players can be assigned to the roles (Sturtevant, 2008). Two of the configurations involve only the same player type assigned to all the roles, thus are not interesting and excluded from the experiments. Each configuration is run the same number of times until at least 100 games have been played in total. For a single game run, the speed of a reasoner is computed as the median speed in nodes per second over all the turns. The median is used to prevent the estimated speed from being influenced by likely low values in the initial turns and likely high values in the endgame. For each of the 13 games,

the speed is computed as the average of the median speed over all the runs of the game. Since the focus of this experiment is on the reasoning speed, the 10s search time per move is not considered strictly. This means that, whenever the search time for a turn expires, each player that is still performing a Monte-Carlo simulation is allowed to terminate it before returning a move.

The final series of experiments aims at evaluating the impact of the reasoners on the win rate of a game-playing agent. These experiments match against each other two instances of the MCTS agents implemented in the GGP Base Package (see Subsection 3.1.4). Both instances use the UCT selection strategy with $C = 0.7$ and the random play-out strategy. One of them uses the fastest version of the Prover (i.e. with the cache) and the other uses the fastest optimized PropNet reasoner (also with the cache). The settings are the same as in the previous experiment, except that the minimum number of played games is increased to 200. Moreover, for this experiment the 10s search time per move is considered strictly. Note that these agent instances, being MCTS-based, are using a transposition table to memorize the tree. The instance that uses the PropNet reasoner represents each state in the transposition table as a sequence of bits, while the one that uses the Prover represents each state as the set of GDL propositions that are true in it.

Before running any of the described experiments, the reasoners based on the PropNet and all its optimized versions were tested against the Prover for consistency. For each game (301 at the time of download) in the GGP Base repository (Schreiber, 2016), for a duration of 60s, the same random simulations were performed querying both the Prover and the currently tested version of the PropNet reasoner for next states, legal moves, terminality and goals in terminal states. The results returned by the PropNet reasoner were compared with the ones returned by the Prover for consistency. All the PropNet reasoners passed this test on all the games in the repository, except for 12 games for which the PropNet construction could not be completed in the given time.

In all experiments, a limit of 10 minutes was given to the program to build the PropNet. The experiments that compare the speed of PropNet and Prover with and without cache were performed on a Linux server consisting of 48 AMD Opteron 6174 2.2-GHz cores. All other experiments were performed on a Linux server consisting of 64 AMD Opteron 6274 2.2-GHz cores.

4.3.2 Comparison of Single Optimizations

The first series of experiments compares the reasoner based on the basic version of the PropNet (BasicPN) with the reasoners based on each of the previously described optimizations of the PropNet applied singularly (Opt0, Opt1, Opt2, Opt3). Table 4.1 shows the obtained results. For each PropNet variant, for each game the first block of the table gives the average simulation speed of the corresponding reasoner in nodes per second, the second block gives the average number of components of the PropNet and the third block gives the average total initialization time (creation + optimization + state initialization) of the PropNet in milliseconds. The line at the bottom of each block reports the average of the percentage-wise increase of the considered values over the 13 games, relative to the basic version of the PropNet.

Table 4.1: Comparison of single optimizations.

	Game	BasicPN	Opt0	Opt1	Opt2	Opt3
Avg. speed (nodes/second)	Amazons	35.1	41.4	32.7	41	40.2
	Battle	34 957	49 666	37 877	51 257	35 276
	Breakthrough	50 557	50 932	65 518	51 357	51 058
	Chin.Checkers1P	426 374	427 773	550 230	444 671	424 516
	Chin.Checkers2P	125 581	128 623	189 368	128 910	127 519
	Chin.Checkers3P	155 886	157 242	169 352	161 000	159 267
	Chin.Checkers4P	105 766	106 738	127 886	107 153	105 660
	Chin.Checkers6P	119 650	118 547	126 863	113 700	118 783
	Connect Four	110 081	113 484	105 081	112 920	109 672
	Othello	290	1 610	235	1 604	295
	Pentago	76 336	76 786	116 065	76 721	96 782
	Skirmish	5 887	6 022	6 780	6 230	6 151
	Tic Tac Toe	223 403	228 056	248 769	234 915	222 952
	Avg. rel. increase	–	40.59%	15.51%	41.44%	3.95%
Avg. number of components	Amazons	1 497 649	1 254 742	741 874	1 192 364	1 023 913
	Battle	51 197	14 267	36 863	14 262	50 721
	Breakthrough	10 745	10 678	5 933	10 678	10 584
	Chin.Checkers1P	793	785	559	785	789
	Chin.Checkers2P	1 540	1 524	1 179	1 524	1 532
	Chin.Checkers3P	2 411	2 389	1 845	2 236	2 400
	Chin.Checkers4P	3 159	3 119	2 465	2 999	3 133
	Chin.Checkers6P	4 451	4 411	3 473	4 123	4 431
	Connect Four	2 164	2 063	1 724	1 291	2 114
	Othello	1 311 988	274 940	1 033 197	274 940	1 305 515
	Pentago	3 696	3 706	1 470	3 708	2 111
	Skirmish	126 019	124 267	108 171	124 267	78 575
	Tic Tac Toe	312	291	249	291	302
	Avg. rel. increase	–	-14.28%	-29.21%	-18.62%	-9.49%
Avg. total init. time (ms)	Amazons	311 335	313 719	314 455	417 097	315 637
	Battle	5 756	6 027	5 897	6 303	5 869
	Breakthrough	3 989	4 007	4 012	4 358	3 910
	Chin.Checkers1P	2 699	2 651	2 659	2 653	2 707
	Chin.Checkers2P	2 848	2 773	2 810	2 873	2 775
	Chin.Checkers3P	3 162	3 140	3 159	3 251	3 149
	Chin.Checkers4P	3 258	3 261	3 241	3 473	3 244
	Chin.Checkers6P	3 225	3 203	3 204	3 639	3 205
	Connect Four	2 437	2 465	2 456	2 698	2 430
	Othello	35 756	36 486	37 074	39 417	36 544
	Pentago	4 249	4 230	4 278	4 390	4 232
	Skirmish	11 887	11 702	11 664	12 089	11 824
	Tic Tac Toe	1 525	1 529	1 523	1 522	1 508
	Avg. rel. increase	–	0.13%	0.24%	7.69%	-0.19%

Although the main interest is the speed increase that the optimizations induce on the PropNet, the number of PropNet components and the initialization time are also relevant aspects. A low number of components means less memory usage, and a shorter initialization time means more time for metagaming at the beginning of a match (or more chances to avoid timing out when the start clock is short). From the table it seems that for most of the games, as expected, the increase in the simulation speed is related to the decrease in the number of components in the PropNet.

As can be seen, none of the optimizations outperforms the others in speed for all games. Opt0 and Opt2 seem to have the best performance in Amazons, Battle, Connect Four and Othello, while Opt1 performs best in the other games. Opt3 produces the lowest speed increase over all the games. When looking at the initialization time, Opt2 is the one that increases it the most for almost all the games. Another observation is that the performance of Opt2 is overall better than the one of Opt0. This was expected because Opt2 is an extension of Opt0, thus for the same PropNet it always removes at least the same number of components as Opt0.

The speed is used as main criterion to choose which of the four optimizations to use as starting point for further experiments that involve testing combinations of optimizations. If we consider the speed, Opt0 and Opt2 are the ones that, on average, produce the highest increase. However, the high average is due to the considerable relative increase that they produce in Othello. If we consider the optimization that produces the highest speed increase in most of the games, then Opt1 is the most suitable to be selected. Moreover, Opt1 is the optimization that reduces the most the number of components of the PropNet without consistently slowing down the initialization process.

4.3.3 Comparison of Combined Optimizations

In this series of experiments Opt1 is combined with other optimizations applied in sequence. In general, OptXY refers to the PropNet optimization obtained by applying OptX and OptY in sequence. These experiments first compare the combinations of optimizations Opt12, Opt102 and Opt13. The combination Opt10 has been excluded from the test since it is considered less interesting. As previously mentioned, Opt0 always removes a subset of the components that are removed by Opt2, thus Opt10 is expected to perform less than Opt12. However, Opt0 has less negative impact than Opt2 on the total initialization time. Therefore, these experiments include the test of Opt102 to see if the application of Opt0 before Opt2 can speed up the initialization process by having Opt2 run on a smaller PropNet.

The results of this series of experiments can be seen in columns 3, 4 and 5 of Table 4.2. The structure of this table is the same as Table 4.1. The average percentage increase reported in the last line of each block is still computed with respect to the basic version of the PropNet (BasicPN). As the table shows, regarding the speed, Opt12 seems to be the one achieving the best overall performance. However, the performance of Opt102 is rather close, as expected, because these two combinations should reduce each PropNet to the same number of components. The small difference in performance is probably due the reasons already mentioned in Subsection 4.3.1. Both the difference in the random seed used for each repetition of the

Table 4.2: Comparison of combined optimizations.

	Game	BasicPN	Opt12	Opt102	Opt13	Opt1023
Avg. speed (nodes/second)	Amazons	35.0	38.5	41.4	32.3	41.0
	Battle	34 957	59 308	59 697	39 981	60 419
	Breakthrough	50 557	66 943	66 551	66 833	66 991
	Chin.Checkers1P	426 374	570 858	562 737	541 682	561 634
	Chin.Checkers2P	125 581	194 442	192 048	190 161	193 752
	Chin.Checkers3P	155 886	175 410	176 162	170 722	176 185
	Chin.Checkers4P	105 766	130 362	130 279	129 194	130 451
	Chin.Checkers6P	119 650	127 535	128 111	127 619	129 000
	Connect Four	110 081	127 053	126 535	105 978	129 272
	Othello	290	1 934	1 894	245	1 979
	Pentago	76 336	116 353	115 064	117 127	121 108
	Skirmish	5 887	7 075	7 042	7 403	7 600
	Tic Tac Toe	223 403	259 980	257 285	247 246	257 525
	Avg. rel. increase	–	70.32%	69.39%	17.38%	73.48%
Avg. number of components	Amazons	1 497 649	623 460	623 460	711 596	596 240
	Battle	51 197	11 084	11 077	36 676	10 902
	Breakthrough	10 745	5 900	5 900	5 869	5 836
	Chin.Checkers1P	793	556	556	559	556
	Chin.Checkers2P	1 540	1 172	1 172	1 179	1 172
	Chin.Checkers3P	2 411	1 718	1 718	1 845	1 718
	Chin.Checkers4P	3 159	2 362	2 362	2 465	2 362
	Chin.Checkers6P	4 451	3 238	3 238	3 473	3 238
	Connect Four	2 164	1 063	1 063	1 724	1 056
	Othello	1 311 988	208 510	208 510	1 031 580	206 846
	Pentago	3 696	1 464	1 473	1 338	1 337
	Skirmish	126 019	107 296	107 296	62 427	61 552
	Tic Tac Toe	312	239	239	249	239
	Avg. rel. increase	–	-42.34%	-42.32%	-32.52%	-45.65%
Avg. total init. time (ms)	Amazons	311 335	411 905	400 113	312 793	401 559
	Battle	5 756	6 367	6 233	5 968	6 329
	Breakthrough	3 989	4 354	4 415	3 982	4 328
	Chin.Checkers1P	2 699	2 693	2 654	2 707	2 652
	Chin.Checkers2P	2 848	2 848	2 843	2 817	2 842
	Chin.Checkers3P	3 162	3 214	3 186	3 160	3 167
	Chin.Checkers4P	3 258	3 405	3 330	3 275	3 379
	Chin.Checkers6P	3 225	3 423	3 430	3 207	3 395
	Connect Four	2 437	2 536	2 555	2 417	2 525
	Othello	35 756	39 170	36 689	35 359	37 804
	Pentago	4 249	4 269	4 286	4 308	4 325
	Skirmish	11 887	12 386	12 285	11 870	12 577
	Tic Tac Toe	1 525	1 532	1 535	1 524	1 555
	Avg. rel. increase	–	6.38%	5.18%	0.18%	5.66%

game and the variance in the number of components generated by the algorithm that creates the initial PropNet can influence the performance.

One more thing that can be noticed from Table 4.2 is that running Opt0 before Opt2 helps reducing the initialization time for large games, while it seems to have almost no effect on smaller games. Moreover, Opt13 is the one that, regarding the speed, performs worse in this series of experiments, thus it has been excluded from further tests. Among Opt12 and Opt102, it has been chosen to keep testing on top of Opt102 because of its shorter initialization time for games with large PropNets, given that its speed is still comparable with the one of Opt12.

Using Opt102 as starting point, there is only one more interesting combination of optimizations left to test: Opt1023. No further gain in performance can be obtained by repeating the same optimizations multiple times in a row, since no further change will take place in the structure of the PropNet. Thus, it is not interesting to evaluate combinations of optimizations that extend Opt1023.

The last column of Table 4.2 shows the statistics for Opt1023. For most of the games, Opt1023 seems to be the fastest. It is also the one that reduces the number of PropNet components the most. As for the initialization time, this optimization is between a few milliseconds and a bit more than 1 second slower that the basic version of the PropNet, except for Amazons. Optimizing the large PropNet of Amazons can slow down the initialization time by more than a minute.

4.3.4 Comparison of PropNet Reasoner and Prover

In this series of experiments the reasoner based on the overall fastest combination of optimizations among the tested ones (Opt1023) is compared with the Prover. More precisely, the PropNet reasoner and the Prover are compared measuring their speed over complete games (as opposed to previous experiments that were comparing the speed only over the first step of the game). Moreover, for both of them also a cached version is tested (i.e. CachedProver and CachedOpt1023).

The results of these experiments are shown in Table 4.3. The last row of this table reports for both CachedProver and CachedOpt1023 the average percentage increase of the speed with respect to their non-cached versions. From the table it is visible how the reasoner with the optimized PropNet achieves a much better performance than the Prover in the considered games. When adding the cache to both reasoners the difference in performance is reduced for some games. Although, the cached PropNet reasoner is still faster than the cached Prover in all of them.

The use of a cache provides some benefits increasing the overall performance of both reasoners with respect to their non-cached version. However, the cache gives more benefits to the Prover. For the Prover the speed is increased for almost all the games, while for the PropNet reasoner it is increased for some, but decreased for others. To be noticed is that the increase in speed provided by the cache is especially relevant in the games of Chinese Checkers with 1 player and Tic Tac Toe. These two games have a relatively small search space, therefore cached query results are reused often and searching in the cache is not too time consuming with only a low number of entries.

Moreover, observing the results for all the Chinese Checkers versions it is clear

Table 4.3: Comparison of the PropNet reasoner with the Prover and effect of the cache.

	Game	Prover	CacheProver	Opt1023	CacheOpt1023
Avg. median speed (nodes/second)	Amazons	7	7	28	29
	Battle	43	44	44 260	39 945
	Breakthrough	233	238	58 763	53 945
	Chin.Checkers1P	2 281	538 930	536 264	921 637
	Chin.Checkers2P	1 456	3 205	170 609	213 181
	Chin.Checkers3P	1 105	1 296	127 493	99 527
	Chin.Checkers4P	544	659	87 252	88 167
	Chin.Checkers6P	608	682	61 627	55 242
	Connect Four	180	217	132 211	177 567
	Othello	3	3	611	643
	Pentago	150	152	99 338	81 016
	Skirmish	23	24	3 007	3 036
	Tic Tac Toe	1 709	323 599	224 083	568 255
	Avg. rel. increase	–	3 274 %	–	17.45 %

that the speed of the cached Prover and the speed of the cached PropNet reasoner both decrease when increasing the number of players. However, for the PropNet reasoner this decrease is slower. For Chinese Checkers with 1 player the cached PropNet reasoner is about 2 times faster than the cached Prover, while for the version with 6 players it is 81 times faster.

When performing the experiments it was also noticed that in many games the cache decreases the speed of the PropNet reasoner during the initial steps. This loss is then balanced towards the endgame, when the chance of finding cached query results increases. It takes some time for the cache to be filled with a sufficient number of entries and thus have a positive impact on the speed of the PropNet reasoner. The same effect was not observed for the Prover. For the first steps of the games the cache did not decrease the speed of the Prover for any of the games, and for some of them increased it. The explanation for this is that the time for computing the answer of a query with the Prover is in general much higher than the one of the PropNet. Thus, for the Prover finding in the cache even a small number of query results saves enough computational time to compensate the extra time spent looking in the cache for results that are not present yet. Detailed results for the performance of the cache on each single game are reported in Appendix C.

The results of Table 4.3 also help putting the PropNet reasoner into perspective with the other GDL reasoners analyzed by Schiffel and Björnsson (2014). Even if they use different experimental settings than in this chapter, some general observations can still be made. Considering the performance of the reasoners that, like the PropNet reasoner, rely on an alternative representation of the GDL description, it seems that the implementation of the PropNet reasoner presented in this chapter provides for most of the games a speed increase of the same order of magnitude when compared to the Prover. Moreover, for Amazons, Othello, Chinese Checkers with

Table 4.4: Win percentage of the PropNet agent against the Prover agent.

Game	**Opt1023**
Battle	100.0(±0.00)
Breakthrough	100.0(±0.00)
Chin.Checkers2P	96.0(±2.72)
Chin.Checkers3P	77.5(±5.75)
Chin.Checkers4P	68.1(±6.32)
Chin.Checkers6P	64.7(±5.73)
Connect Four	99.3(±1.09)
Pentago	100.0(±0.00)
Skirmish	100.0(±0.00)

4 and 6 players, and Skirmish it seems that the reasoner based on the optimized PropNet, could even achieve a better performance in similar circumstances.

4.3.5 Game-Playing Performance

In this series of experiments an instance of the MCTS agent that uses the cached PropNet reasoner with the fastest combination of optimizations (Opt1023) is matched against an instance that uses the cached Prover. Because Subsection 4.3.4 showed the cache to be overall beneficial for both reasoners, it has been included in this experiment. Table 4.4 shows the win percentage of the cached PropNet agent against the cached Prover agent with a 95%-confidence interval. The table does not include the results for the single-player version of Chinese Checkers because this game is tested separately and the score is used to measure the performance of the agent instances. The search space of this game is relatively small, so both instances achieved the maximum score in every match. The score for Tic Tac Toe is not included because its state space is so small that both instances can easily reach a sufficient number of simulations to play optimally, resulting in a tie. No results are shown for Amazons and Othello because for both games, during the first game turns, the cached Prover agent could not return a move within the given time limit. Even with the cache, during the first turns the number of memorized query results is not sufficient to allow the Prover agent to complete even one MCTS simulation within the time limit.

Looking at the results for the remaining games, for most of them the cached PropNet agent achieves a win percentage close or equal to 100%. The games in which the performance of the cached PropNet agent seems to drop are the ones with more than 2 players. Chinese Checkers with 4 and 6 players are the ones where the win percentage for the cached PropNet agent is the lowest, but it is still significantly better than the one of the cached Prover agent. Another reason for the lower performance of the PropNet agent on Chinese Checkers is that agents are using random play-outs, which have been shown to perform poorly on this game. Chinese Checkers presents actions that do not make the game progress toward the

termination of the game (e.g. pieces can be moved back and forth on the board without getting closer to the goal). This causes random play-outs to often be stopped before reaching a terminal state, thus returning uninformative results. With the higher number of simulations performed by the PropNet agent, the performance of random play-outs worsens more than for the Prover agent.

4.4 Encoding PropNets on Field Programmable Gate Arrays

The reasoner evaluated in the previous section is based on a software implementation of the PropNet. However, the resemblance of the PropNet structure to a logic circuit makes it suitable to be encoded on a Field-Programmable Gate Array (FPGA) (Brown *et al.*, 2012). FPGAs are integrated logic circuits that can be configured by the end-user. They are made out of thousands of interconnected Universal Logic Modules (ULMs), which can be individually programmed to perform simple logic operations and arbitrarily connected with each other. Encoding a PropNet on an FPGA could result in a significant increase of the computational speed used for propagating the truth values of the PropNet components. Therefore, the use of a reasoner based on FPGA-PropNets has the potential to speed up MCTS by increasing the number of simulations that can be performed in a fixed amount of time. Another characteristic of FPGAs is that they can be re-programmed when necessary, being particularly suitable for a domain like the Stanford GGP project, which requires to encode a new PropNet whenever a new game is being played.

This section presents results obtained by testing a reasoner based on an FPGA-PropNet. Subsection 4.4.1 gives an overview of the implementation of a reasoner based on an FPGA-PropNet, while Subsection 4.4.2 presents the results obtained by testing such reasoner.

4.4.1 FPGA-PropNet Reasoner Implementation

To implement a reasoner based on the FPGA-PropNet the software version of the PropNet is first generated and optimized as described previously (i.e. applying the sequence of optimizations that performed overall fastest, Opt1023). Subsequently, its structure is coded in *Verilog*, one of the *Hardware Description Languages* that are commonly used to define the structure and the behavior of FPGAs. Each PropNet component is coded as an instance of a Verilog module that implements the behavior of the corresponding component type, and these modules are connected according to the structure of the software version of the PropNet. Once the PropNet has been coded in Verilog, all the defined modules are fitted on the FPGA by a built-in algorithm that decides the placement of each module on the physical board.

Meta-information about the FPGA-PropNet is also recorded, which contains the FPGA-PropNet initial state, the encoding of the actions and other information necessary to operate on the FPGA-PropNet. This meta-information is used to initialize a driver library programmed in Java, which enables communication between

the search agent and the FPGA-PropNet. The library offers the following three methods to interact with the FPGA-PropNet:

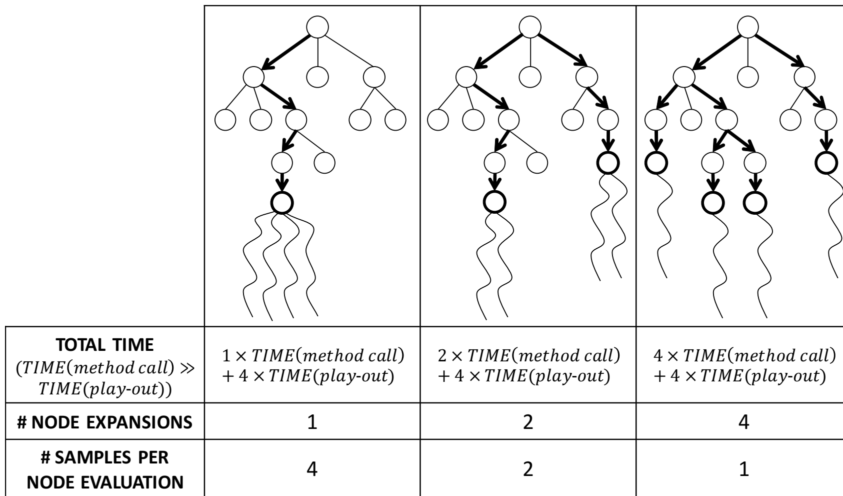
- s **getInitialState():** returns the initial state of the game in the FPGA encoding.

- $(A_{i,j}, \overrightarrow{(s, \vec{a}\,)})$ **getNextStates(s):** given a state s in the FPGA encoding, returns a matrix $A_{i,j}$ where each row i corresponds to the list of legal moves for player i in s, and a list of pairs $(s', \vec{a}\,)$ where $\vec{a}$ is one of the available joint actions in s and s' is the state reached by performing such joint action. States and actions are returned in the FPGA encoding.

- $\vec{q}$ **getScores(s,n):** given an FPGA state s and an integer n, performs n random play-outs from state s and returns the list of payoffs $\vec{q}$, where each entry q_i is the average payoff obtained by player i over all the n play-outs.

These methods are mostly different from the ones available to interact with the software implementation of the PropNet (see Subsection 4.2.3). This is because communication between the software implementing the game-playing agent and the hardware encoding the FPGA-PropNet is time-consuming. Therefore, most of the computation is delegated to the hardware and communication is reduced to the minimum. Given a state, it is more efficient to compute legal actions for all the roles and possible next states all at once, instead of having multiple calls to the FPGA-PropNet to compute them separately. Similarly, it is more efficient to delegate to the FPGA-PropNet the computation of an entire random play-out rather than selecting each random action on the software side and communicating with the FPGA-PropNet to generate intermediate states. Therefore, efficiency is increased further if a batch of random play-outs is performed all at once at a leaf node.

There are, however, some disadvantages to this implementation. First of all, faster play-outs do not always mean that more information is collected about different parts of the game. There is a tradeoff between play-out speed and number of distinct Monte-Carlo evaluations. Figure 4.8 gives examples of this tradeoff. Given a fixed number of play-outs, it is faster to use all of them to evaluate a single state, like it is done for the first tree in the figure, because the time-consuming communication with the FPGA PropNet to call the getScore(s,n) method happens only once. However, it is less informative than dividing the play-outs over more states. For the third tree in the figure it takes more time to perform the same number of play-outs, because each of them is performed from a different node and therefore the number of calls to the getScore(s,n) method equals the number of play-outs. However, more information is collected, because more nodes are added to the tree (one for each play-out instead of one in total). The second tree in the figure presents a situation that is in-between, with fewer nodes being evaluated than in the third tree, but more information being collected than for the first tree. Moreover, the accuracy of the state evaluations changes depending on how play-outs are distributed. The more play-outs are used to evaluate the same node, the more accurate the estimate, but at the same time fewer nodes are added to the tree and evaluated. This situation shares a similarity with leaf parallelization of MCTS (Chaslot, Winands, and



TOTAL TIME ($TIME(method\ call) \gg TIME(play\text{-}out)$)	$1 \times TIME(method\ call) + 4 \times TIME(play\text{-}out)$	$2 \times TIME(method\ call) + 4 \times TIME(play\text{-}out)$	$4 \times TIME(method\ call) + 4 \times TIME(play\text{-}out)$
# NODE EXPANSIONS	1	2	4
# SAMPLES PER NODE EVALUATION	4	2	1

Figure 4.8: Tradeoff between MCTS play-out speed and distinct Monte-Carlo evaluations.

van den Herik, 2008a), which evaluates a leaf node during one MCTS simulation by multiple parallel play-outs instead of using a single play-out. Like using the FPGA to perform multiple play-outs from the same node, using leaf parallelization enables the agent to collect more accurate samples for a single game state. However, it increases the amount of time that is spent on average on a single MCTS simulation, although for a different reason. Leaf parallelization increases this amount of time because it always has to wait for the longest among the parallel play-outs to be over before starting a new MCTS simulation.

Another disadvantage of the implementation of the FPGA-PropNet methods is that the play-out strategy is encoded in the hardware. Whenever a new play-out strategy has to be tested it must first be encoded in the hardware. Thus, no other play-out strategy that might be already implemented by the game-playing agent can be tested with the FPGA-PropNet reasoner. Finally, if the FPGA-PropNet reasoner has to be used by a game-playing agent based on "open-loop" MCTS, similar to the one tested in this thesis, a modification to the algorithm is required. To reduce the number of calls to the methods of the FPGA library, each node in the MCTS tree has to memorize all possible joint moves and corresponding next states. In this way the GETNEXTSTATES(s) method can be called only once whenever a new node is added to the tree. However, this causes an increase in the memory used by the tree.

4.4.2 FPGA-PropNet Reasoner Performance

This section presents results obtained by testing a reasoner based on an FPGA-PropNet.[3] To test the speed of the reasoner the number of states visited during

[3] Cezary Siwek and Jakub Kowalski, from the University of Wrocław, Poland, implemented the FPGA-PropNet reasoner and ran the experiments that test its speed, initialization time and memory usage.

Table 4.5: Comparison of the FPGA-PropNet (FPGA-PN) with the software PropNet (Opt1023) and the Prover, based on running random simulations from the initial game state. The speed of the FPGA-PropNet is equivalent to the clock frequency, which is probed in 1.0Mhz steps.

	Game	FPGA-PN	Opt1023	Prover
Avg. speed (nodes/sec.)	Horseshoe	8 500 000	192 583	3.8
	Connect Four	7 000 000	285 908	561
	Pentago	7 000 000	119 111	342
	Joint Connect Four	4 500 000	171 575	270
	Breakthrough	1 400 000	38 015	601
	Reversi	1 171 875	4 806	19

random play-outs from the initial game state is used. The performance is compared both with the reasoner that uses the software implementation of the PropNet and with the Prover. The initialization times of the FPGA-PropNet and the software implementation of the PropNet are also compared and their memory usage analyzed.

The experiments use the TerasIC DE1-SoC board containing the Altera's Cyclone V series SoC: 5CSEMA5F31C6. The algorithm that uses the FPGA-PropNet reasoner is run on a computer embedded in the before-mentioned SoC with ARM Cortex A9, Dual core @925Mhz with 1 GB RAM, running Debian 9 Strech 32-bit. The FPGA project compilation is performed on Intel Core i5-4670 with 16 GB DDR3 @1600Mhz RAM using Ubuntu 16.04 server 64-bit and Intel Quartus Prime Lite Edition 17.0 as IDE. The reasoner that uses the software implementation of the PropNet and the Prover are tested on a Linux server consisting of 64 AMD Opteron 6174 2.2-GHz cores and 252 GB RAM.

The reasoners are tested on the following games: Horseshoe, Connect Four, Pentago, Joint Connect Four, Breakthrough and Reversi. This set of games differs from the one used in the experiments presented in Section 4.3 for two main reasons. First, some of the games used previously have a PropNet that is too big to fit on the FPGA board. Second, the initial design of the FPGA-PropNet was set-up to manage only 2-player games. Therefore, a new set of games has been selected, trying to include games with different complexities and different PropNet sizes. The rules and relevant properties of these games can be found in Appendix B.1. Differently from the experiments performed in Section 4.3, the GDL descriptions of the games used in this series of experiments can be found on the Stanford Gamemaster repository (Schreiber, 2018). Note that the GDL description of a game, although expressing the same game rules, might differ over different game repositories. Therefore, the software PropNet might have different average speed depending on the GDL game description that is used for a particular game.

The results of the experiments that compare the speed of the reasoners are presented in Table 4.5. They are based on 1 million simulations for the FPGA-PropNet reasoner, and more than 250 000 simulations for the other reasoners, except for Reversi, for which only 1 000 simulations have been performed. This choice is due to

Table 4.6: Initialization time and memory usage of the FPGA-PropNet (FPGA-PN) and the software PropNet (Opt1023).

Game	Initialization time		Memory usage	
	FPGA-PN (min)	**Opt1023 (sec)**	**#Propnet components**	**FPGA chip utilization**
Horseshoe	4:20	0.45	350	7%
Connect Four	5:37	0.67	814	12%
Pentago	5:20	2.70	1 291	13%
Joint Connect Four	5:53	1.00	1 614	16%
Breakthrough	12:03	1.35	17 752	72%
Reversi	14:08	23.91	56 014	41%

the higher amount of time required to simulate this game with respect to the other tested games. As expected, the usage of FPGAs substantially increases the reasoner efficiency. For all games except Reversi, the improvement factors with respect to the software PropNet are between 24.5 (Connect Four) and 58 (Pentago). For Reversi, which produces the largest PropNet among the tested games, the FPGA-PropNet reasoner computes states over 243 times faster than the software PropNet. This high speed increase is mainly due to the hardware PropNet optimizations performed when fitting the PropNet on the board. The closer the graph structure of the PropNet is to being planar, which seems to be the case for Reversi, the easier it is to reduce chip utilization when fitting it on the board, therefore increasing computational speed.

The downside of moving from software to hardware is a considerable increase of initialization time, as shown in Table 4.6. Instead of a few seconds, like for the initialization of the software PropNet, it takes around 5 to 6 minutes for small and medium games, and for larger it is almost 15 minutes. Such times prevent game-playing agents from being ready to play during the typical initialization clock of the Stanford GGP competition.

A preliminary study has also been performed, which tests the performance of an MCTS agent that uses the presented FPGA-PropNet reasoner as described in Subsection 4.4.1. This study showed that, when performing MCTS, the communication between the software implementing the search and the hardware encoding the PropNet introduces too much overhead for the FPGA-PropNet agent to be competitive with the agent that uses the software implementation of the PropNet. More precisely, when performing MCTS play-outs with the FPGA-PropNet, a high batch size enables the FPGA-PropNet reasoner to perform a higher number of simulations than the software PropNet reasoner. However, each batch of simulations is performed on a single node only, causing the tree to be expanded less often, therefore not reaching the same size as the tree built using the software PropNet reasoner. Conversely, a small batch size enables the FPGA-PropNet agent to expand more nodes, but the overhead caused by the communication between software and hardware increases consistently, once again preventing the MCTS tree to grow as much as it does for the software PropNet agent.

4.5 Chapter Conclusions and Future Research

In this chapter the performance of a PropNet-based reasoner has been evaluated, together with four optimizations of the structure of the PropNet and their impact on the performance. Moreover, a reasoner based on the encoding of the PropNet on FPGAs has been tested. Even though the tested implementation of the PropNet is based on the code provided by the GGP Base Package, the principles behind its representation and its optimizations can also be applied in general.

Experiments show that the use of a software implementation of the PropNet substantially increases the reasoning speed by, on average, at least two orders of magnitude with respect to the GGP Base Prover. Moreover, the addition of a combination of optimizations that reduce the size of the PropNet increases the reasoning speed further. Results suggest that both optimizations that simply remove components that are not meaningful for the game, such as Opt1 and Opt3, and optimizations that remove components and re-arrange the connections between them after analyzing their truth values, such as Opt0 and Opt2, are beneficial. It is also important to consider the order in which optimizations are performed to reduce the overhead that they cause on the initialization time.

The addition of a cache has been shown to have an overall positive effect on the speed of the software PropNet. For small games its effect is already visible in the first turns, while for most of the other games it helps only during later game turns, sometimes slowing down the speed in the initial turns. Given that the outcome of the game might be decided already by choices made in the first turns, a slower PropNet speed due to the use of a cache might be detrimental. Therefore, the use of the cache is recommended only for games with a small search space.

Experiments also show that the speed increase has a positive effect on the performance of the software PropNet-based instance of an MCTS agent. This agent achieves a win rate close to 100% in most of the games for which it is matched against an equivalent agent based on the Prover. Finally, research on implementing the PropNet on an FPGA showed that FPGA-PropNets are a faster alternative to software PropNets for reasoning on GDL game descriptions, opening up a promising research direction. Given the experimental results, it may be concluded that using a PropNet with an optimized structure to represent game rules written in GDL is beneficial for MCTS-based agents. A reasoner based on optimized PropNets enables MCTS to perform a higher number of simulations within a fixed time frame.

Future research could further investigate the use of the cache for the software implementation of the PropNet, for example by devising a strategy to detect for each game if and when the use of a cache is helpful. Another interesting aspect that future research could consider is the impact that the use of different strategies to propagate truth values among the components of the PropNet would have on the reasoning speed. For the software implementation of the PropNet truth values are computed for one component at a time. This offers two main propagation options that could be tested. The first is forward propagation, which, whenever a component changes truth value, immediately propagates the change to its outputs. The second is backward propagation, which, whenever the truth value of a component needs to be computed, first computes the truth values of its inputs recursively.

The use of the FPGA-PropNet reasoner can also be further investigated. First, its performance when integrated in an MCTS agent can be improved by compensating the increased communication overhead. This could be done, for example, by embedding MCTS on the FPGA, or by using hardware with shorter communication latency. Moreover, if the integration of the FPGA-PropNet reasoner within MCTS can be improved to achieve a higher simulation speed, it would be interesting to test other MCTS play-out strategies, to see how the speed increase influences their performance.

Chapter 5

Rapid Action Value Estimation Variants

This chapter is based on:

- Sironi, Chiara F., and Winands, Mark H.M. (2016). Comparison of Rapid Action Value Estimation Variants for General Game Playing. *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pp. 309–316.

During the selection phase of MCTS the tree is traversed from the root to a leaf node using a selection strategy to decide which joint action to visit in each node. A commonly used selection strategy is UCT (Kocsis and Szepesvári, 2006). Previous research has shown that enhancing the UCT strategy can consistently improve the overall performance of MCTS (Chaslot *et al.*, 2008b; Finnsson and Björnsson, 2010; Nijssen and Winands, 2011; Gelly and Silver, 2011; Cazenave, 2015). Many enhancements have been proposed to improve UCT. Some have been proposed for particular games as they rely on game-specific knowledge (Chaslot *et al.*, 2008b). This makes them less interesting for GGP. Others, instead, are intrinsically domain-independent (Finnsson and Björnsson, 2010; Gelly and Silver, 2011) or are domain-independent modifications of game-specific methods (Nijssen and Winands, 2011), and are thus suitable to be applied in GGP.

Among the domain-independent enhancements for the selection phase of MCTS that have been shown to be successful in GGP is the Rapid Action Value Estimation technique (RAVE) (Gelly and Silver, 2007; Gelly and Silver, 2011; Finnsson and Björnsson, 2010). One of the most successful agents that took part in the Stanford GGP competition, CadiaPlayer, has implemented the RAVE technique. Recently, a generalization of RAVE, the Generalized Rapid Action Value Estimation (GRAVE) technique, has been proposed (Cazenave, 2015) and has been shown to perform better than RAVE on some variants of Go and some other games. What RAVE and GRAVE have in common is that they both bias action selection in a state using information about the general performance of the actions in the game. The

difference is that RAVE uses information collected locally for the state where the action is being selected, while GRAVE uses information collected for an ancestor state when the current state has been visited only a few times. Using more local or more global information to enhance the search might influence the performance of MCTS differently.

This chapter answers the second research question by proposing another variant of RAVE, History Rapid Action Value Estimation (HRAVE). Differently from RAVE and GRAVE, HRAVE always uses information collected for the current root state of the search tree. The performance of the RAVE, GRAVE and HRAVE strategies is compared to verify how the search is influenced by the use of information at different levels (from more local in RAVE to more global in HRAVE, with GRAVE being in between). In addition, it is also shown how the performance of these RAVE variants is influenced by a play-out strategy that is more informed than the random play-out strategy. This is done by combining RAVE, GRAVE and HRAVE with MAST (Finnsson and Björnsson, 2008). The Stanford GGP project is used as test domain.

This chapter is structured as follows. Section 5.1 introduces the All-Moves-As-First (AMAF) statistics, on which the presented RAVE variants are based. Subsequently, Section 5.2 describes the RAVE, GRAVE and HRAVE strategies. The experimental setup and the obtained results are discussed in Section 5.3. Finally, Section 5.4 gives the conclusion and mentions possible future research.

5.1 All-Moves-As-First

All the RAVE variants evaluated in this chapter are based on the idea of the All-Moves-As-First (AMAF) heuristic (Brügmann, 1993), which collects actions statistics assuming that the order in which actions are visited in a simulation from the current node is irrelevant. This way of collecting statistics enables a search agent to accumulate a higher number of samples in a shorter amount of time. The AMAF statistics are collected updating the expected value of an action in a state not only when the action has been selected immediately in the state, like UCT does, but every time the action has been selected at any moment after the state was visited. More precisely, suppose that after a simulation the statistics of a visited state s are being updated. The UCT backpropagation updates the expected value $\bar{q}_{(s,a_i)}$ only for the action a_i that has been selected for player i in s during the simulation. The AMAF backpropagation, instead, updates the expected value $AMAF_{(s,a_i)}$ of all the actions a_i that have been selected during the simulation in any state visited after s.

Figure 5.1 compares the UCT backpropagation with the AMAF backpropagation. The tree in the figure represents a one-player game. The actions of the only player, Player 1, are identified as $a_{1,j}$, where the subscript j is used to distinguish among the actions of the player. The bold edges identify the path visited during the last performed simulation. For this simulation, the figure reports the action statistics that are updated by UCT ($\bar{q}_{(s,a_i)}$) and the action statistics that are updated by the AMAF backpropagation ($AMAF_{(s,a_i)}$).

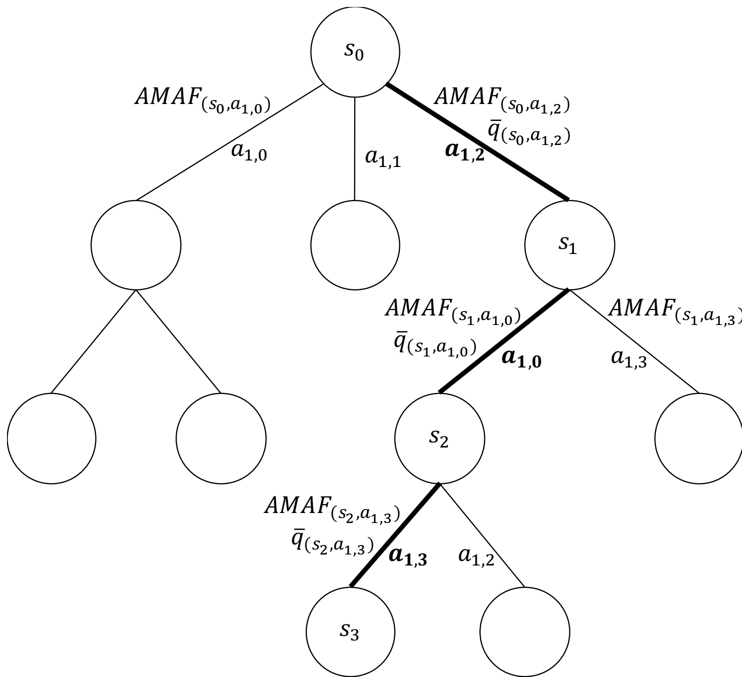


Figure 5.1: UCT vs AMAF statistics update.

5.2 RAVE Variants

This section describes the RAVE variants. RAVE is presented in Subsection 5.2.1, GRAVE in Subsection 5.2.2 and HRAVE in Subsection 5.2.3.

5.2.1 Rapid Action Value Estimation

The *Rapid Action Value Estimation* (RAVE) strategy has been proposed in order to speed up the learning process inside the MCTS tree, and it has been successfully applied in Go (Gelly and Silver, 2007; Gelly and Silver, 2011) and in GGP (Finnsson and Björnsson, 2010). The UCT strategy bases the selection of an action in a node on the estimated value obtained by sampling this action in the node multiple times. However, especially when the state space is large, UCT requires many simulations before it can sample all the actions in a node and more simulations before it can accumulate enough samples for the actions to reduce the variance of their estimated payoff. To overcome this issue, RAVE combines the UCT value of the actions with their AMAF value, therefore it memorizes in each tree node, for each action a_i of each player i, the following statistics:

- The expected payoff $\bar{q}_{(s,a_i)}$ obtained from all the simulations in which move a_i is selected for player i in state s. This is the same value used in the UCT formula (Formula 2.9).

- The expected payoff $AMAF_{(s,a_i)}$ obtained from all the simulations in which move a_i is selected for player i further down the path that passes by node s.

This means that, when backpropagating the result of a simulation in a certain node s of the tree, the value $\bar{q}_{(s,a_i)}$ is updated for the action a_i that was directly played in the state, and the value $AMAF_{(s,a_i)}$ is updated for all the legal actions a_i in s that have been encountered at a later stage of the simulation. In this way RAVE can collect more samples and use them to reduce the variance of the action value estimates for the nodes that do not have many visits. Using the AMAF statistics enables to gather more information faster, although this information is more global than the UCT statistics. AMAF statistics are useful for less visited nodes, but when the number of visits increases, the UCT statistics become more reliable and the influence of the AMAF statistics should progressively decrease. This is why the RAVE strategy keeps track of the two scores separately and uses a weight β to reduce the importance of the AMAF statistics over time.

Different variants for the RAVE action evaluation formula and for the β parameter computation have been proposed (Gelly and Silver, 2007; Teytaud and Teytaud, 2010; Gelly and Silver, 2011). This chapter uses the same formula that has been first used in GGP by the CadiaPlayer agent (Björnsson and Finnsson, 2009). RAVE selects an action a_i^* for player i in a state s according to Formula 5.1).

$$
\begin{aligned}
a_i^* &= \operatorname*{argmax}_{a_i \in \mathcal{A}_{(s,i)}} \left\{ UCT_{\mathrm{RAVE}}(s, a_i) \right\} \\
UCT_{\mathrm{RAVE}}(s, a_i) &= (1 - \beta_s) \times \bar{q}_{(s,a_i)} + \beta_s \times AMAF_{(s,a_i)} + C \times \sqrt{\frac{\ln n_s}{n_{(s,a_i)}}} \qquad (5.1) \\
\beta_s &= \sqrt{\frac{K}{3 \times n_s + K}}
\end{aligned}
$$

The parameter K is known as the *equivalence parameter* and indicates for how many simulations the UCT and the AMAF statistics of an action are weighted equal. Note the similarity of the RAVE formula with the UCT formula (Formula 2.9). The only difference is that the expected value of an action is computed as a weighted average of the expected action value used by the original UCT formula and the AMAF expected action value.

5.2.2 Generalized Rapid Action Value Estimation

GRAVE (Cazenave, 2015) is a modification of RAVE that has been proposed to overcome one of its drawbacks. A problem of RAVE is that for the nodes close to the leaves of the tree not only the UCT statistics are based on a low number of samples, but also the AMAF statistics. Therefore, in these nodes the estimates of the moves values have less accuracy.

To solve this problem, for the nodes that have a number of visits lower than a given *ref* threshold, GRAVE uses the AMAF statistics of an ancestor node to compute the value of the actions. Each node in the tree memorizes its own AMAF

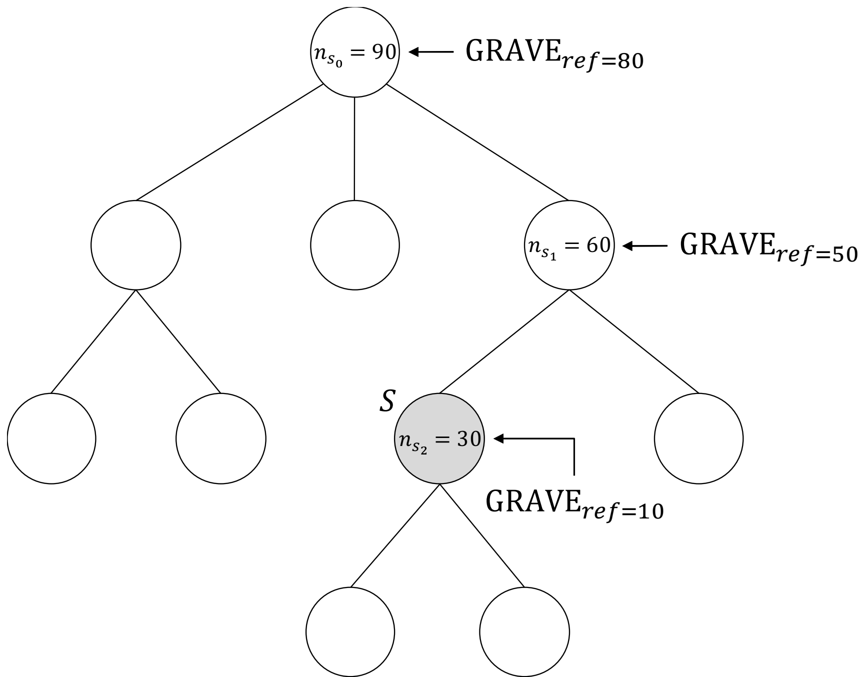


Figure 5.2: Use of AMAF statistic for the GRAVE strategy.

statistics, but keeps also a reference to the closest ancestor that has a sufficient number of visits for its AMAF statistics to be considered reliable. When a node s has sufficient visits ($n_s > ref$), it starts using its own AMAF statistics instead of the ones of an ancestor, and the strategy in that node starts behaving like RAVE. Note that, if $ref = 0$, GRAVE behaves exactly like RAVE from the beginning of the search. Figure 5.2 gives an example for the GRAVE strategy. Assume that node S (in gray) is the node where the strategy has to select an action. Each node relevant for the example reports its number of visits. The figure shows in which node GRAVE would look for the AMAF statistics depending on how the *ref* parameter is set.

The GRAVE strategy enables the agent to increase the accuracy of the estimates for the less visited nodes. However, the AMAF scores of an ancestor might be less relevant for its descendants, because these scores refer to a different game state. Another aspect to be mentioned is the increased memory usage of GRAVE with respect to RAVE. The latter needs only to store an extra statistic for each legal move in the node. With GRAVE, instead, the AMAF scores in a node might be used for other nodes lower in the tree that have a different set of legal moves. Therefore, each node has to memorize the AMAF scores for all the moves that can be encountered at any lower level in the tree.

5.2.3 History Rapid Action Value Estimation

HRAVE is exactly the same as GRAVE, except that it always uses the AMAF statistics of the current root of the tree (i.e. the *ref* parameter is set to infinity). Note that HRAVE shares similarities with the domain-independent selection strategy known as Progressive History (PH) (Nijssen and Winands, 2011). As described in Subsection 2.5.1, PH adds to the UCT formula a bonus that depends on the *relative history* of the move being evaluated. This relative history is defined as the average result of all the simulations where the move was played. The influence of this bonus decreases over time as the number of visits of the node increases and the UCT estimate becomes more reliable. In the case of HRAVE, the AMAF estimate that is used to compute the value of an action can be compared to the Progressive History bonus. This is because both the AMAF estimate and the PH bonus are computed using the same statistics. In each turn of the game, the AMAF statistics in the root of the tree correspond exactly to the history heuristic statistics used by PH. Moreover, like in PH, the influence of the AMAF statistics decreases over time and makes the move evaluation formula converge to a pure UCT strategy.

A difference between HRAVE and PH is that the decrease of the influence of the AMAF statistics for HRAVE only depends on the increase in number of visits of a node, while for PH it also depends on the number of losses associated with the action being evaluated (Nijssen and Winands, 2011). In this way, PH makes sure that actions that have been performing generally well in a state are biased longer than actions that have been performing poorly. Another difference between HRAVE and PH is that, for HRAVE, at the beginning of the search for a given turn, the root node already contains some statistics collected during previous turns. PH, instead, starts each turn with no statistics (Nijssen, 2013). For HRAVE it was decided to collect the statistics in each node also during the previous turns to have a fair comparison with GRAVE and RAVE. Both RAVE and GRAVE, at every turn except the first, start the search already having some AMAF statistics in the nodes of the tree. In order to collect statistics that can be reused in subsequent turns, HRAVE memorizes the AMAF statistics in the same way as GRAVE does, thus their memory consumption is the same.

HRAVE can also be seen as the opposite of RAVE. While the latter uses the most local AMAF statistics, the former uses the most global ones. GRAVE can be placed in between, it starts with more global AMAF statistics and then converges to the most local ones. Figure 5.2 gives an example of the use of AMAF statistics for the three RAVE variants. The number reported in the relevant nodes is the number of their visits. For the selection of an action in the gray node S, the figure shows in which node each strategy looks for the AMAF statistics to use, assuming $ref = 50$ for GRAVE.

5.3 Experiments

In this section an empirical evaluation of the performance of RAVE, GRAVE and HRAVE is presented. Subsection 5.3.1 introduces the games used in the experiments, while Subsection 5.3.2 describes the setup of the performed experiments.

Next, Subsection 5.3.3 shows the results obtained by tuning the equivalence parameter K. The RAVE variants are matched against UCT in Subsection 5.3.4 and in Subsection 5.3.5 they are matched against UCT with the addition of the MAST playout strategy. Next, the RAVE variants are matched against each other in Subsection 5.3.6. Finally, Subsection 5.3.7 analyzes the memory usage of RAVE, GRAVE and HRAVE.

5.3.1 Games

The discussed RAVE variants have been tested on 15 different games: 3D Tic Tac Toe, Breakthrough, Knightthrough, Skirmish, Battle, Chinook, Chinese Checkers with three players, Checkers, Connect Five, Othello, Quad (the version played on a 7×7 board), Sheep and Wolf, Tic-Tac-Chess-Checkers-Four (TTCC4) with 2 and 3 players, and Zhadu. Appendix B.1 gives an overview of the rules and the main properties of these games. This set of games has been chosen because of its heterogeneity due to various game properties (i.e. number of players, constant-sum or variable-sum, simultaneous or sequential move). Some of these games differ from the ones used to test the GDL reasoners in chapter 4 because this chapter is only considering games that have already been used in previous literature to test RAVE and MAST in GGP (Finnsson, 2012b; Tak *et al.*, 2014b).

In the experiments that tune the *equivalence parameter* K for the strategies only the games 3D Tic Tac Toe, Breakthrough, Knightthrough, Skirmish, Battle, Chinook, Chinese Checkers with three players have been used to void overfitting the values of K to the whole set of tested games. All the other experiments, instead, have been run on all the 15 games. The GDL description of the considered games can be found on the GGP Base repository (Schreiber, 2016), and their rules and properties are reported in Appendix B.1.

5.3.2 Setup

The aforementioned RAVE variants were implemented for the agent developed in the GGP Base Package (see Subsection 3.1.4). The agent tested in the experiments uses a reasoner based on the software implementation of the PropNet with the combination of optimizations that performed best in the experiments presented in Chapter 4 (i.e. Opt1023). No cache is used for the PropNet because the game space of the tested games might be too large for the cache to have a positive effect already in the initial game turns. In all the series of experiments, two instances of the agent at a time are matched against each other. As mentioned in Subsection 4.3.1, the creation of the PropNet is non-deterministic and PropNets with different structures might be generated for the same game. Therefore, for each game run, the PropNet of the game is generated in advance and both agent instances use the same structure. This prevents one of the agents from getting an advantage due to a more efficient PropNet. Play-clock and start-clock are set to 1s, except for the experiments presented in Subsection 5.3.4 that are repeated also with start-clock and play-clock set to 10s.

For each game, if r is the number of roles in the game, there are 2^r different ways

in which 2 types of agents can be assigned to the roles (Sturtevant, 2008). Two of the configurations involve only the same agent type assigned to all the roles, thus are not interesting and excluded from the experiments. Each configuration is run the same number of times until the desired number of games runs have been performed.

For each of the performed experiments, the reported result is the average winning percentage of one of the two tested agent instances with a 95%-confidence interval. For each game run the instance that achieves the highest score is considered the winner and gets 1 point, while the other gets 0 points. When both instances achieve the same score, the outcome of the game run is considered a draw and both instances get 0.5 points (half win).

Two baselines to compare the different selection strategies are used, an instance of the agent implementing the MCTS algorithm with UCT selection and random play-out strategy (P_{UCT}) and an instance of the agent implementing the MCTS algorithm with UCT selection strategy and MAST play-out strategy ($P_{\text{UCT-MAST}}$). The UCT selection strategy uses the Formula 2.9 with $C = 0.7$. For the MAST strategy ϵ is set to 0.4, because it is the value that overall performed better when evaluated in (Tak *et al.*, 2012). The first-play urgency for MAST, fpu_{MAST}, is set to 100 (i.e. the maximum score that can be obtained as payoff). Moreover, the MAST statistics are decayed after playing every move with a factor $\omega = 0.2$ (i.e. 20% of the statistics is kept for the next turn). This value is set lower than the one that was found to be the best by Tak *et al.* (2014b) because for each turn the agents perform a higher number of simulations. This means that the number of collected statistics is higher and their influence needs to be decreased more strongly. A comparison of the two baseline agents can be found in Appendix D. The results in the appendix confirm the superiority of the agent that uses the more informed MAST play-out strategy over the agent that uses a random play-out strategy.

The aim of the first series of experiments is to tune the *equivalence parameter* K used to compute the weight β_s in Formula 5.1). The tested values for K are 10, 50, 100, 250, 500, 750, 1 000 and 2 000, and the parameter is tuned using the subset of games mentioned in Subsection 5.3.1. The agent instances P_{RAVE}, P_{GRAVE} and P_{HRAVE} have been implemented and matched singularly against P_{UCT} for each value of K for at least 500 runs per game. As selection strategy they use RAVE, GRAVE and HRAVE, respectively, and they all use the random play-out strategy. All of them use the value 0.2 for the C constant because a lower value than the one used for the plain UCT strategy empirically showed to achieve a better performance. For P_{GRAVE} the *ref* parameter is set to 50. For each of the three agent instances, the value of K that performed overall best in these series of experiments is also used in subsequent experiments.

In the second series of experiments, the agent instances P_{RAVE}, P_{GRAVE} and P_{HRAVE} with the best value of K are matched against P_{UCT} on all the games. Testing the instances on a wider set of games enables to detect a potential overfitting of the K value to the games used for tuning. Moreover, it enables to check whether the tuned value works well also on other games. These experiments are performed with a start-clock and play-clock of 1s and then repeated with a start-clock and play-clock of 10s. This is to verify how an increased amount of time, and thus of simulations, influences the performance of the three RAVE variants. The

minimum number of played runs per game is increased to 1000. This provides a more precise estimate of the average winning percentage, detecting with a higher confidence which of the strategies performs best.

The aim of the third series of experiments is to verify the effect of adding the MAST play-out strategy to the three variants of RAVE. For this series of experiments the random play-out strategy has been replaced with MAST to obtain the agent instances $P_{\text{RAVE-MAST}}$, $P_{\text{GRAVE-MAST}}$ and $P_{\text{HRAVE-MAST}}$. These instances have been matched only for the best value of K against $P_{\text{UCT-MAST}}$ on all the games with 1000 runs per game. Each of these instances has the same settings of the corresponding version without MAST and for the MAST strategy the settings are the same as $P_{\text{UCT-MAST}}$. Moreover, to verify whether it is beneficial to use MAST in combination with RAVE, GRAVE and HRAVE, the agents $P_{\text{RAVE-MAST}}$, $P_{\text{GRAVE-MAST}}$ and $P_{\text{HRAVE-MAST}}$ are matched against P_{RAVE}, P_{GRAVE} and P_{HRAVE}, respectively. These pairs of agents are tested on all the games with 1000 runs per game.

As a validation of the results obtained in the previous series of experiments, the last series of experiments matches P_{RAVE}, P_{GRAVE} and P_{HRAVE} against each other two at a time and $P_{\text{RAVE-MAST}}$, $P_{\text{GRAVE-MAST}}$ and $P_{\text{HRAVE-MAST}}$ against each other two at a time. A total of at least 1000 runs per game have been played. All the experiments presented in the next sections were performed on a Linux server consisting of 64 AMD Opteron 6274 2.2-GHz cores.

5.3.3 Tuning the Equivalence Parameter K

Table 5.1 shows the performance of P_{RAVE}, P_{GRAVE} and P_{HRAVE} against P_{UCT} for different values of K. For each agent instance, the value of K that achieves the highest robustness is selected to be used in subsequent experiments. The *robustness* of a certain K for an agent instance is computed by summing 1 point for each game in which the instance with such K achieved a statistically significant improvement over P_{UCT} and subtracting 1 point for each game in which it obtained a statistically significant worsening of the performance. In case more values of K have the same robustness, the one with highest average win percentage over all the games is chosen.

For P_{RAVE} none of the values of K reaches the maximum robustness, however, for more than one value the agent instance achieves a statistically significant improvement in all games but one. Among these values, $K = 250$ is chosen because it is the one with the highest average win percentage. For P_{GRAVE} the value $K = 250$ is selected because among the values with highest robustness is also the one with highest average win percentage. Finally, for P_{HRAVE} the value $K = 50$ is selected because it is the only one that reaches the highest robustness.

5.3.4 Matching RAVE Variants against UCT

In this series of experiments, P_{RAVE}, P_{GRAVE} and P_{HRAVE} are matched against P_{UCT}, both with 1s and 10s play-clock. Table 5.2 shows the performance of P_{RAVE}, P_{GRAVE} and P_{HRAVE} with the best K against P_{UCT} with 1s play-clock and start-clock. P_{HRAVE} is the only one that achieves a significant improvement over P_{UCT} in all games, despite not being the one with the highest average win percentage.

Table 5.1: Win percentage of PRAVE, PGRAVE and PHRAVE against PUCT for different values of K.

	Game	$K = 10$	$K = 50$	$K = 100$	$K = 250$	$K = 500$	$K = 750$	$K = 1000$	$K = 2000$
PRAVE vs PUCT	3D Tic Tac Toe	68.7(±4.06)	70.2(±4.00)	72.3(±3.91)	80.3(±3.47)	75.0(±3.78)	81.9(±3.36)	79.5(±3.53)	74.4(±3.81)
	Breakthrough	58.2(±4.33)	60.6(±4.29)	63.8(±4.22)	65.8(±4.16)	72.2(±3.93)	72.0(±3.94)	71.6(±3.96)	65.8(±4.16)
	Knightthrough	68.4(±4.08)	70.8(±3.99)	71.4(±3.96)	73.6(±3.87)	70.6(±4.00)	71.2(±3.97)	71.2(±3.97)	70.8(±3.99)
	Skirmish	64.7(±4.16)	57.1(±4.28)	53.5(±4.34)	49.3(±4.35)	41.2(±4.26)	41.9(±4.27)	40.8(±4.27)	39.7(±4.23)
	Battle	58.0(±3.83)	60.2(±3.78)	54.3(±3.86)	57.2(±3.81)	55.8(±3.88)	58.0(±3.85)	54.0(±3.94)	52.5(±4.00)
	Chinook	45.2(±4.04)	51.5(±4.06)	55.2(±4.10)	59.2(±4.05)	57.2(±4.09)	59.3(±4.01)	55.9(±4.12)	52.4(±4.11)
	Chin.Checkers3P	63.9(±4.20)	61.1(±4.26)	63.9(±4.20)	58.7(±4.30)	64.3(±4.19)	64.9(±4.17)	59.6(±4.28)	58.5(±4.31)
	Avg. Win%	61.0(±1.57)	61.6(±1.56)	62.1(±1.56)	63.4(±1.55)	62.3(±1.56)	**64.2**(±1.54)	61.8(±1.57)	59.2(±1.59)
	Robustness	5	**6**	**6**	**6**	5	5	5	3
PGRAVE vs PUCT	3D Tic Tac Toe	63.5(±4.21)	71.4(±3.96)	75.0(±3.78)	75.3(±3.78)	80.1(±3.49)	80.7(±3.45)	79.9(±3.51)	77.9(±3.61)
	Breakthrough	52.8(±4.38)	58.4(±4.32)	61.2(±4.28)	65.0(±4.19)	67.8(±4.10)	68.6(±4.07)	65.2(±4.18)	62.2(±4.25)
	Knightthrough	72.6(±3.91)	72.0(±3.94)	74.0(±3.85)	71.2(±3.97)	70.6(±4.00)	74.4(±3.83)	68.0(±4.09)	68.6(±4.07)
	Skirmish	62.2(±4.20)	57.0(±4.28)	51.2(±4.34)	55.7(±4.30)	46.1(±4.31)	44.6(±4.27)	42.0(±4.28)	42.2(±4.28)
	Battle	68.7(±3.38)	72.7(±3.33)	71.7(±3.29)	69.6(±3.36)	71.6(±3.31)	72.6(±3.25)	69.6(±3.46)	67.5(±3.46)
	Chinook	55.0(±4.08)	64.4(±4.00)	66.6(±3.89)	67.3(±3.80)	69.6(±3.80)	70.5(±3.70)	66.5(±3.87)	64.2(±3.94)
	Chin.Checkers3P	63.9(±4.20)	67.5(±4.09)	63.3(±4.21)	63.6(±4.20)	60.0(±4.28)	64.9(±4.17)	62.5(±4.23)	57.7(±4.32)
	Avg. Win%	62.7(±1.55)	66.2(±1.52)	66.1(±1.52)	66.8(±1.51)	66.5(±1.51)	**68.0**(±1.49)	64.8(±1.54)	62.9(±1.55)
	Robustness	6	**7**	6	**7**	6	5	5	5
PHRAVE vs PUCT	3D Tic Tac Toe	66.5(±4.12)	63.1(±4.22)	71.9(±3.93)	76.1(±3.74)	76.1(±3.71)	75.4(±3.75)	77.0(±3.67)	68.5(±4.04)
	Breakthrough	53.6(±4.38)	57.0(±4.34)	62.6(±4.25)	65.2(±4.18)	65.4(±4.17)	60.4(±4.29)	63.2(±4.23)	59.2(±4.31)
	Knightthrough	74.0(±3.85)	72.4(±3.92)	74.0(±3.85)	77.4(±3.67)	74.0(±3.85)	73.8(±3.86)	70.4(±4.01)	68.2(±4.09)
	Skirmish	62.6(±4.21)	57.4(±4.31)	52.0(±4.33)	48.9(±4.33)	46.2(±4.32)	41.8(±4.26)	44.2(±4.30)	37.0(±4.18)
	Battle	72.3(±3.21)	75.7(±3.25)	73.2(±3.17)	69.2(±3.37)	70.9(±3.34)	67.4(±3.44)	73.5(±3.26)	69.4(±3.48)
	Chinook	54.9(±4.10)	66.0(±3.89)	66.4(±3.94)	74.9(±3.54)	72.9(±3.58)	75.4(±3.55)	73.4(±3.63)	73.8(±3.61)
	Chin.Checkers3P	67.9(±4.08)	64.1(±4.19)	66.3(±4.13)	65.5(±4.16)	62.7(±4.23)	60.3(±4.28)	61.3(±4.26)	60.2(±4.27)
	Avg. Win%	64.5(±1.53)	65.1(±1.54)	66.6(±1.51)	**68.2**(±1.49)	66.9(±1.51)	64.9(±1.53)	66.1(±1.52)	62.3(±1.56)
	Robustness	6	**7**	6	6	6	5	5	5

Table 5.2: Win percentage of P_{RAVE}, P_{GRAVE} and P_{HRAVE} with best K against P_{UCT} with 1s play-clock and start-clock.

Game	P_{RAVE}	P_{GRAVE}	P_{HRAVE}
3D Tic Tac Toe	**78.4**(±2.54)	74.3(±2.70)	64.0(±2.97)
Breakthrough	66.6(±2.92)	**67.6**(±2.90)	57.8(±3.06)
Knightthrough	73.0(±2.75)	**73.6**(±2.73)	71.3(±2.81)
Skirmish	47.5(±3.07)	54.5(±3.05)	**59.3**(±3.02)
Battle	57.0(±2.69)	69.7(±2.34)	**73.7**(±2.29)
Chinook	59.6(±2.84)	**68.3**(±2.71)	65.1(±2.74)
Chin.Checkers3P	61.9(±3.00)	63.2(±2.98)	**64.4**(±2.96)
Checkers	63.5(±2.83)	**70.8**(±2.65)	60.7(±2.84)
Connect Five	70.8(±2.76)	**75.5**(±2.62)	66.8(±2.89)
Othello	36.9(±2.96)	42.9(±2.99)	**57.4**(±3.02)
Quad	**75.1**(±2.67)	73.6(±2.72)	73.3(±2.73)
Sheep and Wolf	**66.0**(±2.94)	62.9(±3.00)	56.7(±3.07)
TTCC4 2P	**72.9**(±2.73)	71.2(±2.77)	62.3(±3.00)
Zhadu	69.3(±2.86)	67.4(±2.91)	**71.3**(±2.80)
TTCC4 3P	52.1(±3.03)	52.6(±3.03)	**53.4**(±3.05)
Avg. Win%	63.4(±0.75)	**65.9**(±0.74)	63.8(±0.75)
Robustness	11	12	**15**

Table 5.3: Simulations per second of P_{UCT}, P_{RAVE}, P_{GRAVE} and P_{HRAVE}.

Game	P_{UCT}	P_{RAVE}	P_{GRAVE}	P_{HRAVE}
3D Tic Tac Toe	3 093	2 831	2 920	2 877
Breakthrough	1 453	1 378	1 430	1 435
Knightthrough	2 285	2 100	2 146	2 210
Skirmish	106	105	104	106
Battle	2 149	2 001	1 898	1 916
Chinook	2 178	2 085	2 150	2 144
Chin.Checkers3P	4 995	4 108	4 235	4 229
Checkers	532	518	511	518
Connect Five	1 191	1 160	1 144	1 148
Othello	39	39	39	38
Quad	2 767	2 617	2 627	2 684
Sheep and Wolf	2 110	2 071	2 063	2 097
TTCC4 2P	1 124	1 277	1 321	1 368
Zhadu	494	484	477	480
TTCC4 3P	2 058	2 207	2 220	2 257

Table 5.4: Win percentage of P_{RAVE}, P_{GRAVE} and P_{HRAVE} with $K = 250$ against P_{UCT} with 10s play-clock and start-clock.

Game	$\mathbf{P}_{\textbf{RAVE}}$	$\mathbf{P}_{\textbf{GRAVE}}$	$\mathbf{P}_{\textbf{HRAVE}}$
3D Tic Tac Toe	**69.7**(±2.81)	68.2(±2.86)	60.7(±2.99)
Breakthrough	**67.5**(±2.90)	65.1(±2.96)	60.4(±3.03)
Knightthrough	**84.8**(±2.23)	84.1(±2.27)	84.4(±2.25)
Skirmish	**60.2**(±3.01)	60.0(±3.00)	55.8(±3.07)
Battle	**59.1**(±2.19)	57.2(±2.29)	54.6(±2.35)
Chinook	39.8(±2.78)	56.6(±2.84)	**71.8**(±2.52)
Chin.Checkers3P	54.0(±3.08)	**54.7**(±3.07)	49.7(±3.09)
Checkers	52.2(±2.77)	56.3(±2.73)	**61.8**(±2.69)
Connect Five	**66.9**(±2.36)	59.9(±2.50)	53.3(±2.47)
Othello	61.8(±2.97)	**62.0**(±2.97)	60.6(±2.97)
Quad	**10.7**(±1.87)	8.5(±1.68)	7.9(±1.64)
Sheep and Wolf	**69.6**(±2.85)	69.0(±2.87)	67.2(±2.91)
TTCC4 2P	61.1(±2.90)	**66.4**(±2.80)	65.7(±2.80)
Zhadu	63.4(±2.94)	66.5(±2.86)	**68.5**(±2.82)
TTCC4 3P	54.4(±2.97)	**58.5**(±2.95)	50.3(±3.02)
Avg. Win%	58.3(±0.75)	**59.5**(±0.75)	58.2(±0.75)
Robustness	10	**13**	11

P_{RAVE} and P_{GRAVE} still obtain a significant improvement in most of the games, only in Othello they are significantly outperformed by P_{UCT}. Table 5.3 reports for each game the average median number of simulations per second that each of the agent instances can perform. This shows how the overhead, which each strategy introduces, influences the search speed. As can be seen, for most of the games the instances that use a RAVE variant seem to be slightly slower than the instance that uses plain UCT. However, given the results in Table 5.2, the extra information used by these strategies seems to make up for this difference.

Table 5.4 shows the results obtained by repeating the experiment with 10s play-clock and start-clock. For P_{HRAVE}, results for the value $K = 250$ are reported instead of $K = 50$. The value $K = 50$ is the one with highest robustness for P_{HRAVE} in Table 5.1. However, in this series of experiments it makes P_{HRAVE} achieve noticeably lower results than P_{RAVE} and P_{GRAVE}, reaching an average win percentage of only 53.0 and a robustness of 7. For this reason, P_{HRAVE} has been tested also with the value of K that produced the highest average win percentage in Table 5.1 (i.e. $K = 250$), showing that with an increased amount of thinking time, this value performs better.

As can be seen, in most of the games the longer search time reduces the performance increase of P_{RAVE}, P_{GRAVE} and P_{HRAVE} against P_{UCT}. In Quad it even makes the use of RAVE, GRAVE and HRAVE detrimental, substantially reducing the win percentage around 10%. As it will be seen in Chapter 6, this game seems to be particularly sensitive to how parameter values are set. With an increased search time parameter settings for the RAVE variants might become sub-optimal for this

Table 5.5: Win percentage of $P_{\text{RAVE-MAST}}$, $P_{\text{GRAVE-MAST}}$ and $P_{\text{HRAVE-MAST}}$ with best K against $P_{\text{UCT-MAST}}$.

Game	$P_{\text{RAVE-MAST}}$	$P_{\text{GRAVE-MAST}}$	$P_{\text{HRAVE-MAST}}$
3D Tic Tac Toe	64.9(±2.76)	**65.3**(±2.75)	57.3(±2.89)
Breakthrough	**78.5**(±2.55)	74.6(±2.70)	72.3(±2.78)
Knightthrough	**81.9**(±2.39)	74.7(±2.70)	75.6(±2.66)
Skirmish	56.1(±3.04)	53.6(±3.04)	**64.9**(±2.92)
Battle	72.5(±2.32)	76.9(±2.20)	**80.8**(±2.03)
Chinook	32.2(±2.60)	**61.3**(±2.85)	58.3(±2.91)
Chin.Checkers3P	**58.7**(±3.04)	57.5(±3.05)	56.1(±3.07)
Checkers	65.1(±2.80)	**67.1**(±2.74)	59.1(±2.85)
Connect Five	**60.2**(±2.25)	58.4(±2.29)	46.6(±2.41)
Othello	36.8(±2.94)	42.6(±3.00)	**50.1**(±3.06)
Quad	**34.5**(±2.80)	29.2(±2.65)	29.8(±2.67)
Sheep and Wolf	56.3(±3.08)	56.6(±3.07)	**57.3**(±3.07)
TTCC4 2P	63.3(±2.92)	**66.2**(±2.85)	46.6(±3.04)
Zhadu	**73.8**(±2.73)	64.8(±2.96)	65.1(±2.96)
TTCC4 3P	**56.0**(±2.98)	55.6(±2.98)	55.9(±3.01)
Avg. Win%	59.4(±0.75)	**60.3**(±0.75)	58.4(±0.76)
Robustness	9	**11**	8

game, causing the low win rate. Only in Knightthrough, Othello, and Sheep and Wolf more search time increases the performance of all the three RAVE variants with respect to UCT. It could be that these games benefit from more accurate AMAF statistics, for which more samples are collected using more search time.

A reason why in most of the games the difference in performance between RAVE variants and UCT decreases with more search time might be related to the law of diminishing returns (Heinz, 2001). This rule states that the performance gain of a search algorithm decreases with the increase of search effort (e.g. longer search time, higher number of simulations). It is possible that for the agents using RAVE, GRAVE and HRAVE this decrease is faster than for UCT in most of the games, thus more search time closes the gap between the performance of the RAVE variants and UCT. At the same time, the opposite could be true for Knightthrough, Othello, and Sheep and Wolf. For these games UCT might be the strategy for which the performance gain decreases faster when using a longer search time.

5.3.5 Matching RAVE Variants against UCT with MAST

Table 5.5 shows the performance of $P_{\text{RAVE-MAST}}$, $P_{\text{GRAVE-MAST}}$ and $P_{\text{HRAVE-MAST}}$ with the best K against $P_{\text{UCT-MAST}}$. For most of the games the addition of MAST as play-out strategy seems to benefit more $P_{\text{UCT-MAST}}$. The agents $P_{\text{RAVE-MAST}}$, $P_{\text{GRAVE-MAST}}$ and $P_{\text{HRAVE-MAST}}$ perform significantly better than $P_{\text{UCT-MAST}}$ in most of the games. However, for many of these games the difference in performance

Table 5.6: Win percentage of $P_{\text{RAVE-MAST}}$ against P_{RAVE}, $P_{\text{GRAVE-MAST}}$ against P_{GRAVE} and $P_{\text{HRAVE-MAST}}$ against P_{HRAVE}. The win percentage always refers to the first of the two agents.

Game	**$P_{\text{RAVE-MAST}}$ vs P_{RAVE}**	**$P_{\text{GRAVE-MAST}}$ vs P_{GRAVE}**	**$P_{\text{HRAVE-MAST}}$ vs P_{HRAVE}**
3D Tic Tac Toe	51.5(±3.04)	54.3(±3.03)	58.0(±3.00)
Breakthrough	78.1(±2.56)	80.6(±2.45)	82.5(±2.36)
Knightthrough	87.6(±2.04)	87.4(±2.06)	84.4(±2.25)
Skirmish	39.9(±3.00)	44.9(±3.04)	43.4(±3.05)
Battle	15.4(±2.15)	6.5(±1.39)	6.7(±1.38)
Chinook	64.3(±2.47)	79.1(±2.33)	74.6(±2.49)
Chin.Checkers3P	67.5(±2.90)	60.9(±3.02)	60.4(±3.03)
Checkers	74.2(±2.53)	75.8(±2.45)	71.9(±2.63)
Connect Five	75.5(±2.34)	68.2(±2.48)	66.5(±2.68)
Othello	59.6(±2.99)	58.0(±2.99)	63.0(±2.93)
Quad	43.2(±3.03)	38.9(±2.95)	38.2(±2.96)
Sheep and Wolf	37.0(±2.99)	37.3(±3.00)	41.4(±3.05)
TTCC4 2P	83.8(±2.23)	80.7(±2.38)	76.6(±2.60)
Zhadu	64.1(±2.97)	56.1(±3.07)	57.4(±3.06)
TTCC4 3P	60.3(±2.96)	56.9(±2.97)	59.6(±2.96)
Avg. Win%	60.1(±0.76)	59.0(±0.77)	59.0(±0.77)
Robustness	6	7	7

achieved by $P_{\text{RAVE-MAST}}$, $P_{\text{GRAVE-MAST}}$ and $P_{\text{HRAVE-MAST}}$ against $P_{\text{UCT-MAST}}$ is not as high as the difference in performance achieved by P_{RAVE}, P_{GRAVE} and P_{HRAVE} against P_{UCT}. Some examples are the games 3D Tic Tac Toe, Connect Five and TTCC4 with 2 players.

The game for which MAST has the biggest impact is Quad. In this game, all RAVE variants were obtaining an improvement over UCT when a random play-out strategy was used by all agents. Moreover, results for Quad reported in Appendix D show that the agent using the MAST play-out strategy together with UCT achieves a much higher win rate than the UCT agent that uses random play-outs. However, when the RAVE variants are combined with MAST the performance in Quad drops significantly with respect to using only MAST. A reason for this could be, once again, the fact that Quad is sensitive to parameter values. The combination of the MAST play-out strategy with the RAVE variants might require different values for the parameter settings of such selection strategies, therefore making the default values sub-optimal. Another reason might be that, as it will be seen in Chapter 7, Quad is the game that seems to benefit the most from the diversification of the search. Combining selection and play-out strategies that try to bias the search mostly towards generally good actions might make the search too focused for this game.

Among the RAVE variants, the one that seems to get more benefit from being combined with MAST is RAVE. This could be explained by considering that the AMAF scores used by RAVE in the nodes with a low number of visits only have a small number of samples. MAST can compensate the lack of local information near the leaf nodes of the tree. Using its global statistics, MAST steers the simulations towards more promising parts of the state space during the play-out improving its quality. The quality of a simulation for GRAVE and HRAVE, instead, is already improved near the leaf nodes by the use of the AMAF statistics of an ancestor. Thus, the addition of MAST in the play-out has less added benefit to the overall simulation quality.

Although MAST seems to improve more the performance of the UCT selection strategy rather than the one of the RAVE variants, it is still positive to use MAST together with RAVE, GRAVE and HRAVE. This is confirmed by the results shown in Table 5.6, obtained by matching $P_{\text{RAVE-MAST}}$ against P_{RAVE}, $P_{\text{GRAVE-MAST}}$ against P_{GRAVE} and $P_{\text{HRAVE-MAST}}$ against P_{HRAVE}, respectively. For most of the games, the agents that use MAST perform significantly better than the ones that do not. Only in four games MAST is shown to decrease the performance independently of the RAVE variant it is combined with: Skirmish, Battle, Quad, and Sheep and Wolf. For Skirmish, Battle, and Sheep and Wolf also plain UCT, when combined with MAST, shows a decrease in the performance, as visible from the results reported in Appendix D. In Quad, instead, MAST and any of the RAVE variants have a positive impact on the search when used separately, but when combined with each other they are detrimental for the search.

5.3.6 Matching RAVE Variants against Each Other

As a validation of previous results the agents based on the three RAVE variants have been matched two at a time against each other. Table 5.7 shows the obtained results. For each pair of strategies the table reports a column of results without MAST and a column with MAST.

These results are in line to what has been observed in previous experiments. P_{GRAVE} performs better than P_{RAVE} in some games and equally in others. The performance of P_{RAVE} and P_{HRAVE} is more game dependent. In some games they perform equally, in games such as 3D Tic Tac Toe and Breakthrough P_{RAVE} performs best and in games such as Skirmish and Battle P_{HRAVE} performs best. A similar game-dependent performance can be observed for P_{GRAVE} and P_{HRAVE}, but in this case there are more games in which P_{GRAVE} performs best. When MAST is added to all the agents, the difference in their performance diminishes. $P_{\text{GRAVE-MAST}}$ and $P_{\text{RAVE-MAST}}$ perform similarly, one outperforming the other in a few games and vice-versa. MAST also benefits both $P_{\text{RAVE-MAST}}$ and $P_{\text{GRAVE-MAST}}$ against $P_{\text{HRAVE-MAST}}$.

Finally, the results obtained for Knightthrough by P_{GRAVE} against P_{RAVE} can be compared with the ones reported by Cazenave (2015). It can be noticed that the agent instances evaluated in this section did not achieve the same performance increase. Cazenave (2015) shows that the agent based on GRAVE achieves a win rate of 67.8% against the one based on RAVE when both of them have a limit of 1 000

Table 5.7: Win percentage of all possible combinations of agents with and without MAST. The win percentage always refers to the first of the two agents.

Game	P_{GRAVE} vs P_{RAVE}	$P_{GRAVE\text{-}MAST}$ vs $P_{RAVE\text{-}MAST}$	P_{HRAVE} vs P_{RAVE}	$P_{HRAVE\text{-}MAST}$ vs $P_{RAVE\text{-}MAST}$	P_{GRAVE} vs P_{HRAVE}	$P_{GRAVE\text{-}MAST}$ vs $P_{HRAVE\text{-}MAST}$
3D Tic Tac Toe	50.4(±3.09)	51.1(±2.93)	40.1(±3.01)	41.9(±2.89)	63.0(±2.97)	57.1(±2.91)
Breakthrough	46.9(±3.09)	46.1(±3.09)	38.8(±3.02)	44.6(±3.08)	57.1(±3.07)	54.6(±3.09)
Knightthrough	52.8(±3.10)	38.7(±3.02)	49.6(±3.10)	40.3(±3.04)	48.7(±3.10)	45.6(±3.09)
Skirmish	52.3(±3.04)	54.0(±3.04)	62.6(±2.97)	59.5(±3.02)	40.7(±3.02)	44.2(±3.01)
Battle	66.8(±2.34)	54.5(±2.65)	68.4(±2.38)	62.6(±2.52)	50.0(±2.46)	48.4(±2.67)
Chinook	58.3(±2.76)	70.1(±2.41)	55.4(±2.76)	70.8(±2.38)	54.0(±2.79)	49.7(±2.84)
Chin.Checkers3P	55.1(±3.07)	50.1(±3.09)	55.3(±3.08)	49.4(±3.10)	46.2(±3.09)	50.1(±3.10)
Checkers	53.4(±2.91)	53.1(±2.91)	46.5(±2.94)	43.0(±2.91)	54.3(±2.90)	58.5(±2.89)
Connect Five	57.9(±2.97)	47.3(±2.19)	51.0(±3.04)	39.6(±2.17)	58.7(±2.99)	57.8(±2.22)
Othello	52.8(±3.04)	54.5(±3.04)	65.0(±2.91)	65.0(±2.90)	37.5(±2.95)	36.4(±2.93)
Quad	50.5(±3.09)	42.4(±2.90)	48.9(±3.07)	44.4(±2.89)	52.5(±3.06)	53.4(±2.93)
Sheep and Wolf	51.0(±3.10)	49.2(±3.10)	43.8(±3.08)	48.8(±3.10)	57.2(±3.07)	49.8(±3.10)
TTCC4 2P	52.3(±3.03)	54.2(±2.96)	43.7(±3.03)	37.7(±2.92)	60.9(±2.96)	66.0(±2.85)
Zhadu	50.1(±3.10)	42.0(±3.06)	52.3(±3.10)	40.2(±3.04)	46.7(±3.09)	49.5(±3.10)
TTCC4 3P	48.6(±3.02)	50.0(±2.98)	52.8(±3.04)	50.4(±3.01)	48.7(±3.03)	48.5(±3.01)
Avg. Win%	53.3(±0.78)	50.5(±0.76)	51.6(±0.78)	49.2(±0.76)	51.7(±0.78)	51.3(±0.76)
Robustness	4	1	0	−4	3	3

Table 5.8: Average number of move statistics per node of Prave and Pgrave.

Game	PRAVE	PGRAVE
3D Tic Tac Toe	4.58	9.11
Breakthrough	3.49	21.14
Knightthrough	3.16	13.44
Skirmish	4.02	54.38
Battle	8.40	19.92
Chinook	2.46	13.81
Chin.Checkers3P	2.59	16.01
Checkers	2.58	41.94
Connect Five	4.47	9.69
Othello	2.41	14.87
Quad	3.66	7.57
Sheep and Wolf	2.95	32.04
TTCC4 2P	2.27	28.82
Zhadu	2.73	23.12
TTCC4 3P	2.47	13.32

simulations per turn, and a win rate of 67.2% when the limit is 10 000 simulations per turn. However, this difference might be due to the different formula that is used for β and to the fact that in the experiments performed in this chapter agents do not have a limit on the number of simulations per turn but on the amount of time. Moreover, the implementation of RAVE presented in this chapter is achieving a higher win rate against UCT than in the experiments performed by Cazenave (2015), where the win rate of RAVE is 69.4% for 1,000 simulations per turn and 56.2% for 10 000. This, therefore, reduces the potential gain by GRAVE in the experiments performed in this chapter.

5.3.7 Memory Usage

As mentioned in Subsection 5.2.2, GRAVE needs to memorize in each node the AMAF statistics for all the actions that are encountered during every simulation that passes through the node. The RAVE strategy, instead, only needs to memorize in each node the AMAF statistics for the moves that are legal in the corresponding game state.

Table 5.8 shows for RAVE and GRAVE the average number of distinct AMAF statistics that are memorized in each node for every game. These results give an idea of the difference between the strategies in memory usage. The space required by GRAVE ranges between 2 times the space required by RAVE in 3D Tic Tac Toe and 16 times the space required by RAVE in Checkers. The number of AMAF statistics memorized by RAVE is proportional to the branching factor of the search tree. For GRAVE, instead, it depends also on whether or not distinct actions are legal at different stages of the game. If the descendant of a node are often presenting

different legal moves than the ones available in the node, then the space required to memorize AMAF statistics for GRAVE will be much higher than the one required by RAVE.

Note that, given that the strategies are remembering all the statistics between turns, the memory usage of HRAVE is the same of GRAVE. As mentioned in Subsection 5.2.3, HRAVE requires the same statistics as GRAVE in order to reuse them in subsequent turns of the search.

5.4 Chapter Conclusions and Future Research

In this chapter the performance of three selection strategies, RAVE, GRAVE and HRAVE was compared. These three strategies bias the selection phase of MCTS towards actions that perform well in general in the game. The difference is that RAVE uses more local information about the performance of the actions, HRAVE uses more global information, and GRAVE uses information in-between. These RAVE variants have been tested in order to verify the effect of using more local or more global information to bias action selection in MCTS, when MCTS is applied to GGP. Moreover, they have been tested also in combination with the MAST play-out strategy to verify how a more informed play-out strategy affects their performance.

When combined with a random play-out strategy, it may be concluded that the performance of GRAVE is, in the worst case, comparable with the one of RAVE both when using 1s or 10s play-clock. GRAVE never has an inferior performance than RAVE, except in Connect Five when using a 10s play-clock. Regarding HRAVE, it may be concluded that its performance is more game dependent when a random play-out strategy is used. In some games HRAVE is either better or comparable to RAVE and GRAVE, but there are some games where it performs worse. Moreover, when looking at the average win percentage, in none of the experiments its overall performance proved to be better than both RAVE and GRAVE.

When combined with the MAST play-out strategy, GRAVE still seems to be overall better than RAVE. However, it does not have the same advantage over RAVE that it has when both strategies are combined with the random play-out. MAST, apparently, compensates the lack of information near the leaf nodes for RAVE, closing the performance gap between RAVE and GRAVE. There are also a few games where the combination GRAVE-MAST actually performs worse than RAVE-MAST. Moreover, when using MAST, HRAVE is the strategy that appears to be the least beneficial among the three strategies. However, experiments confirm that combining RAVE, GRAVE and HRAVE with the MAST play-out strategy has more benefit than combining them with a random play-out. Therefore, subsequent chapters will consider an agent that uses MAST as play-out strategy.

As seen in the experiments, the difference in performance between RAVE, GRAVE and HRAVE is not large. However, in a context like GGP, GRAVE might be the most suitable strategy to use. In all the experiments its overall win percentage is never inferior to the one of RAVE and HRAVE. Moreover, GRAVE shows to be robust over all the performed series of experiments, and it is important for a strategy in GGP to be robust over a heterogeneous set of games. Therefore, it may be con-

cluded that a strategy that starts biasing action selection with global information and uses more local information the more the nodes have been visited is the most suitable to enhance MCTS for GGP.

Future research could investigate further the strengths of GRAVE over RAVE and HRAVE by tuning also its *ref* parameter. Moreover, the formula proposed more recently by Gelly and Silver (2011) to compute the β parameter could be tested. According to their findings, with this formula the performance of the three RAVE variants could improve further. Moreover, in this chapter these strategies were only tested in combination with MAST. Other play-out strategies might influence them in a different way. Testing the combination with the NST play-out strategy could be an idea for future research.

Chapter 6

On-line Search-Control Parameter Tuning for MCTS

This chapter is based on:

- Sironi, Chiara F., and Winands, Mark H.M. (2018a). On-Line Parameter Tuning for Monte-Carlo Tree Search in General Game Playing. *Computer Games*, pp. 75–95.
- Sironi, Chiara F. and Liu, Jialin and Perez-Liebana, Diego and Gaina, Raluca D. and Bravi, Ivan and Lucas, Simon M. and Winands, Mark H.M. (2018). Self-Adaptive MCTS for General Video Game Playing. *21st International Conference on the Applications of Evolutionary Computation* (eds. K. Sim and P. Kaufmann), Vol. 10784 of LNCS, pp. 358–375, Springer, Berlin, Germany.
- Sironi, Chiara F., and Winands, Mark H.M. (2018b). Analysis of Self-Adaptive Monte Carlo Tree Search in General Video Game Playing. *Computational Intelligence and Games (CIG), 2018 IEEE Conference on*, pp. 397–400.
- Sironi, Chiara F., and Winands, Mark H.M. (2018c). On-Line Parameter Tuning for Monte-Carlo Tree Search in General Game Playing. *Proceedings of the 30th Benelux Conference on Artificial Intelligence*, pp. 235–236. Extended Abstract.
- Sironi, Chiara F., Liu, Jialin, and Winands, Mark H.M. (2019). Self-Adaptive Monte-Carlo Tree Search in General Game Playing. *IEEE Transactions on Games*. In press.

Search-control strategies and enhancements have been proposed for MCTS in various domains (Browne *et al.*, 2012) and many have successfully been applied in GGP (Finnsson and Björnsson, 2010; Tak *et al.*, 2012; Soemers *et al.*, 2016). The behavior of MCTS strategies is normally controlled by a certain number of parameters, and the performance of such strategies depends on how parameter values are set. Sometimes, extensive off-line tuning is required to find the best values.

Parameters might also be inter-constrained, so either a large amount of time is spent testing all possible combinations of values or the parameters are tuned separately ignoring the inter-dependency. Research has also shown that in some cases the best values for the parameters could be game dependent (Cazenave, 2015; Nijssen and Winands, 2011; Tak *et al.*, 2014a) and tuning parameters per game might improve the performance of MCTS. An example of this has been seen also in Chapter 5, where for each RAVE variant no value of the parameter K was clearly superior to all the others, but the performance depended on the game.

In the context of GGP, off-line tuning of parameters per game is infeasible because agents have to deal with a theoretically unlimited number of games, treating each of them as a new game that they have never seen before. This is why off-line parameter tuning in GGP usually looks for a single combination of values to use for all games, picking the one that performs overall best on a certain (preferably heterogeneous) set of benchmark games. Tuning search-control parameters for each game in GGP is still possible if done on-line. A Self-Adaptive MCTS (SA-MCTS) can be designed by devising an on-line tuning method that adjusts the parameter values for each new game being played. Such method should also aim at tuning the parameters in combination, because parameter values are usually interdependent. In this case, the number of possible combinations of parameters can become very large. Therefore, an efficient strategy has to be designed to decide how to allocate the available samples to test them.

This chapter answers the third research question by proposing a method that tunes search-control parameters on-line for MCTS. On-line parameter tuning is interleaved with the search, achieving a self-adaptive MCTS strategy (SA-MCTS). This on-line tuning method uses each MCTS simulation to evaluate a different combination of parameter values. An allocation strategy is required to decide how many simulations must be used to evaluate each combination of values. This chapter presents and evaluates seven allocation strategies: Multi-Armed Bandit (MAB) allocation (Auer *et al.*, 2002; Ontañón, 2013), Hierarchical Expansion (HE) (Roelofs, 2015; Roelofs, 2017), Naïve Monte-Carlo (NMC) (Ontañón, 2013; Ontañón, 2017), Linear Side Information (LSI) (Shleyfman, Komenda, and Domshlak, 2014), an Evolutionary Algorithm (EA), the N-Tuple Bandit Evolutionary Algorithm (NTBEA) (Kunanusont *et al.*, 2017; Lucas, Liu, and Perez-Liebana, 2018) and the Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) (Hansen, Müller, and Koumoutsakos, 2003). The first six allocation strategies consider a discrete domain for parameter values, while the last one considers a continuous domain. This chapter evaluates the robustness of the allocation strategies by testing them on MCTS-based agents with different skills, on an increasing number of tuned parameters, for different time constraints and in a real-time domain.

The remainder of this chapter is structured as follows. First, Section 6.1 introduces previous work related to parameter tuning. Next, how to design a SA-MCTS algorithm using on-line parameter tuning and how to formulate the tuning problem is discussed in Section 6.2. Subsequently, Section 6.3 describes the seven allocation strategies. Results obtained by testing these strategies in the context of GGP are presented in Section 6.4, and finally, Section 6.5 gives the conclusion and outlines possible future research.

6.1 Related Work

In the research area of game playing, on-line tuning of search parameters has not been well explored. An example of work in this direction is performed by Björnsson and Marsland (2003). They present both an off-line and an on-line approach for learning search-extension parameters for $\alpha\beta$-search. The method is based on gradient descent and looks for the parameter values that minimize the growth rate (in the number of visited nodes) of the search.

More attention has been given to automatic off-line tuning of search parameters. An example is the work of Kocsis, Szepesvári, and Winands (2006b), which uses an enhanced version of the Simultaneous Perturbation Stochastic Approximation (SPSA) algorithm to tune parameters for Poker and Lines of Action. Another approach, which recalls the structure of evolutionary approaches, is presented by Chaslot *et al.* (2008c), which propose to tune parameters for the game of Go using the Cross-Entropy Method (CEM). This method keeps a probability distribution over possibly good parameter values and uses the evaluation of samples drawn from it to refine the distribution over time. A genetic algorithm is the solution proposed by Cole, Louis, and Miles (2004) to tune the parameters that control the behavior of a rule-based agent for a first-person shooter game. CLOP (Coulom, 2012) is another algorithm proposed for tuning game parameters and is based on local quadratic regression. What most of these methods have in common is the need for a high number of samples (e.g. game simulations) against a benchmark player in order to find an optimal parameter configuration, which could then be evaluated by playing against an "identical" agent with manually tuned parameters.

Other research focuses on designing Hyper-Heuristics, which are *"a search method or learning mechanism for selecting or generating heuristics to solve computational search problems"* (Burke *et al.*, 2013). This concept has been applied to devise a hyper-agent (Mendes, Togelius, and Nealen, 2016) for the GVG-AI framework. The agent's hyper-heuristic is trained off-line to recognize from a portfolio of sub-agents the best one for the game at hand. The hyper-heuristic approach presented in (Świechowski and Mańdziuk, 2014), instead, devises an on-line mechanism to select from a portfolio of strategies the one that is best suited for the current game. A similar concept is *algorithm selection* (Bischl *et al.*, 2016), which consists in choosing the most appropriate algorithm for the instance at hand, given a set of problem instances and a set of algorithms that present a varying performance depending on the instance they are applied to.

The research presented in this chapter is similar to the idea of hyper-heuristic and algorithm selection. Some parameters can decide whether to (de)activate a certain search-control strategy depending on the value that is assigned to them, actually originating multiple search algorithms. Parameter tuning can be seen as a hyper-heuristic to choose which strategies or algorithms to use from a portfolio determined by the available parameter configurations.

Moreover, three of the allocation strategies discussed in this chapter are based on evolutionary algorithms. Evolutionary computation (Ashlock, 2006) is a set of nature-inspired optimization algorithms. In general, a population of individuals (candidate solutions) is randomly initialized when the optimization process starts,

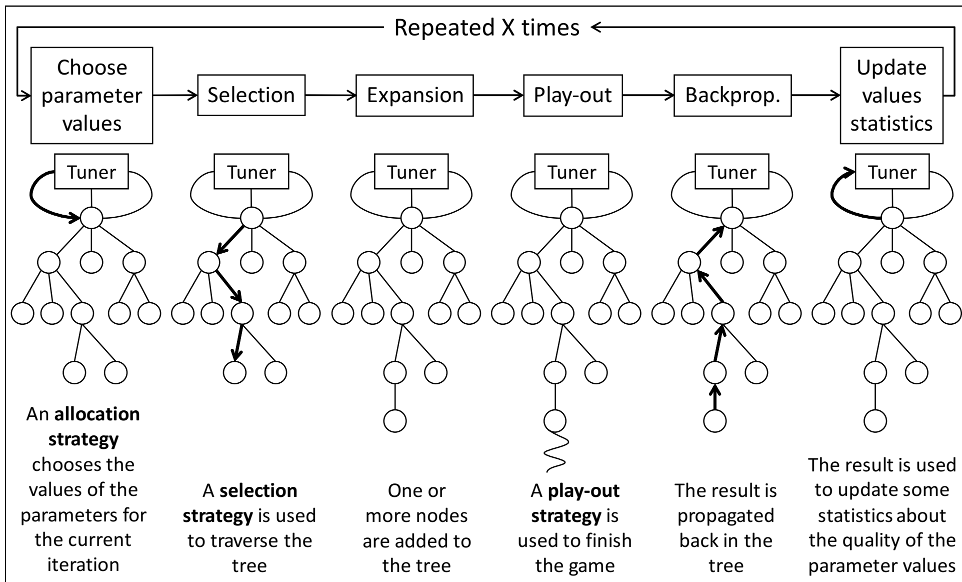


Figure 6.1: Interleaving on-line tuning with MCTS (Inspired by Chaslot *et al.*, 2008b).

then updated iteratively by evaluation, selection and reproduction until stopping conditions are met. The population can be composed by one or multiple individuals. Evolutionary computation techniques can be classified by different ways of reproduction, the size of the population, the use of a discrete or continuous search space, etc. The previously mentioned works of Chaslot *et al.* (2008c) and Cole *et al.* (2004) are examples of applications of evolutionary computation techniques to game playing. Other applications to game playing focus on evolving sequences of actions to play (Gaina *et al.*, 2017), or evolving game parameters or game levels (Liu *et al.*, 2017; Kunanusont *et al.*, 2017).

6.2 Design of Self-Adaptive MCTS

This section presents the two main aspects of the proposed SA-MCTS. First, Subsection 6.2.1 discusses how on-line parameter tuning can be integrated within the MCTS algorithm to make it adaptive. Next, Subsection 6.2.2 presents the formulation of the parameter-tuning problem.

6.2.1 Integration of Parameter Tuning with MCTS

Figure 6.1 shows how parameter tuning is interleaved with MCTS simulations to be performed on-line. For each iteration of the algorithm a tuner uses an *allocation strategy* to choose a combination of values for the parameters being tuned. Next, the four standard phases of MCTS are performed using the selected parameter values to control the search. The result obtained by the simulation is used to update statistics

about the quality of the chosen parameter combination. This process continues until the end of the game, such that the parameters are tuned on-line for the whole duration of the search.

In two- or multi-player games two possible ways to set up on-line parameter tuning were identified. The first consists in selecting before each MCTS simulation a single combination of parameter values for all the roles in the game, and using the payoff obtained by the role played by the agent to update the statistics of such combination. The second consists in separating the on-line parameter tuning problem into multiple instances, one for each role. Therefore, a different combination of parameter values is selected before each MCTS simulation for each role in the game independently, and the statistics of each combination are updated with the payoff of the role for which the combination controlled the search. The advantage of using the second option is that a different model for each role can be tuned, possibly obtaining a more efficient search. Different roles might benefit from searching with different strategies instead of using all the same. For this reason, this is the solution that this chapter investigates.

6.2.2 Formulation of the Parameter Tuning Problem

An allocation strategy is required to decide how to divide the available number of samples among all the combinations of parameter values that have to be evaluated. An ideal allocation strategy for the on-line parameter tuning problem should aim to assign most of the samples to good value combinations, reducing the number of samples assigned to bad value combinations. This is because each evaluated combination has an impact on the quality of the actual search. If bad combinations are evaluated too often, the quality of the search results will decrease.

The main idea behind the formulation of the tuning problem is based on the work of Świechowski and Mańdziuk (2014). They discusses multiple allocation strategies for a problem similar to the one tackled in this chapter: on-line adaptation of the search strategy to the played game. Among all the approaches they show that the one considering the simulation allocation as a MAB problem combined with UCB selection is the one that assigns the highest number of samples to the best search strategy and the lowest to the worst. Moreover, a bandit-based approach is also used by Baier (2015) to perform automated off-line parameter tuning for search-based game-playing agents. An UCB1 variant is used to decide how to distribute test games to evaluate different parameter settings, so that the number of tests spent on inferior settings is reduced.

The action-space of the on-line parameter tuning problem has a combinatorial structure (i.e. the action of choosing a parameter setting consists of multiple sub-actions, each of which assign a certain value to one of the parameters). For this reason, instead of considering the tuning problem as a MAB, this chapter considers it as a Combinatorial Multi-Armed Bandit (CMAB), either with discrete or continuous variables, and bases the design of the allocation strategies on this formulation.

The CMAB with discrete variables is defined by the following three components:

- Set of d variables, $\mathcal{P} = \{P_1, ..., P_d\}$, where each variable P_i can take m_i different values $\mathcal{V}_i = \{v_{i,1}, ..., v_{i,m_i}\}$.

- Payoff distribution $Q : \mathcal{V}_1 \times ... \times \mathcal{V}_d \rightarrow \mathbb{R}$ that depends on the combination of values assigned to the variables.
- Function $LEGAL : \mathcal{V}_1 \times ... \times \mathcal{V}_d \rightarrow \{true, false\}$ that determines which combinations of values are legal.

Analogously, the CMAB with continuous variables can be defined by the following three components:

- Set of d variables, $\mathcal{P} = \{P_1, ..., P_d\}$, where each variable P_i can take a value in the interval $I_i = [min_{P_i}, max_{P_i}]$.
- Payoff distribution $Q : I_1 \times ... \times I_d \rightarrow \mathbb{R}$ that depends on the combination of values assigned to the variables.
- Function $LEGAL : I_1 \times ... \times I_d \rightarrow \{true, false\}$ that determines which combinations of values are legal.

For the parameter tuning problem, the parameters are considered as the variables of the CMAB.

6.3 Allocation Strategies

This section introduces seven allocation strategies: Multi-Armed Bandit (MAB) allocation (Auer *et al.*, 2002; Ontañón, 2013), Hierarchical Expansion (HE) (Roelofs, 2015; Roelofs, 2017), Naïve Monte-Carlo (NMC) (Ontañón, 2013; Ontañón, 2017), Linear Side Information (LSI) (Shleyfman *et al.*, 2014), an Evolutionary Algorithm (EA), the N-Tuple Bandit Evolutionary Algorithm (NTBEA) (Kunanusont *et al.*, 2017; Lucas *et al.*, 2018) and the Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) (Hansen *et al.*, 2003). The first six consider a discrete domain for the parameter values and are introduced in Subsection 6.3.1, while the latter considers a continuous domain and is introduced in Subsection 6.3.2.

When reporting the pseudocode of the different allocation strategies, it is assumed that each of them has access to the set of parameters being tuned P, the set of feasible values $\mathcal{V}_i$ for each parameter P_i and the legality function $LEGAL$. For the sake of simplicity, the pseudocode is given for one-player games, assuming that each allocation strategy implicitly ensures that no illegal combinations of parameter values are generated.

When tuning parameters for two- or multi-player games, all the allocation strategies compute a different combination of parameters for each role in the game independently, therefore considering a separate CMAB representation of the tuning problem for each role. All the computed combinations of parameters are then used to control the same MCTS simulation. In this way, parameter value combinations are co-evolved for all the roles. Note that having a different parameter combination for each role means that during the MCTS simulation different instantiations of the selection or the play-out strategies might be used to choose the actions for the different roles.

```
1: procedure MabParameterTuning(π_MAB)
   Input: The policy π_MAB to select an action from the MAB problem.
2:     MAB ← MAB problem with an arm for each legal parameter combination
3:     while game not over do
4:         p⃗ ← π_MAB.ChooseCombination(MAB)
5:         q ← PerformMctsSimulation(p⃗)
6:         MAB.UpdateArmStatistics(p⃗, q)
```

Algorithm 9: Pseudocode for the MAB allocation strategy.

When generating combinations of parameter values to be evaluated, the following is the general procedure used by the allocation strategies to ensure that they are all legal. When the allocation strategy is selecting a combination of parameter values all at once, all the illegal combinations are excluded in advance from the selection. When the allocation strategy is selecting the value for one parameter at a time, the check for illegal values is split into multiple steps. More precisely, given the partial combination of values selected so far and the next parameter to be set, all the values of such parameter that will make the partial combination illegal are excluded from the selection.

6.3.1 Discrete Allocation Strategies

This section presents the six allocation strategies that consider the tuning problem for a player as a CMAB with a discrete domain. The first strategy has already been proposed to be applied to MAB problems, to which CMAB problems can be reduced. The next three strategies, HE, NMC and LSI, have been proposed by previous research to specifically deal with CMABs. EA and NTBEA, being based on evolutionary computation, have not been specifically designed for CMABs, but can still be applied in this context.

Multi-Armed Bandit Allocation

A straightforward solution for dealing with a CMAB problem is to translate it back to a MAB (Auer *et al.*, 2002; Ontañón, 2013), where each arm corresponds to a possible legal combination of values for the parameters. Algorithm 9 shows the pseudocode for the MAB allocation strategy. This strategy uses a policy π_{MAB} to select the next parameter combination $\vec{p}$ to sample from the MAB problem (MAB). The parameter combination is used to control the next MCTS simulation, performed by the procedure PerformMctsSimulation($\vec{p}$), and the result of the simulation is used to update the expected value of $\vec{p}$ in the MAB. The procedure PerformMctsSimulation($\vec{p}$), other than running an MCTS simulation with the parameters $\vec{p}$ and implementing the four MCTS phases, takes care of checking the search budget for each game turn and playing a move in the real game when this budget expires. This procedure is the same for all the allocation strategies discussed in this section.

When using the MAB allocation strategy, however, the information on the combinatorial structure of the parameter values is lost. Often, a value that is good (or

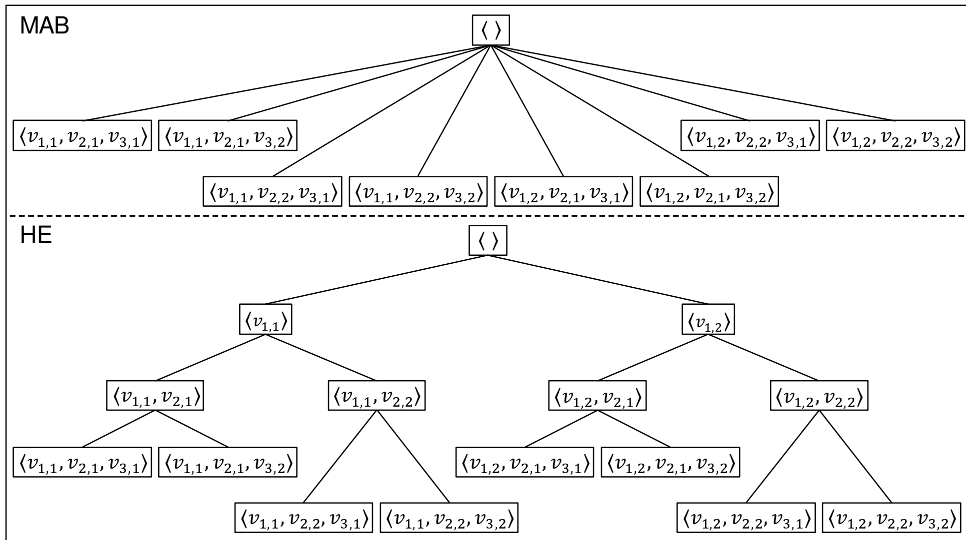


Figure 6.2: MAB and HE representation of the combinatorial action-space of the parameter tuning problem.

bad) for a parameter in a certain combination of values, is also good (or bad) in general or in many other combinations. With a MAB this information is ignored and cannot be exploited.

Hierarchical Expansion

The HE strategy was proposed by Roelofs (2015; 2017) to offer an alternative to the representation of a combinatorial space with respect to the one used by the MAB problem. The main idea behind HE is to represent the combinatorial parameter space as a tree, where each level corresponds to a different parameter for which a value must be selected. Each node in a level has one child for each available legal value of the corresponding parameter. In this way, the depth of the search for a combination of parameters is increased, but the branching factor is reduced.

Figure 6.2 shows how the problem of tuning 3 parameters, each with 2 possible values, is represented both with the MAB and with the HE approach. To build the parameter tree used by HE an order must first be imposed on the parameters. Each node of the tree corresponds to a partial combination of parameter values (the root corresponds to the empty combination). At every level of the tree the partial combination is extended by assigning a value to the next parameter in the order, until the combination is complete. The parameter tree is then used to sample a combination of parameters for each game simulation. MCTS can be applied to the parameter tree, reducing to a MAB problem the choice of the next value to add to the combination.

Algorithm 10 gives the pseudocode for the HE allocation strategy. The procedure HeParameterTuning(π_{HE}) implements the structure discussed in Subsection

```
1: procedure HEPARAMETERTUNING(π_HE)
   Input: The policy π_HE to select an action at each node of the parameter tree.
2:     paramTree ← parameter tree with one level per parameter
3:     while game not over do
4:         p⃗ ← CHOOSEPARAMETERVALUES(paramTree, π_HE)
5:         q ← PERFORMMCTSSIMULATION(p⃗)
6:         UPDATESTATISTICS(paramTree, p⃗, q)

7: procedure CHOOSEPARAMETERVALUES(paramTree, π_HE)
   Input: Tree, paramTree, where each level corresponds to one of the tunable
   parameters, and the policy π_HE to select an action at each node of this tree.
   Output: Combination of parameter values p⃗ = ⟨p_1, ..., p_d⟩.
8:     p⃗ = ⟨p_1, ..., p_d⟩ ← empty array of size d
9:     node ← paramTree.GETROOT( )
10:    for i ← 1, ..., d do
11:        p_i ← π_HE.CHOOSEVALUE(node)
12:        node ← paramTree.GETNEXTNODE(node, p_i)
13:    return p⃗

14: procedure UPDATESTATISTICS(paramTree, p⃗, q)
   Input: Tree, paramTree, where each level corresponds to one of the tunable
   parameters, chosen parameter values p⃗, payoff q obtained from the simulation
   controlled by parameter values p⃗.
15:    node ← paramTree.GETROOT( )
16:    for i ← 1, ..., d do
17:        node.UPDATEARMSTATISTICS(p_i, q)
18:        node ← paramTree.GETNEXTNODE(node, p_i)
```

Algorithm 10: Pseudocode for the HE allocation strategy.

6.2.1. First of all, this procedure requires the tree, $paramTree$, that represents the search space of the parameters and is built as discussed previously. A combination of parameters $\vec{p}$ is chosen from the tree and used to control an MCTS simulation. The result of the simulation is then used to update the statistics in the tree for the selected combination of parameters.

The procedure ChooseParameterValues($paramTree, \pi_{\text{HE}}$) selects a parameter combination $\vec{p}$ by visiting the tree representation of the parameter-space. Starting from the root, the procedure visits one path in the tree. In each visited node a policy π_{HE} is used to select the value for the next parameter, and this value is added to the partial combination computed so far. The procedure UpdateStatistics($paramTree, \vec{p}, q$) propagates in the tree the payoff q obtained by the game simulation controlled by $\vec{p}$.

```
1:  procedure NmcParameterTuning(π_0, π_l, π_g)
    Input: The policy π_0 to select among performing the exploration and exploita-
    tion step, the policy π_l to select an action from the local MAB problems, the
    policy π_g to select an action from the global MAB problem.
2:      MAB_g ← MAB problem with no arms
3:      for i ← 1, ..., d do
4:          MAB_i ← MAB problem with an arm for each legal value of parameter
    P_i
5:      while game not over do
6:          p⃗ ← ChooseParameterValues(π_0, π_l, π_g)
7:          q ← PerformMctsSimulation(p⃗)
8:          UpdateStatistics(p⃗, q)

9:  procedure ChooseParameterValues(π_0, π_l, π_g)
    Output: Combination of parameter values p⃗ = ⟨p_1, ..., p_d⟩.
10:     phase ← π_0.choosePhase( )
11:     if phase = exploration then                      ▷ Generate combination
12:         p⃗ = ⟨p_1, . . . , p_d⟩ ← empty array of size d
13:         for i ← 1, ..., d do
14:             p_i ← π_l.ChooseValue(MAB_i)
15:         MAB_g.Add(p⃗)
16:     else if phase = exploitation then                ▷ Evaluate combination
17:         p⃗ ← π_g.ChooseCombination(MAB_g)
18:     return p⃗

19: procedure UpdateStatistics(p⃗, q)
    Input: Chosen parameter values p⃗, payoff q obtained from the simulation con-
    trolled by parameter values p⃗.
20:     MAB_g.UpdateArmStatistics(p⃗, q)
21:     for i ← 1, ..., d do
22:         MAB_i.UpdateArmStatistics(p_i, q)
```

Algorithm 11: Pseudocode for the NMC allocation strategy.

Naïve Monte-Carlo

This strategy was first proposed by Ontañón (2013; 2017) and applied to real-time strategy games, which are known for having a combinatorial action-space. Algorithm 11 shows the pseudocode for NMC. The procedure NmcParameterTuning(π_0, π_l, π_g) implements the structure discussed in Subsection 6.2.1. The procedure ChooseParameterValues(π_0, π_l, π_g) shows how NMC chooses the combination of parameter values to test before an MCTS simulation. Two main steps are distinguished, the *exploration*, which generates new parameter combinations, and the *exploitation*, which evaluates the combinations generated so far. These two steps are interleaved and for each iteration a policy π_0 chooses which one to perform. Figure

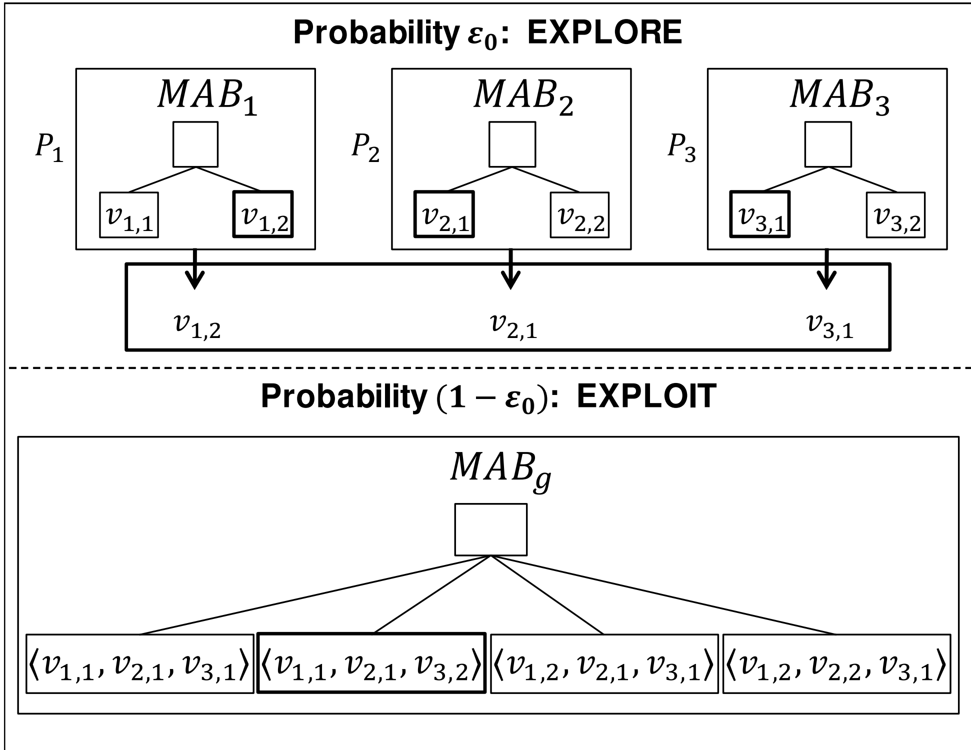


Figure 6.3: Overview of the *exploration* and *exploitation* phases of NMC.

6.3 gives an overview of the two phases of NMC using as an example the problem of tuning three parameters, P_1, P_2 and P_3, each of which can assume two different values. The *exploration* phase of NMC is based on the *naïve assumption*, shown in Formula 6.1.

$$Q(\vec{p} = \langle p_1, ..., p_d \rangle) \approx \sum_{i=1}^{d} Q_i(p_i). \tag{6.1}$$

Here, $\vec{p}$ is a vector representing a possible assignment of values $\langle p_1, ..., p_d \rangle$ to the parameters. This assumption means that the expected payoff of a certain configuration of parameter values can be approximated by a linear combination of expected payoffs of single parameter values, as if considering the parameters to be independent. More precisely, the *exploration* considers d local MABs, one per parameter, and uses them independently to generate a new combination of parameter values. Each local MAB has an arm for each possible value of the associated parameter. A policy π_l is used to select one value p_i for each parameter P_i using the corresponding local MAB (i.e. MAB_i). The resulting combination of values, if not yet present, is added to the global MAB (i.e. MAB_g) used during the *exploitation*. MAB_g considers each arm to be associated to a possible parameter combination. Initially it has no arms and

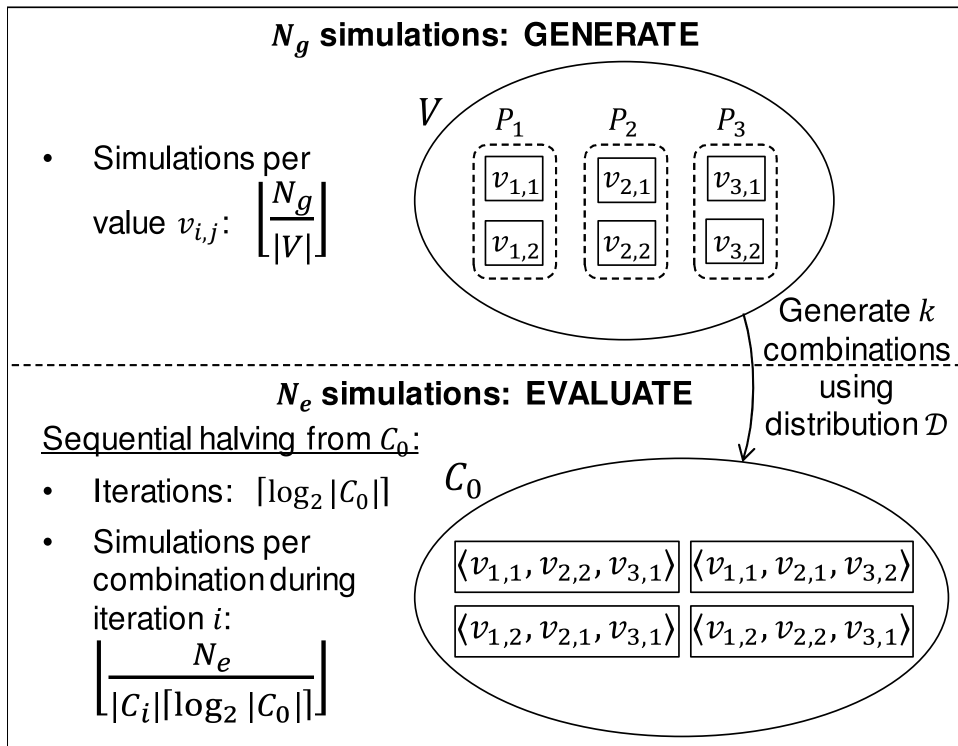


Figure 6.4: Overview of the *generation* and *evaluation* phases of LSI.

is filled during *exploration*. The *evaluation* uses a policy π_g to select from MAB_g a parameter combination to evaluate.

The procedure UPDATESTATISTICS($\vec{p}, q$) shows how the payoff of the MCTS simulation is used to update statistics about the chosen parameter values. Statistics are updated in the global MAB for the given combination and in the local MABs for each value in the combination.

Linear Side Information

The LSI algorithm (Shleyfman *et al.*, 2014) is similar to NMC. It distinguishes two main steps, called *generation* and *evaluation*. The *generation* step, like the *exploration* step of NMC, generates new combinations of parameter values considering them as independent (i.e. making a *naïve assumption*), while the *evaluation* step, like the *exploitation* step of NMC, evaluates the generated combinations. The main difference with NMC is that LSI performs these two steps in sequence instead of interleaving them, and a total predefined budget of available samples $N_{tot} = N_g + N_e$ is divided among them.

Algorithm 12 gives the pseudocode for LSI, while Figure 6.4 gives an overview of the two phases of LSI using as example the problem of tuning three parameters,

```
1: procedure LsiParameterTuning(N_g, N_e, k)
   Input: Number of samples N_g for the generation phase, number of samples N_e for the evaluation phase, number of candidates k to generate.
2:     C* ← Generate(N_g, k)
3:     p⃗* ← Evaluate(C*, N_e)
4:     while game not over do
5:         PerformMctsSimulation(p⃗*)

6: procedure Generate(N_g, k)
   Input: Number of samples N_g for the generation phase, number of candidates k to generate.
   Output: Set of candidate parameter combinations to evaluate, C*.
7:     Q̂ ← SideInfo(N_g)
8:     C* ← ∅
9:     for k times do
10:        p⃗ = ⟨p_1, ..., p_d⟩ ← empty array of size d
11:        𝒱 ← ⋃_{i=1}^{d} 𝒱_i
12:        while 𝒱 ≠ ∅ do
13:            v_{i,j} ∼ 𝒟[Q̂ ↾𝒱]
14:            𝒱 ← 𝒱 \ 𝒱_i
15:            p_i ← v_{i,j}
16:        C* ← C* ∪ {p⃗}
17:    return C*

18: procedure SideInfo(N_g)
    Input: Number of samples N_g for the generation phase.
    Output: Weight function Q̂ over single parameter values.
19:    𝒱 ← ⋃_{i=1}^{d} 𝒱_i
20:    x ← ⌊ N_g / |𝒱| ⌋
21:    for x times do
22:        for each v_{i,j} ∈ 𝒱 do
23:            p⃗ ← RandomlyExtend(v_{i,j})
24:            q ← PerformMctsSimulation(p⃗)
25:            average Q̂(v_{i,j}) with q
26:    return Q̂

27: procedure Evaluate(C*, N_e)
    Input: Set of parameter combinations to evaluate C*, number of samples N_e for the evaluation phase.
    Output: Best parameter combination.
28:    C_0 ← C*
29:    for i ← 0 to (⌈log_2 |C*|⌉ − 1) do
30:        x ← ⌊ N_e / (|C_i| ⌈log_2 |C*|⌉) ⌋
31:        for x times do
32:            for each p⃗ ∈ C_i do
33:                q ← PerformMctsSimulation(p⃗)
34:                average expected value of p⃗ with q
35:        C_{i+1} ← ⌈|C_i|/2⌉ elements with highest estimated value
36:    return the only combination p⃗ ∈ C_{⌈log_2 |C*|⌉}
```

Algorithm 12: Pseudocode for the LSI allocation strategy.

with two feasible values each. The procedure LsiParameterTuning(N_g, N_e, k) implements the main logic of LSI. The *generation* uses up to N_g samples (i.e. MCTS simulations) to generate a set $C^* \subseteq C = \mathcal{V}_1 \times ... \times \mathcal{V}_d$ of at most k legal combinations of parameters. The *evaluation* uses up to N_e samples to evaluate the combinations of values in C^* and recommend the best one, $\vec{p}^*$. When both phases of LSI are over, the recommended best combination $\vec{p}^*$ is used to control the rest of the MCTS simulations until the game terminates. The PerformMctsSimulation($\vec{p}$) procedure, before returning the control to the LSI procedure to continue the tuning, takes care of playing a move in the real game if the timeout for the current game step is reached.

The procedure SideInfo(N_g) constructs the function $\hat{Q} : \bigcup_{i=1}^{d} \mathcal{V}_i \rightarrow \mathbb{R}$, that associates to each parameter value $v_{i,j}$ the average payoff $\hat{Q}(v_{i,j})$ obtained by all the MCTS simulations that were allocated to $v_{i,j}$. To construct $\hat{Q}$ the procedure divides equally over all the parameter values the total number of generation samples N_g. Each time a parameter value $v_{i,j}$ is sampled using an MCTS simulation the other parameters are set to random values. Note that for multi-player games, when parameters are tuned for all the roles, it is necessary to randomize the order in which values are iterated over in Line 32, to avoid a value for a role to always be tested against the same values for the other roles.

The procedure Generate(N_g, k) uses the function $\hat{Q}$ to generate up to k combinations of parameter values. To do so, the function $\hat{Q}$ is normalized to create a probability distribution over (a subset of) its domain. The notation $\mathcal{D}[\hat{Q} \restriction_{\mathcal{V}}]$ indicates the probability distribution induced by $\hat{Q}$ over the subset $\mathcal{V}$ of its domain. Each combination is generated by repeatedly sampling a value from the distribution $\mathcal{D}[\hat{Q} \restriction_{\mathcal{V}}]$. The first time $\mathcal{V} = \bigcup_{i=1}^{d} \mathcal{V}_i$ (i.e. all the domain). For each subsequent step the set of available values $\mathcal{V}_i$ for the last set parameter P_i is removed from $\mathcal{V}$.

The procedure Evaluate(C^*, N_e) uses *sequential halving* (Karnin, Koren, and Somekh, 2013) to repeatedly evaluate the generated combinations and finally recommend one. *Sequential halving* performs multiple iterations dividing equally among them the available samples N_e. During each iteration the combinations are sampled uniformly and only half of them is kept for the next iteration (the half with the highest expected value). This process ends when only one combination is left. Line 30 shows how the number of simulations per combination is computed for each iteration.

Note that when LSI is used to tune parameters for all the roles in a multi-player game it is necessary to randomize the order in which the parameter combinations of each role are evaluated during one iteration of sequential halving. Randomizing ensures that each combination for a role is not always evaluated against the same combinations for the opponents.

It is important to note that LSI, as opposed to the other allocation strategies, is based on a fixed number of samples N_{tot} that must be set in advance. Choosing a value for N_{tot} is not trivial, if the value is too high the search is likely controlled by parameter values selected randomly, because the game might terminate before LSI reaches the *evaluation* step. On the contrary, if the value is too low the search is likely controlled by a sub-optimal combination, recommended using only a low number of samples. Ideally, N_{tot} should correspond to the total number of available

simulations for the game. Because in GGP is not possible to know this exact number in advance, N_{tot} is estimated during the start-clock according to Formula 6.2.

$$\begin{aligned} N_{tot} &= cps \times playclock \times \overline{turns} \\ cps &= \kappa \times \frac{\#sim}{startclock} \end{aligned} \tag{6.2}$$

First, the average number of parameter combinations that can be evaluated in a second for the game, *cps*, is computed dividing the total number of simulations performed during the start-clock, $\#sim$, by the duration of the start-clock, *startclock*, and multiplying the result by a factor κ. This factor is used to avoid underestimating the actual average number of combinations per second that can be evaluated over all the game. During the start-clock, simulations are longer than on average over all the game, therefore fewer of them can be performed per second. Without using κ, *cps* would be underestimated. Subsequently, *cps* is used to estimate N_{tot} by multiplying it by the estimated number of game turns, $\overline{turns}$ and the duration of the play-clock for each turn, *playclock*. The average number of turns for the game, $\overline{turns}$, is also estimated during the start-clock as the average length of all the performed simulations.

An alternative to deal with the necessity of knowing the available search budget in advance could be to modify LSI to tune parameters per move instead of per game. In this way the known play clock time can be used to estimate the available budget for the tuning. The search time T available for each move can be divided among a generation phase and an evaluation phase, $T = T_g + T_e$, and used to control the execution of the phases of LSI instead of the total simulations budget N_{tot}. However, preliminary results obtained by testing this strategy showed that it does not improve upon the implementation of LSI that estimates N_{tot} during the start-clock.

Evolutionary Algorithm Allocation

A subset of evolutionary computation is Evolutionary Algorithms (EAs) (Ashlock, 2006), which, at each generation, select a subset of individuals from the current population as elite, and reproduce the population using the elite by crossover and mutation. EAs do not rely on any assumption on the fitness function or fitness landscape. For the tuning problem, an EA can be seen as an allocation strategy which decides to spend more or less budget on some individuals, thus parameter settings of the agent. A combination of parameter values is considered as an individual and each single parameter as gene. The fitness of an individual is computed as the payoff obtained by the MCTS simulation controlled by the corresponding parameter values.

Algorithm 13 shows the pseudocode for the EA allocation strategy and Figure 6.5 gives an overview of this strategy using the same example used for NMC and LSI, tuning three parameters with two possible values each. The main algorithm is implemented by the procedure EaParameterTuning(λ, μ, p_{cross}). The initial population Λ of size λ is generated randomly. Until the game is over, the current population is evaluated using MCTS simulations and evolved. When evolving, the μ elite individuals of the population (i.e. the ones with highest fitness) are used to

```
1: procedure EaParameterTuning(λ, μ, p_cross)
   Input: Population size λ, elite size μ, probability of generating an individual
   by uniform crossover p_cross.
2:     Λ ← ∅                                              ▷ Empty population set
3:     for i ← 1, ..., λ do
4:         p⃗ ← GenerateRandomIndividual( )
5:         Λ ← Λ ⋃{p⃗}
6:     while game not over do
7:         for p⃗ ∈ Λ do
8:             q ← PerformMctsSimulation(p⃗)
9:             UpdateFitness(p⃗, q)
10:            if game over then
11:                return
12:        M ← get μ individuals in Λ with highest fitness
13:        Λ ← M
14:        for i ← μ + 1, ..., λ do
15:            p⃗ ← GenerateIndividual(M, p_cross)
16:            Λ ← Λ ⋃{p⃗}

17: procedure GenerateIndividual(M, p_cross)
    Input: Set M of elite individuals in the population, probability of generating
    an individual by uniform crossover p_cross.
    Output: The generated individual.
18:     if Rand(0, 1) < p_cross then
19:         parent_1 ← random individual in M
20:         parent_2 ← random individual in M
21:         return UniformCrossover(parent_1, parent_2)
22:     else
23:         parent ← random individual in M
24:         return SingleRandomMutation(parent)
```

Algorithm 13: Pseudocode for the EA allocation strategy.

generate $\lambda - \mu$ new individuals. These new individuals, together with the elite will form the new population.

The procedure GenerateIndividual(M, p_{cross}) shows how a new individual is generated. With probability p_{cross} a new individual is generated by uniform crossover of two randomly selected elite individuals. Otherwise it is generated by random mutation of a single parameter value of a randomly selected elite individual.

N-Tuple Bandit Evolutionary Algorithm

The N-Tuple Bandit Evolutionary Algorithm (NTBEA) has been proposed by Lucas *et al.* (2018), which also used it to tune the parameters of a game-playing agent. It has also been applied by Kunanusont *et al.* (2017) to achieve automatic game parameter tuning. Like the previously presented EA, NTBEA considers each com-

```
1: procedure Init(L)
   Input: Set L with the length of the n-tuples that have to be considered.
2:     nTuples ← ∅
3:     for l ∈ L do
4:         lTuples ← generate from the set of parameters P all n-tuples of length l
5:         nTuples ← nTuples ∪ lTuples
6:     for t ∈ nTuples do
7:         LUT_t ← ∅

8: procedure UpdateStatistics(p⃗, q)
   Require: Initialized set of n-tuples, nTuples.
   Input: Combination of parameter values, p⃗, and payoff q obtained from the
   simulation controlled by parameter values p⃗.
9:     for t ∈ nTuples do
10:        if not LUT_t.contains(p⃗|_t) then
11:            LUT_t.put(p⃗|_t)
12:        entry ← LUT_t.get(p⃗|_t)
13:        entry.rsum ← entry.rsum + q
14:        entry.n ← entry.n + 1
15:        LUT_t.n ← LUT_t.n + 1

16: procedure UcbValue(p⃗, C_NTBEA)
   Require: Initialized set of n-tuples, nTuples.
   Input: Parameter value combination p⃗ for which to compute the UCB value,
   exploration constant C_NTBEA for the UCB formula.
   Output: UCB value of the given parameter combination.
17:    UCB ← 0
18:    count ← 0
19:    for t ∈ nTuples do
20:        entry ← LUT_t.get(p⃗|_t)
21:        if entry ≠ null then
22:            UCB_{p⃗|t} ← entry.rsum / entry.n + C_NTBEA × sqrt( ln(LUT_t.n) / entry.n )
23:            UCB ← UCB + UCB_{p⃗|t}
24:            count ← count + 1
25:    if count > 0 then
26:        return UCB / count
27:    else
28:        return 0
```

Algorithm 14: Pseudocode for *LModel*.

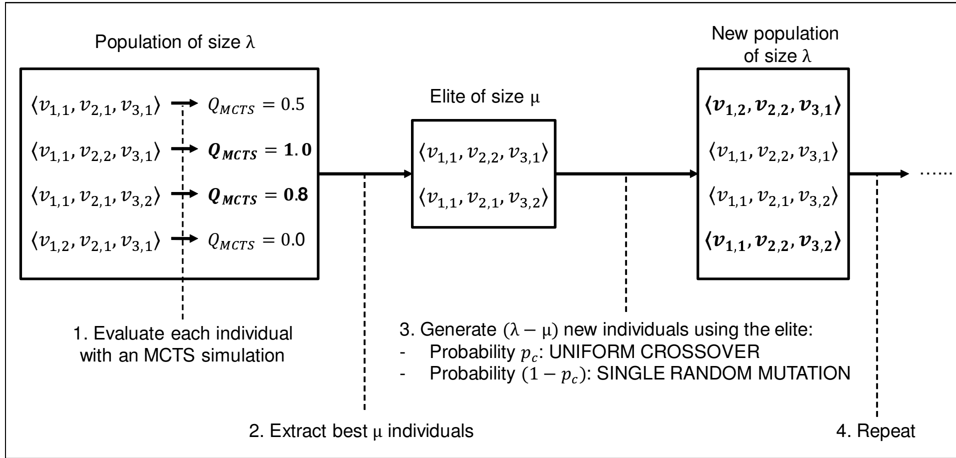


Figure 6.5: Overview of the execution of EA.

```
1: procedure NTBEAPARAMETERTUNING(X, 𝓛, C_NTBEA)
   Input: Number of neighbors X to generate during evolution, set 𝓛 with the
   length of the n-tuples that have to be considered, exploration constant C_NTBEA
   to compute the UCB values.
2:     LModel.INIT(𝓛)
3:     p⃗ ← GENERATERANDOMINDIVIDUAL( )
4:     while game not over do
5:         q ← PERFORMMCTSSIMULATION(p⃗)
6:         LModel.UPDATESTATISTICS(p⃗, q)
7:         𝒩 ← generate X neighbors of p⃗ by single random mutation
8:         p⃗ ← argmax_{p⃗′∈𝒩}(LModel.UCBVALUE(p⃗′, C_NTBEA))
```

Algorithm 15: Pseudocode for the NTBEA allocation strategy.

bination of parameters as an individual that is evolved over time by mutating single parameter values. Three components can be identified for NTBEA, the main *Evolutionary Algorithm*, a *noisy fitness evaluator* (represented in the parameter tuning problem by the MCTS simulations that evaluate parameter combinations), and an *n-tuple bandit fitness landscape model* (*LModel*, Algorithm 14).

NTBEA uses *LModel* to memorize statistics (e.g. mean, standard deviation, number of evaluations) of every legal value of every parameter and uses this model in combination with a bandit approach to decide which of the individuals should be evaluated next. Similar to the NMC approach, beside modeling each parameter as a MAB and each value of a parameter as an arm, each tuple formed by a subset of the available parameters is modeled as a macro-arm (e.g. if all the 2-tuples of a d-dimensional problem are considered, then $\frac{d(d-1)}{2}$ macro-arms will be used).

Algorithm 14 gives the pseudocode for the implementation of *LModel*. Given a d-dimensional search space, this model sub-samples its dimensions with a num-

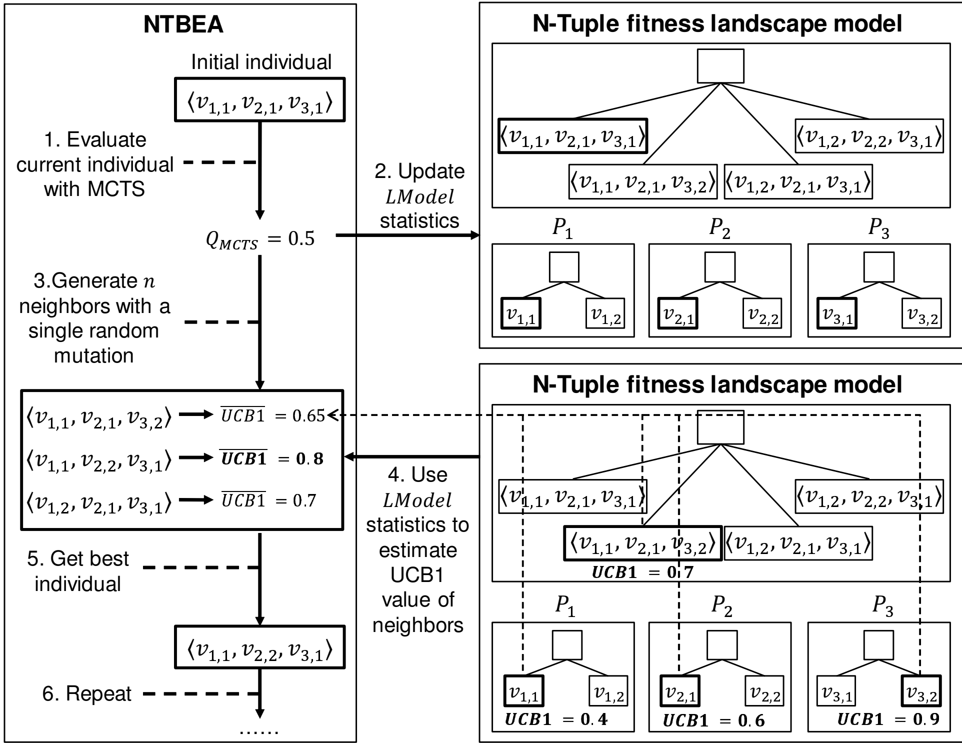


Figure 6.6: Overview of the execution of NTBEA.

ber of n-tuples with lengths that can range from 1 to d. Not all lengths in the range $[1, d]$ must necessarily be considered, but a set $\mathcal{L} \subseteq \{1, \ldots, d\}$ with the lengths l that have to be used can be specified. The procedure Init($\mathcal{L}$) shows how the landscape model is initialized. Given the set of parameters P, for each of the specified lengths $l \in \mathcal{L}$ all the possible l-tuples t are generated, and for each of them an empty look-up table LUT_t is created. For example, given $P = \{P_1, P_2, P_3\}$ and $\mathcal{L} = \{1, 2, 3\}$ the following n-tuples with their own LUT would be created: $\langle P_1\rangle, \langle P_2\rangle, \langle P_3\rangle, \langle P_1, P_2\rangle, \langle P_1, P_3\rangle, \langle P_2, P_3\rangle, \langle P_1, P_2, P_3\rangle$.

Whenever a combination of parameter values $\vec{p}$ is evaluated by performing an MCTS simulation, the obtained payoff q is used to update the LUT of each n-tuple as shown in the procedure UpdateStatistics$\vec{p}, q$. The notation $\vec{p}|_t$ indicates the vector of values in $\vec{p}$ that are assigned to the parameters considered by the n-tuple t. For example, given the set of parameters $P = \{P_1, P_2, P_3\}$, the parameter combination $\vec{p} = \langle p_1, p_2, p_3\rangle$, and the n-tuple $t = \langle P_1, P_3\rangle$: $\vec{p}|_{\langle P_1, P_3\rangle} = \langle p_1, p_3\rangle$. For each n-tuple t, the entry in LUT_t that corresponds to the value assignment $\vec{p}|_t$ is updated by increasing by 1 the number of visits and by q the total payoff sum. Finally, the number of visits of LUT_t is also increased by 1.

The procedure UcbValue($\vec{p}$, C_{NTBEA}) computes the UCB value of a given combination of parameters $\vec{p}$ using *LModel*. First, for each vector of values $\vec{p}|_t$ with at

least one visit the UCB1 value $UCB_{\vec{p}|_t}$ is computed, considering the corresponding LUT_t as a MAB. This means that each arm of the MAB corresponds to an entry in LUT_t, and thus to a possible assignment of values to the parameters considered by the n-tuple t. Then, all the $UCB_{\vec{p}|_t}$ values are averaged to obtain the UCB value for combination $\vec{p}$.

Algorithm 15 gives the pseudocode of the NTBEA allocation strategy and shows how it uses *LModel*. Figure 6.6 uses an example with three parameters, each with two possible values, to give an overview of the execution of NTBEA. This strategy starts with a randomly generated parameter combination $\vec{p}$, evaluates it with an MCTS simulation and uses the obtained payoff q to update statistics in *LModel*. Then, it generates x neighbors of $\vec{p}$ using single random mutations and computes their UCB value using *LModel*. The neighbor with the highest UCB value becomes the currently considered solution and the procedure is repeated.

6.3.2 Continuous Allocation Strategy

This subsection describes how CMA-ES is used as allocation strategy that considers the tuning problem as a CMAB with a continuous domain.

Covariance Matrix Adaptation Evolutionary Strategy

A powerful evolutionary computation technique is CMA-ES (Hansen *et al.*, 2003), a second-order method using the covariance matrix estimated iteratively by finite differences. It has been proved to be efficient for optimizing non-linear non-convex problems in the continuous domain without a-priori domain knowledge, thus no knowledge of the fitness landscape or the gradient function is required. Moreover, it has also been used to implement game-playing agents for domains with a continuous state or action space (Hausknecht *et al.*, 2014; Samothrakis *et al.*, 2014).

For each generation $(g+1)$, CMA-ES generates a population of λ new individuals $\vec{p}_j^{\,(g+1)}$ by sampling the multivariate normal distribution $\mathcal{N}(\vec{0}, \mathbf{C}^{(g)})$ as follows:

$$\vec{p}_j^{\,(g+1)} \sim \vec{x}^{\,(g)} + \sigma^{(g)} \mathcal{N}\left(\vec{0}, \mathbf{C}^{(g)}\right) \quad . \tag{6.3}$$

Here, $\vec{x}^{\,(g)}$, $\sigma^{(g)}$ and $\mathbf{C}^{(g)}$ are the mean value of the search distribution, the step-size and the covariance matrix at generation g, respectively. After computing the fitness of the population at generation $(g+1)$, the highest ranked individuals are used to update $\vec{x}$, σ and $\mathbf{C}$ for the next generation. More details can be found in the tutorial (Hansen, 2016).

As allocation strategy for the tuning problem an existing implementation of CMA-ES is used.[1] Algorithm 16 shows how it has been integrated in the code. The procedure CmaEsParameterTuning($\vec{x}, \sigma, \alpha$) shows how the strategy works. The variable *cma* refers to an instance of the CMA-ES algorithm initialized with the given start point $\vec{x}$ and step-size σ. Until CMA-ES meets one of its termination criteria, *cma*.SamplePopulation( ) samples a new population and its fitness is used to update the distribution by the procedure *cma*.UpdateDistribution($\vec{f}$).

[1] Code and details available at `cma.gforge.inria.fr`.

1: **procedure** CmaEsParameterTuning($\vec{x}, \sigma, \alpha$)
Input: Initial point (distribution mean) $\vec{x}$ and initial standard deviation (i.e. step-size) σ for the CMA-ES strategy, factor α used to compute the fitness penalty for infeasible solutions.
2: $cma \leftarrow$ InitializeCmaEs($\vec{x}, \sigma$)
3: **while not**(cma.IsStopped( )) **do**
4: $\Lambda \leftarrow cma$.SamplePopulation( )
5: $\vec{f} \leftarrow$ new vector of size $|\Lambda|$
6: **for** $\vec{p}_j \in \Lambda$ **do**
7: $f_j \leftarrow$ ComputeFitness($\vec{p}_j$)
8: cma.UpdateDistribution($\vec{f}$)
9: $\vec{p} \leftarrow cma$.GetMeanSolution( )
10: $f \leftarrow$ ComputeFitness($\vec{p}$)
11: cma.SetFitnessOfMean(f)
12: $\vec{p}^* \leftarrow cma$.GetBestSolution( )
13: $(\vec{p}^*, penalty) \leftarrow$ RepairIndividual($\vec{p}^*$, α)
14: $\vec{p}^* \leftarrow$ DenormalizeIndividual($\vec{p}^*$)
15: **while** game not over **do**
16: PerformMctsSimulation($\vec{p}^*$)

17: **procedure** ComputeFitness($\vec{p}$)
Input: Individual for which to compute the fitness $\vec{p}$.
Output: The fitness value of the given individual.
18: $(\vec{p}, penalty) \leftarrow$ RepairIndividual($\vec{p}$, α)
19: $\vec{p} \leftarrow$ DenormalizeIndividual($\vec{p}$)
20: $q \leftarrow$ PerformMctsSimulation($\vec{p}$)
21: **return** $(100 - q + penalty)$

22: **procedure** RepairIndividual($\vec{p}$, α)
Input: Individual $\vec{p}$ to be repaired if at least one of its values is infeasible ($\notin [0, 1]$).
Output: The repaired individual with its penalty.
23: $\vec{p}\,' \leftarrow$ new vector of size $|\vec{p}|$
24: **for** $p_i \in \vec{p}$ **do**
25: **if** $p_i \notin [0, 1]$ **then**
26: **if** $p_i < 0$ **then**
27: $p'_i \leftarrow 0$
28: **else**
29: $p'_i \leftarrow 1$
30: **else**
31: $p'_i \leftarrow p_i$
32: $penalty \leftarrow \alpha \left\|\vec{p} - \vec{p}\,'\right\|$
33: **return** $(\vec{p}\,', penalty)$

34: **procedure** DenormalizeIndividual($\vec{p}$)
Input: Individual $\vec{p}$ for which each value $p_i \in \vec{p}$ must be denormalized from $[0, 1]$ to its interval of feasible values $[min_{P_i}, max_{P_i}]$.
Output: Individual with denormalized values.
35: $\vec{p}\,' \leftarrow$ new vector of size $|\vec{p}|$
36: **for** $p_i \in \vec{p}$ **do**
37: $p'_i \leftarrow min_{P_i} + p_i(max_{P_i} - min_{P_i})$
38: **return** $\vec{p}\,'$

Algorithm 16: Pseudocode for the CMA-ES allocation strategy.

Upon termination, the mean value of the distribution is evaluated and its fitness updated. The overall best solution (parameter combination) is used to control the rest of the search.

Note that the computation of the fitness of an individual shown in procedure ComputeFitness($\vec{p}$) needs some precautions. First of all, CMA-ES minimizes the fitness function, while for the tuning problem it should be maximized, thus the fitness is computed as $(max_q - q)$, where max_q is the maximum achievable payoff for the game and q is the payoff obtained by the MCTS simulation controlled by the given parameter combination $\vec{p}$. Moreover, when tuning with CMA-ES each parameter is considered to be feasible in the interval $[0, 1]$ and the optimum is expected to be in the hyper-cube $[0, 1]^d$ (d number of parameters). This has two implications: (i) CMA-ES could still sample individuals with some values outside $[0, 1]$ and (ii) the actual interval of feasible values for the parameters might be different from $[0, 1]$. In the first case, whenever an individual is infeasible, its fitness is computed as the fitness of a repaired individual to which a penalty is added. This has been implemented according to the tutorial (Hansen, 2016) and is shown in the procedure RepairIndividual($\vec{p}, \alpha$). In the second case, before evaluating a combination of parameters with an MCTS simulation, all the values are denormalized from $[0, 1]$ to their own interval of feasible values as shown in the procedure DenormalizeIndividual($\vec{p}$).

6.4 Empirical Evaluation

This section presents an empirical evaluation of on-line parameter tuning and a comparison of the discussed allocation strategies. First, the setup of the performed experiments is presented in Subsection 6.4.1. Subsequently, Subsections 6.4.2, 6.4.3, 6.4.4, 6.4.5, 6.4.6, 6.4.7 and 6.4.8 report the obtained results. Finally, a discussion on how to select which parameters to tune is reported in Subsection 6.4.9.

6.4.1 Setup

On-line search-control parameter tuning has been tested on MCTS agents both for the Stanford GGP project and for the GVG-AI project. The Stanford GGP project has been used to evaluate all the different allocation strategies, while the GVG-AI project has been used to evaluate in a real-time domain the NTBEA and the MAB allocation strategies. The MAB strategy has been included in the experiments on the GVG-AI project in order to have a comparison of NTBEA with a strategy that does not exploit the combinatorial structure of the parameter space. First, this subsection presents the settings for the allocation strategies. Subsequently, it describes the experimental setup for the Stanford GGP project, and the experimental setup for the GVG-AI project.

Allocation Strategies Settings

MAB. The policy π_{MAB} is set to UCB1 with $C = 0.7$ (this value was shown to perform best among a predefined set of tested values). While performing experiments for the Stanford GGP project, it was noticed that this allocation

strategy introduces a high overhead on each MCTS simulation because of the exponential number of combinations that it has to check. To alleviate this issue the MAB allocation strategy chooses a new parameter combination every 10 simulations instead of 1. In this way for each selected combination 10 samples are collected all at once and the overhead is distributed over them. A comparison of the standard implementation of MAB with the implementation that uses a batch of simulations is reported in Appendix E. For the experiments on the GVG-AI project domain, instead, no batch is used. This decision is due to the low number of simulations that can be performed in such domain, therefore performing simulations in batches would reduce the number of visited distinct parameter combinations too much.

HE. The policy π_{HE} is set to the UCB1 policy with $C = 0.7$. When building the HE tree, the order of the parameters is randomized before every run of a game. This choice was determined by the fact that experiments with a fixed order for the parameters did not show a particular improvement in the performance of the agent. Detailed results of these experiments are reported in Appendix E

NMC. The policy π_0 is set to an ϵ-greedy strategy, which performs exploration with probability $\epsilon_0 = 0.75$ and exploitation with probability $(1 - \epsilon_0) = 0.25$. These values are the same as in (Ontañón *et al.*, 2013). The policies π_l and π_g are both set to UCB1 with exploration constants $C_l = C_g = 1$, thus with high exploration. Experiments with different values of C were performed and the results seemed to suggest that the performance might increase with more exploration. These results are reported in Appendix E.

LSI. The total number of available samples N_{tot} is estimated during start-clock using $\kappa = 3$. Three values for κ have been tested, and the one that seemed to perform best has been selected. Results for this test are reported in appendix E. N_{tot} is divided among the *generation* and *evaluation* steps as follows: $N_g = 0.75 \times N$ and $N_e = 0.25 \times N$ (this keeps the proportion between *generation* and *evaluation* the same as the proportion between *exploration* and *exploitation* in NMC). When tuning only two parameters the number of generated combinations k is set to 20, while when tuning more than two parameters $k = 2\,000$.

EA. The population size λ is set to 50 and the elite size μ is set to 25. The probability of generating a new individual by uniform crossover p_{cross} is set to 0.5. Smaller values for μ were tested, but resulted in a decreased performance, as shown in Appendix E.

NTBEA. The number of neighbors that are generated X is set to 5 (higher values cause a decrease in performance for the agent). The length considered to generate the n-tuples are set to $\mathcal{L} = \{1, d\}$, where d is the number of tuned parameters. Experiments considering all possible lengths for the n-tuples showed no improvement in performance and a decrease in simulations per second performed by the agents. The exploration constant C_{NTBEA} used to compute UCB1 values in *LModel* is set to 0.2. Appendix E reports detailed results for

Table 6.1: Default values, discrete and continuous domains of the parameters considered in the experiments on the Stanford GGP project.

Param.	Default value	Discrete domain	Continuous domain
C	0.2	$\{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$	$[0, 1]$
ϵ_{MAST}	0.4	$\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$	$[0, 1]$
K	250	$\{0, 10, 50, 100, 250, 500, 750, 1\,000, 2\,000, \infty\}$	$[0, 2\,000]$
ref	50	$\{0, 50, 100, 250, 500, 1\,000, 10\,000, \infty\}$	$[0, 10\,000]$
VO	0.01	$\{0.001, 0.005, 0.01, 0.015, 0.02, 0.025\}$	$[0, 0.025]$
T	0	$\{0, 5, 10, 20, 30, 40, 50, 100, 200, \infty\}$	$[0, 200]$

the experiments that tested different values for X and the use of all possible lengths for the n-tuples.

CMA-ES. According to the suggestions in the tutorial by Hansen (2016), the initial point $\vec{x}$ is set to a random point in $[0, 1]^d$ and σ is set to 0.3. The value of α used to compute the penalty of infeasible individuals is set to 100. In addition, all termination criteria regarding the fitness function are disabled, so that the optimization continues even if the minimum fitness is reached or if no significant change in fitness is observed. The motivation behind this choice is that the best parameter combination for MCTS might change over time, thus the strategy should keep exploring the search space. All other settings for CMA-ES are left to the default values (see tutorial by Hansen (2016)).

Stanford GGP Project Setup

On-line parameter tuning for the Stanford GGP project has been implemented for the agent developed in the GGP Base package (see Subsection 3.1.4). More precisely, it is applied to tune the search parameters of the following two agent instances:

- **SP**: an MCTS agent instance that uses UCT as selection strategy and MAST as play-out strategy.
- **AP**: a more advanced MCTS agent instance that uses GRAVE as selection strategy and MAST as play-out strategy.

The purpose of using a more advanced agent instance is twofold. First of all, AP has more search-control parameters and enables the experiments to verify how the allocation strategies scale when the search space increases. Second, it can be used to verify how on-line parameter tuning performs with a more informed selection strategy. When reporting the results of the experiments, the names SP and AP will identify the off-line tuned instances of the agent. When tuning the parameters on-line, the agent instances are represented with a subscript indicating the type of allocation strategy used (e.g. SP_{MAB} to indicate the SP instance being tuned on-line with the MAB allocation strategy).

The parameters that are considered tunable for the agent in subsequent experiments are the following:

- ***C*:** exploration constant used to compute the UCB1 value of an action in the UCT and GRAVE selection strategies.
- **$\epsilon_{\mathbf{MAST}}$:** probability of selecting a random action with the MAST play-out strategy.
- ***K*:** *equivalence parameter* of GRAVE (note that when $K = 0$ the selection strategy becomes pure UCT).
- ***ref*:** visit threshold used by GRAVE to choose the ancestor from which to compute the AMAF values (note that when $ref = 0$ the selection strategy becomes RAVE and when $ref = \infty$ the selection strategy becomes HRAVE).
- ***VO*:** value offset used to implement the random tie-breaking rule during action selection.
- ***T*:** this parameter is used to modify the MCTS selection strategy. During the selection for state s, if s has been visited less than T times, the next action is selected using the play-out strategy instead of the selection strategy (Coulom, 2007a).

Note that the parameters K and *ref* are relevant only for the AP instance of the agent, which is implementing GRAVE. Table 6.1 reports all the tunable parameters used by either SP or AP. Their default values are obtained by tuning them off-line in sequence on the following set of games taken from the GGP Base repository (Schreiber, 2016): 3D Tic Tac Toe, Breakthrough, Knightthrough, Skirmish, Battle, Chinook, and Chinese Checkers with 3 players. For K, *ref* and T the continuous domain has been restricted to a value much smaller than infinity (i.e. 2 000, 10 000 and 200, respectively) because after a certain threshold all values have more or less the same effect on the search. All combinations of parameter values are considered legal, except combinations with $K = 0$, that are legal only if the *ref* parameter is not considered in the combination. This is because when $K = 0$ the GRAVE strategy is not used, thus the *ref* parameter has no influence on the search.

In one of the series of experiments, the last available version of CadiaPlayer[2] (Finnsson, 2012b) is used as a benchmark to compare the performance of the best on-line tuning agent instance with the one of the off-line tuned agent instance.

Note that in the Stanford GGP project it is assumed that agents cannot remember previously learned knowledge in-between games. This means that both the game tree built by MCTS and the parameter statistics collected by the allocation strategies will be reset before each new game run.

All agent types are tested on a set of 14 heterogeneous games taken from the GGP Base repository (Schreiber, 2016): 3D Tic Tac Toe, Breakthrough, Knightthrough, Chinook, Chinese Checkers with 3 players, Checkers, Connect 5, Quad (the version

[2]Version of 18-11-2012. Downloaded from `http://cadia.ru.is/wiki/public:cadiaplayer:main`

played on a 7×7 board), Sheep and Wolf, Tic-Tac-Chess-Checkers-Four (TTCC4) with 2 and 3 players, Connect 4, Pentago and Reversi.[3] Most of these games are the same used in Chapter 5. On-line parameter tuning is expected to require a certain number of simulations to possibly have an effect on the search, therefore the games of Skirmish and Zhadu have been removed from the set. For these games the agents can perform on average a lower number of simulations per second than on the other games. Two more games for which the number of simulations per second reaches the same order of magnitude as the other games have been added, Connect Four and Pentago. Othello, although having a low number of simulations as well, instead of being removed has been substituted with a version that has a different GDL description. This GDL description, identified as Reversi, enables the agent to perform a slightly higher number of simulations per second, even if an order of magnitude lower than for all the other games.

For each experiment, two agent types at a time are matched against each other. All agent instances use the software implementation of the PropNet. A new PropNet is created before each game run and the same structure is given to both the involved agent instances to prevent any of them from having an advantage due to a faster structure. For each game, all possible assignments of agent types to the roles are considered, except the two configurations that assign the same agent to each role. All configurations are run the same number of times until at least 500 games have been played. Each game run has 1*s* start- and play-clock, except for the experiments that involve CadiaPlayer. In these experiments CadiaPlayer uses 10*s* start- and play-clock while the instances of the other agent use 1*s* start- and play-clock, because they are using a PropNet-based reasoner, and thus can perform a higher number of simulations per second. Experimental results always report the average win percentage of one of the two involved agent types with a 95%-confidence interval. The average win percentage of an agent type for a game is computed by assigning 1 point to the agent type that achieved the highest score and 0 to the other. If they both achieve the same score, they are given 0.5 points each.

GVG-AI Project Setup

To be tested on the GVG-AI project, on-line parameter tuning has been implemented in the GVG-AI framework for the single-player planning track agent MaastCTS2 (see Subsection 3.2.4). The parameters that are considered tunable for this agent in subsequent experiments are the following:

- $\boldsymbol{C}$**:** exploration constant used to compute the UCB1 value of an action in the PH selection strategy.
- $\boldsymbol{\epsilon_{\mathrm{NST}}}$**:** probability of selecting a random action with the NST play-out strategy.
- $\boldsymbol{W}$**:** weight used by the PH selection strategy.

[3] In this case Reversi has the same rules as Othello, but with a different formulation. This chapter is using a different GDL description of the game from the one used in Chapters 4 and 5. The name Reversi is used in this chapter to be consistent with the name used in the GGP Base repository to distinguish the GDL description of the game from the one identified as Othello.

Table 6.2: Default values, discrete domains and sub-optimal values of the parameters considered in the experiments on the GVG-AI project.

Param.	**Default value**	**Discrete domain**	**Sub-optimal value**
C	0.6	$\{0.2, 0.4, 0.5, 0.6, 0.7, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0\}$	2.0
ϵ_{NST}	0.5	$\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$	0.0
W	1	$\{0.1, 0.25, 0.5, 1, 3, 5, 7.5, 10, 20, 50\}$	0.1
N	7	$\{1, 3, 5, 7, 10, 15, 20, 30, 50\}$	50
L	3	$\{1, 2, 3, 4, 5\}$	1

- ***N*:** visit threshold used by NST to decide whether to use an N-Gram when computing the value of an action.
- ***L*:** maximum N-Gram length used by NST.

Table 6.2 reports all the tunable parameters for MaastCTS2, together with their default values, their discrete domains and their (expected to be) sub-optimal values. The latter are extreme values in the discrete domain that are expected to perform more poorly than the other values (e.g. $C = 2.0$ makes UCT explore too much and exploit too little, ϵ_{NST} makes the play-out strategy too greedy, etc.). The sub-optimal values are used to verify how much parameter settings are actually influencing the search. The default values are the ones to which parameters are set in the original implementation of the MaastCTS2 agent.

As for the Stanford GGP project, also for the GVG-AI project it is assumed that agents cannot remember previously learned knowledge in-between games. Therefore, both the game tree built by MCTS and the parameter statistics collected by the allocation strategies will be reset before each new game run.

The agent is tested on a set of 20 heterogeneous single-player games from the GVG-AI framework (Perez-Liebana *et al.*, 2016; Perez-Liebana, 2018): Aliens, Bait, Butterflies, Camel Race, Chase, Chopper, Crossfire, Dig Dug, Escape, Hungry Birds, Infection, Intersection, Lemmings, Missile Command, Modality, Plaque Attack, Roguelike, Sea Quest, Survive Zombies, and Wait for Breakfast. These games are the same used by Gaina *et al.* (2017) for studying the parameters of a GVG-AI agent based on a vanilla Rolling Horizon Evolutionary Algorithm. They have been selected uniformly at random from a list of games on which a simple MCTS agent has been shown to perform differently (Nelson, 2016; Bontrager *et al.*, 2016). In each series of experiments, the considered agent is tested equally on all 5 levels of these games until 500 samples per game have been collected.

Two series of experiments have been performed, one for which the agent has $40ms$ per game tick to choose an action and one for which the agent has $100ms$ per game tick. In all the games, there is no draw. A game terminates if the agent wins or loses the game before 2 000 game ticks have passed, otherwise the game is forced to terminate as a loss for the agent. This is the same setting as in the GVG-AI Single-Playing Planning competition. The only difference is that if the agent exceeds the

time limit per game tick it will not be disqualified and can still apply its selected move. Experimental results always report the average win percentage of the tested agent type with a 95%-confidence interval.

6.4.2 On-line Parameter Tuning for the SP Agent

This series of experiments evaluated the application of on-line tuning to the MCTS agent SP. Table 6.3 shows the results obtained by the agents that tune the parameters C and ϵ_{MAST} on-line with each of the presented allocation strategies against the off-line tuned SP agent. The on-line tuning agents SP_{NMC}, SP_{LSI}, SP_{EA} and SP_{NTBEA} reach at least the same overall performance of the off-line tuned agent, with SP_{NTBEA} seeming slightly better. Although the agents SP_{MAB}, SP_{HE} and $\text{SP}_{\text{CMA-ES}}$ seem to have an overall lower performance than the SP agent, they are still quite close.

The agent that performs overall best is SP_{NTBEA}. Among all tuning agents this is the one that achieves the highest win rate in most of the games. The performance of SP_{NMC}, SP_{LSI} and SP_{EA} is also better than the one of SP in many of the games. Each of these agents achieves the best performance in one or two of the tested games. Particularly remarkable is the performance of SP_{LSI} in *Quad*. For this game this agent reaches a much higher win rate than the other agents (81.2%). Also SP_{MAB}, SP_{HE} and $\text{SP}_{\text{CMA-ES}}$ are shown to perform better than SP in a few games, despite not being the ones with the best performance among the on-line tuning agents. However, they also show an evident decrease in performance in some of the games. For example, in Knightthrough $\text{SP}_{\text{CMA-ES}}$ wins only 20.8% of the times and in Breakthrough only 35.6% of the times. Moreover, SP_{MAB} and SP_{HE} show a low performance in Chinook, Connect Five and Quad.

6.4.3 On-line Tuning for the AP Agent

These series of experiments evaluated the application of on-line tuning to the more advanced MCTS agent AP. These series of experiments were performed for two (K and *ref*), four (C, ϵ_{MAST}, K and *ref*) and six (C, ϵ_{MAST}, K, *ref*, VO and T) on-line tuned parameters, and results are reported in Tables 6.4, 6.5 and 6.6, respectively. These tables show the results obtained by each of the agents that use one of the allocation strategies against the agent that is tuned manually off-line.

When tuning two parameters, other than the combination K and *ref*,also the combination C and ϵ was tested, but the latter achieved a worse performance. For this reason, only results for K and *ref* are reported. Looking at the results in Table 6.4, for two tuned parameters AP_{NMC}, AP_{LSI}, AP_{EA} and AP_{NTBEA} achieve overall at least the same performance of the off-line tuned agent. AP_{MAB} and AP_{HE} have a close performance to the off-line tuned AP as well. $\text{AP}_{\text{CMA-ES}}$ is the one with the lowest overall performance, while AP_{EA} is the one with the highest. However, if compared with the results obtained by tuning the simpler agent SP, in this case none of the allocation strategies seems to be superior to all the others. Each of them is one of the best in a few of the games. This suggests that it is more difficult to tune parameters for an agent that uses a more informed search strategy. Moreover, parameter values might have less influence than on a less informed search strategy.

Table 6.3: Win percentage of the on-line tuned SP agent that tunes two parameters with different allocation strategies against the SP agent with default parameter values.

Game	$\mathbf{SP_{MAB}}$	$\mathbf{SP_{HE}}$	$\mathbf{SP_{NMC}}$	$\mathbf{SP_{LSI}}$	$\mathbf{SP_{EA}}$	$\mathbf{SP_{NTBEA}}$	$\mathbf{SP_{CMA\text{-}ES}}$
3D Tic Tac Toe	43.0(±4.14)	42.9(±4.20)	42.9(±4.18)	**48.9**(±4.12)	43.6(±4.16)	48.0(±4.13)	42.9(±4.11)
Breakthrough	60.8(±4.28)	58.2(±4.33)	**61.0**(±4.28)	51.2(±4.39)	51.0(±4.39)	**61.0**(±4.28)	35.6(±4.20)
Knightthrough	45.4(±4.37)	45.0(±4.37)	48.0(±4.38)	35.2(±4.19)	40.0(±4.30)	**48.8**(±4.39)	20.8(±3.56)
Chinook	35.2(±3.24)	33.6(±3.25)	39.4(±3.41)	40.6(±3.50)	56.1(±3.58)	**65.7**(±3.37)	58.9(±3.51)
Chin.Checkers3P	41.7(±4.31)	33.1(±4.11)	45.0(±4.35)	40.7(±4.29)	44.6(±4.34)	**46.8**(±4.36)	42.5(±4.32)
Checkers	59.8(±4.15)	66.6(±3.91)	69.4(±3.83)	47.6(±4.17)	70.9(±3.79)	**74.6**(±3.63)	48.7(±4.17)
Connect Five	33.5(±3.18)	34.2(±3.20)	39.6(±3.24)	45.9(±3.37)	45.6(±3.42)	**46.0**(±3.28)	42.0(±3.51)
Quad	34.0(±3.97)	32.2(±3.95)	37.5(±4.04)	**81.2**(±3.22)	50.9(±4.10)	43.8(±4.12)	58.0(±4.11)
Sheep and Wolf	42.8(±4.34)	43.8(±4.35)	44.0(±4.36)	**52.2**(±4.38)	47.0(±4.38)	44.4(±4.36)	49.2(±4.39)
TTCC4 2P	63.5(±4.16)	61.1(±4.17)	69.0(±3.97)	60.9(±4.24)	70.9(±3.88)	**73.3**(±3.80)	57.1(±4.28)
TTCC4 3P	41.8(±4.17)	44.0(±4.18)	**48.2**(±4.28)	45.7(±4.23)	45.8(±4.26)	44.4(±4.20)	46.1(±4.26)
Connect Four	56.2(±4.15)	50.4(±4.18)	51.1(±4.21)	**79.7**(±3.36)	62.8(±4.06)	58.3(±4.11)	71.9(±3.77)
Pentago	61.0(±4.23)	56.8(±4.27)	63.0(±4.16)	63.2(±4.10)	66.2(±4.04)	**69.8**(±3.90)	63.5(±4.13)
Reversi	45.0(±4.28)	46.6(±4.27)	49.3(±4.32)	41.6(±4.23)	**55.5**(±4.31)	48.8(±4.32)	44.1(±4.29)
Avg. Win%	47.4(±1.12)	46.3(±1.11)	50.5(±1.12)	52.5(±1.12)	53.6(±1.12)	**55.3**(±1.11)	48.7(±1.12)

Table 6.4: Win percentage of the on-line tuned AP agent with different allocation strategies that tune two parameters against the AP agent with default parameter values.

Game	$\mathbf{AP_{MAB}}$	$\mathbf{AP_{HE}}$	$\mathbf{AP_{NMC}}$	$\mathbf{AP_{LSI}}$	$\mathbf{AP_{EA}}$	$\mathbf{AP_{NTBEA}}$	$\mathbf{AP_{CMA\text{-}ES}}$
3D Tic Tac Toe	43.9(±4.05)	46.2(±4.09)	47.2(±4.10)	42.0(±4.00)	**52.0**(±4.13)	46.0(±4.11)	41.5(±4.00)
Breakthrough	49.2(±4.39)	43.6(±4.35)	**51.4**(±4.39)	40.8(±4.31)	47.8(±4.38)	48.6(±4.39)	35.2(±4.19)
Knightthrough	53.4(±4.38)	57.8(±4.33)	**58.0**(±4.33)	49.6(±4.39)	46.2(±4.37)	46.8(±4.38)	47.2(±4.38)
Chinook	53.7(±3.97)	53.4(±3.95)	56.6(±4.04)	55.0(±4.11)	60.2(±3.98)	63.5(±3.95)	**65.0**(±3.88)
Chin.Checkers3P	50.6(±4.37)	52.4(±4.36)	**52.8**(±4.36)	49.4(±4.37)	52.6(±4.36)	51.0(±4.37)	44.2(±4.34)
Checkers	47.8(±4.09)	46.7(±4.08)	44.2(±4.06)	**48.2**(±4.11)	47.8(±4.10)	47.6(±4.13)	41.8(±4.04)
Connect Five	45.1(±3.13)	44.2(±3.20)	44.6(±3.12)	**49.2**(±3.16)	46.2(±3.08)	45.7(±3.05)	42.9(±3.17)
Quad	53.1(±4.18)	56.0(±4.04)	57.0(±4.14)	**64.3**(±3.91)	60.4(±3.99)	60.1(±4.05)	52.2(±4.14)
Sheep and Wolf	48.8(±4.39)	**53.4**(±4.38)	50.4(±4.39)	49.4(±4.39)	53.2(±4.38)	52.2(±4.38)	50.8(±4.39)
TTCC4 2P	49.4(±4.24)	49.1(±4.26)	49.2(±4.23)	52.0(±4.25)	**52.9**(±4.22)	51.5(±4.19)	44.4(±4.22)
TTCC4 3P	46.9(±4.23)	49.7(±4.20)	48.4(±4.21)	**53.0**(±4.21)	49.5(±4.26)	48.4(±4.26)	43.0(±4.22)
Connect Four	53.8(±4.19)	49.3(±4.21)	54.1(±4.12)	53.3(±4.14)	55.3(±4.15)	**55.6**(±4.18)	50.8(±4.12)
Pentago	48.5(±4.22)	44.9(±4.23)	54.1(±4.25)	48.5(±4.26)	**56.2**(±4.17)	55.3(±4.21)	51.9(±4.18)
Reversi	44.1(±4.29)	**48.9**(±4.32)	48.0(±4.32)	45.2(±4.28)	47.1(±4.28)	46.9(±4.33)	39.4(±4.23)
Avg. Win%	49.2(±1.11)	49.7(±1.11)	51.1(±1.11)	50.0(±1.11)	**52.0**(±1.11)	51.4(±1.12)	46.4(±1.11)

Table 6.5: Win percentage of the on-line tuned AP agent with different allocation strategies that tune four parameters against the AP agent with default parameter values.

Game	$\mathbf{AP_{MAB}}$	$\mathbf{AP_{HE}}$	$\mathbf{AP_{NMC}}$	$\mathbf{AP_{LSI}}$	$\mathbf{AP_{EA}}$	$\mathbf{AP_{NTBEA}}$	$\mathbf{AP_{CMA\text{-}ES}}$
3D Tic Tac Toe	20.8(±3.38)	34.5(±3.97)	39.8(±4.05)	42.3(±4.11)	**48.7**(±4.15)	39.5(±4.08)	37.1(±4.00)
Breakthrough	8.0(±2.38)	53.6(±4.38)	**60.6**(±4.29)	37.2(±4.24)	59.8(±4.30)	55.2(±4.36)	18.2(±3.39)
Knightthrough	20.0(±3.51)	69.8(±4.03)	**74.2**(±3.84)	53.8(±4.37)	68.4(±4.08)	68.2(±4.09)	28.6(±3.96)
Chinook	21.5(±3.31)	30.8(±3.70)	36.7(±3.75)	24.8(±3.55)	**52.3**(±4.05)	51.0(±4.05)	48.9(±4.17)
Chin.Checkers3P	38.7(±4.26)	34.9(±4.17)	36.9(±4.22)	36.1(±4.20)	39.5(±4.27)	**42.7**(±4.32)	40.3(±4.29)
Checkers	8.3(±2.23)	33.6(±3.92)	37.8(±3.91)	19.7(±3.28)	**42.4**(±4.00)	40.4(±4.06)	30.2(±3.84)
Connect Five	13.8(±2.29)	25.9(±3.01)	30.7(±3.11)	**40.3**(±3.27)	29.1(±2.95)	28.6(±3.07)	20.3(±2.68)
Quad	46.8(±4.23)	29.9(±3.76)	37.9(±4.04)	**75.6**(±3.56)	35.3(±3.98)	51.9(±4.21)	45.9(±4.09)
Sheep and Wolf	42.6(±4.34)	43.2(±4.35)	45.2(±4.37)	**49.0**(±4.39)	47.6(±4.38)	44.8(±4.36)	47.4(±4.38)
TTCC4 2P	22.4(±3.63)	38.6(±4.15)	44.6(±4.25)	22.6(±3.60)	45.4(±4.25)	**49.7**(±4.20)	34.4(±4.08)
TTCC4 3P	43.9(±4.25)	40.6(±4.14)	40.4(±4.17)	**47.3**(±4.24)	43.2(±4.21)	43.1(±4.18)	45.4(±4.26)
Connect Four	42.0(±4.14)	37.0(±4.09)	42.9(±4.15)	**59.1**(±4.18)	50.1(±4.19)	46.8(±4.20)	49.4(±4.24)
Pentago	26.1(±3.75)	40.6(±4.14)	38.8(±4.07)	**48.8**(±4.22)	41.7(±4.16)	43.5(±4.15)	41.2(±4.16)
Reversi	31.1(±4.00)	41.5(±4.26)	39.8(±4.24)	27.1(±3.83)	42.9(±4.28)	**45.1**(±4.33)	34.9(±4.12)
Avg. Win%	27.6(±1.01)	39.6(±1.10)	43.3(±1.11)	41.7(±1.11)	46.2(±1.12)	**46.5**(±1.12)	37.3(±1.09)

Table 6.6: Win percentage of the on-line tuned AP agent with different allocation strategies that tune six parameters against the AP agent with default parameter values.

Game	**AP$_{\mathrm{MAB}}$**	**AP$_{\mathrm{HE}}$**	**AP$_{\mathrm{NMC}}$**	**AP$_{\mathrm{LSI}}$**	**AP$_{\mathrm{EA}}$**	**AP$_{\mathrm{NTBEA}}$**	**AP$_{\mathrm{CMA\text{-}ES}}$**
3D Tic Tac Toe	1.1(±0.89)	24.0(±3.55)	27.3(±3.71)	28.3(±3.74)	**35.2**(±3.99)	38.6(±4.05)	36.1(±3.89)
Breakthrough	1.0(±0.87)	34.0(±4.16)	28.6(±3.96)	23.0(±3.69)	**37.4**(±4.25)	31.6(±4.08)	18.0(±3.37)
Knightthrough	1.6(±1.10)	45.8(±4.37)	46.2(±4.37)	28.4(±3.96)	45.6(±4.37)	**50.2**(±4.39)	39.2(±4.28)
Chinook	4.1(±1.52)	13.8(±2.72)	18.3(±3.21)	9.5(±2.23)	21.7(±3.40)	31.6(±3.94)	**48.6**(±4.22)
Chin.Checkers3P	19.8(±3.49)	36.7(±4.21)	28.0(±3.92)	**41.1**(±4.30)	30.2(±4.01)	28.0(±3.92)	35.9(±4.19)
Checkers	1.7(±0.97)	14.7(±2.83)	17.7(±3.13)	19.3(±3.25)	17.2(±3.06)	**20.9**(±3.42)	19.6(±3.32)
Connect Five	5.4(±1.52)	20.4(±2.74)	24.6(±2.98)	**41.2**(±3.31)	25.5(±2.89)	33.2(±3.26)	37.6(±3.21)
Quad	3.1(±1.48)	16.4(±3.04)	16.2(±3.02)	**63.5**(±4.01)	15.2(±3.00)	17.2(±3.13)	19.0(±3.26)
Sheep and Wolf	30.0(±4.02)	43.8(±4.35)	40.2(±4.30)	48.8(±4.39)	46.2(±4.37)	46.2(±4.37)	**49.4**(±4.39)
TTCC4 2P	2.0(±1.23)	26.6(±3.82)	26.2(±3.74)	17.9(±3.32)	28.7(±3.91)	**33.6**(±4.03)	25.0(±3.73)
TTCC4 3P	24.2(±3.70)	**41.0**(±4.20)	34.0(±4.04)	38.7(±4.18)	38.9(±4.12)	39.4(±4.15)	40.6(±4.23)
Connect Four	9.6(±2.49)	29.4(±3.80)	35.0(±3.97)	**53.8**(±4.15)	43.8(±4.14)	30.6(±3.89)	36.6(±4.05)
Pentago	5.0(±1.89)	34.6(±4.03)	35.2(±4.05)	40.2(±4.17)	42.3(±4.19)	42.1(±4.18)	**42.8**(±4.22)
Reversi	16.9(±3.22)	32.2(±4.00)	33.9(±4.08)	31.4(±4.01)	32.2(±4.02)	33.5(±4.07)	**34.2**(±4.11)
Avg. Win%	9.0(±0.65)	29.5(±1.03)	29.4(±1.03)	**34.7**(±1.07)	32.9(±1.06)	34.0(±1.07)	34.5(±1.07)

Results in Tables 6.5 and 6.6 show that for all tuning agents the overall performance decreases with the increase in number of tuned parameters. It might be that not all parameters have the same importance and by tuning them some noise is being introduced in the process. Moreover, more parameters mean a larger search space, with fewer good parameter combinations. For this reason, it could be more difficult for the tuning agents to converge to an optimal combination and they keep evaluating sub-optimal ones. It might also be that, by the time they identify better parameter combinations, the AP agent has already an advantage in the game because it was making better decisions from the start due to already tuned parameters. In particular, $\mathrm{AP_{MAB}}$ is the agent that loses the most in performance when increasing the number of parameters.

The poor performance of $\mathrm{AP_{MAB}}$ is because of the high number of possible values combinations. This prevents the agent to be able to sample each combination a sufficient number of times to start converging. Another reason for its poor performance is that every time a combination must be selected there is quite some computational overhead due to the necessity of iterating over all possible combinations to compute the one with the highest UCB1 value. This reduces the number of simulations that can be performed. Performing the evaluation of each parameter combination using a batch of simulations instead of a single simulation is still not sufficient to increase the performance to the same level as the off-line tuned agent.

Despite the statistically significant worsening of the performance in most of the games, it still seems to be beneficial to tune four parameters for a few of them. For *Knightthrough* and *Breakthrough*, for example, $\mathrm{AP_{NMC}}$, $\mathrm{AP_{EA}}$ and $\mathrm{AP_{NTBEA}}$ perform better than AP when tuning four parameters rather than only two. The performance of $\mathrm{P_{LSI}}$ also increases when tuning four parameters for *Quad*. A reason for this might be that for these games the fixed parameters of AP are not optimal, but it could also be that optimal values are changing during the search and on-line tuning detects this.

Comparing the tuning agents with each other, for four parameters $\mathrm{AP_{EA}}$ and $\mathrm{AP_{NTBEA}}$ show the overall best performance. For six parameters $\mathrm{AP_{LSI}}$, $\mathrm{AP_{NTBEA}}$ and $\mathrm{AP_{CMA\text{-}ES}}$ are the ones performing best, having a win percentage around 34.0%. Over all the experiments, $\mathrm{AP_{NTBEA}}$ seems to be the best performing agent.

An aspect worth investigating that might be influencing the performance of the on-line tuning agents is the impact of the overhead of selecting parameter values. Table 6.7 gives as reference the speed (i.e. average median of number of visited nodes per second) of the off-line tuned AP agent. For the self-adaptive AP agents that tune four parameters the table reports the percentage of speed variation with respect to the off-line tuned AP. For almost all games parameter tuning decreases the speed, especially for $\mathrm{AP_{MAB}}$ because of the previously mentioned high overhead. Also for $\mathrm{AP_{NMC}}$ and $\mathrm{AP_{NTBEA}}$ the speed seems to decrease more than for $\mathrm{AP_{LSI}}$, $\mathrm{AP_{EA}}$ and $\mathrm{AP_{CMA\text{-}ES}}$. When looking at the speed decrease for two and six parameters as well it was noticed that for $\mathrm{AP_{LSI}}$, $\mathrm{AP_{EA}}$ and $\mathrm{AP_{CMA\text{-}ES}}$ it is on average between -0.8% and -4.9%, while for $\mathrm{AP_{NMC}}$ it goes from -2.0% for two parameters to -25.5% for six, and for $\mathrm{AP_{NTBEA}}$ from -8.5% for two parameters to -12.7% for six. This can be explained by the frequency with which these two agents have to perform costly UCB1 evaluations to select parameters. The speed decrease of $\mathrm{AP_{HE}}$ is somewhere

Table 6.7: Variation (%) of visited nodes per second of the on-line tuned AP agent that tunes four parameters with respect to the off-line tuned AP agent.

Game	**AP speed**	$\mathbf{AP_{MAB}}$	$\mathbf{AP_{HE}}$	$\mathbf{AP_{NMC}}$	$\mathbf{AP_{LSI}}$	$\mathbf{AP_{EA}}$	$\mathbf{AP_{NTBEA}}$	$\mathbf{AP_{CMA\text{-}ES}}$
3D Tic Tac Toe	58731	-44.7%	-7.7%	-18.4%	0.2%	-6.9%	-19.7%	-8.0%
Breakthrough	51089	-44.4%	-6.5%	-11.9%	-3.3%	-4.2%	-14.6%	-4.5%
Knightthrough	42691	-47.1%	-6.0%	-12.9%	-2.2%	-5.1%	-18.7%	-7.5%
Chinook	33817	-27.0%	-4.1%	-9.8%	-0.5%	-3.1%	-11.2%	-4.5%
Chin.Checkers3P	106074	-36.8%	-6.4%	-27.9%	-1.8%	-1.4%	-20.1%	-5.9%
Checkers	29767	-4.2%	1.8%	-1.4%	1.9%	1.8%	1.1%	0.9%
Connect Five	37488	-19.5%	-5.4%	-10.7%	3.2%	-1.9%	-9.4%	-4.8%
Quad	42358	-32.5%	-5.2%	-14.1%	-2.6%	-4.9%	-15.2%	-6.5%
Sheep and Wolf	51036	-18.2%	-3.2%	-22.0%	-1.1%	-1.3%	-8.6%	-4.0%
TTCC4 2P	26936	-11.3%	-3.5%	-6.5%	-1.9%	-2.2%	-6.2%	-3.1%
TTCC4 3P	23462	-24.8%	-5.9%	-15.7%	-1.9%	0.6%	-8.1%	-6.8%
Connect Four	114623	-62.6%	-7.1%	-24.3%	-6.0%	-3.9%	-23.6%	-9.3%
Pentago	86132	-25.5%	-2.3%	-12.7%	0.1%	1.4%	-9.0%	-3.1%
Reversi	8533	-1.1%	0.1%	-1.9%	-0.5%	0.1%	-0.5%	-1.6%

in-between, because it is goes from −4.1% for two parameters to −9.0% for six. AP_{HE} has to perform UCB1 evaluations for each parameter, but differently from AP_{NMC} and AP_{NTBEA} these evaluations are always on MABs that choose values for single parameters and never for global MABs with many possible parameter combinations.

A decrease in visited nodes per second, however, does not seem to always imply a negative performance. Table 6.8 gives the percentage variation in MCTS iterations per second performed by the agents that tune four parameters on-line with respect to the number of iterations per second (average of the median) reached by the AP agent. From these results it is visible that in many games the on-line tuned AP agents can perform more iterations than the off-line tuned AP agent, and for some games this might have a positive effect on the search. This might be happening for the games of *Quad* and *Connect Four* with AP_{LSI}. This agent for these games has an increase of 13.2% and 15.5% in number of iterations per second, respectively, and it shows a better performance than AP. The explanation for the increase in iterations might be that the constantly changing search-control parameters cause the agents to explore different parts of the search space (with shorter paths) than the ones explored by the off-line tuned AP.

6.4.4 On-line Parameter Tuning Validation

This series of experiments is designed to verify if on-line parameter tuning has an advantage over fixed parameter values when such values are performing poorly. Given that NTBEA seems to be the allocation strategy that performs best in previous experiments, this series of experiments considers the SP_{NTBEA} agent tuning two parameters (C and ϵ_{MAST}), and the AP_{NTBEA} agents tuning two (K and *ref*), four (C, ϵ_{MAST}, K and *ref*) and six parameters (C, ϵ_{MAST}, K, *ref*, VO and T). Each agent is matched against the corresponding non-tuning agent that, before each game run, sets its parameters to randomly chosen values among the available ones. Each of these non-tuning agents will randomize only the values of the parameters that the corresponding self-adaptive agent is tuning. Testing the tuning agents against all possible agents with fixed settings is too time consuming because of the large number of available parameter combinations. Therefore, randomization is used in these experiments to guarantee that many of the fixed parameter combinations used by the non-tuning agents will be performing poorly. Results are shown in Table 6.9. For almost all games, the self-adaptive agents have a significantly better performance than the agents that randomize parameter values before each game run, proving that on-line parameter tuning can converge to better parameter settings when the opponent's parameters are set to sub-optimal values.

6.4.5 Parameters Inter-dependency

All the proposed allocation strategies are designed to take into account that there is inter-dependency among the tuned parameters. As a validation of this inter-dependency assumption, this series of experiments tests the performance of an agent, AP_{LOCAL}, that tunes four parameters, C, ϵ_{MAST}, K, and *ref*, considering no inter-

Table 6.8: Variation (%) of MCTS iterations per second of the on-line tuned AP agent that tunes four parameters with respect to the off-line tuned AP agent.

Game	**AP speed**	**AP_{MAB}**	**AP_{HE}**	**AP_{NMC}**	**AP_{LSI}**	**AP_{EA}**	**AP_{NTBEA}**	**$AP_{CMA\text{-}ES}$**
3D Tic Tac Toe	5143	-26.5%	15.1%	11.0%	17.3%	27.4%	10.6%	19.2%
Breakthrough	4258	-29.1%	7.0%	6.0%	-4.1%	22.6%	9.6%	13.0%
Knightthrough	5426	-29.6%	7.9%	3.0%	4.3%	20.4%	1.3%	14.2%
Chinook	3562	-16.0%	-7.3%	-7.5%	7.2%	-1.2%	-9.2%	0.4%
Chin.Checkers3P	5027	-37.3%	-2.6%	-20.3%	-7.3%	3.8%	-12.9%	-5.6%
Checkers	624	1.6%	3.6%	3.3%	2.2%	5.9%	3.5%	1.3%
Connect Five	2172	-0.2%	15.8%	16.1%	1.9%	26.7%	19.3%	17.1%
Quad	3641	-14.3%	6.8%	3.9%	13.2%	14.5%	4.6%	9.7%
Sheep and Wolf	2411	-14.2%	0.5%	-19.0%	-4.7%	4.2%	-5.2%	-6.3%
TTCC4 2P	1422	0.8%	7.4%	6.8%	-3.2%	13.5%	10.7%	9.5%
TTCC4 3P	2119	-28.0%	-0.5%	-8.5%	-6.5%	5.9%	-2.3%	-5.6%
Connect Four	8300	-45.9%	2.3%	-7.5%	15.5%	13.9%	-7.2%	12.9%
Pentago	4149	-29.2%	3.1%	-5.0%	-3.0%	10.2%	1.0%	0.2%
Reversi	290	-2.0%	0.4%	-1.3%	-0.4%	0.6%	-0.1%	-0.7%

Table 6.9: Win percentage of SP_{NTBEA} and AP_{NTBEA} against agents that randomize parameter values before each game run.

Game	SP_{NTBEA}	AP_{NTBEA}		
	2 param.	**2 param.**	**4 param.**	**6 param.**
3D Tic Tac Toe	58.4(±4.11)	59.3(±4.06)	65.4(±3.94)	64.0(±4.03)
Breakthrough	80.6(±3.47)	65.2(±4.18)	88.4(±2.81)	80.8(±3.46)
Knightthrough	80.6(±3.47)	55.0(±4.37)	87.2(±2.93)	83.6(±3.25)
Chinook	77.8(±2.83)	62.8(±3.91)	77.1(±3.43)	63.7(±4.04)
Chin.Checkers3P	54.5(±4.35)	56.0(±4.34)	58.1(±4.31)	49.2(±4.37)
Checkers	78.9(±3.34)	59.1(±4.10)	74.7(±3.57)	57.6(±4.15)
Connect Five	62.4(±3.30)	58.5(±3.19)	46.5(±3.73)	54.5(±3.73)
Quad	41.4(±4.12)	60.6(±4.02)	54.6(±4.21)	35.5(±4.08)
Sheep and Wolf	53.6(±4.38)	51.8(±4.38)	47.6(±4.38)	51.4(±4.39)
TTCC4 2P	77.1(±3.63)	62.8(±4.10)	78.8(±3.47)	71.3(±3.92)
TTCC4 3P	52.4(±4.24)	56.6(±4.24)	50.8(±4.29)	49.5(±4.30)
Connect Four	44.6(±4.20)	65.6(±4.02)	52.6(±4.23)	51.3(±4.27)
Pentago	60.3(±4.01)	61.1(±4.15)	55.4(±4.13)	57.9(±4.16)
Reversi	63.1(±4.15)	56.2(±4.27)	58.1(±4.28)	54.3(±4.32)
Avg. Win%	63.2(±1.07)	59.3(±1.10)	63.9(±1.08)	58.9(±1.12)

dependency. This agent chooses which combination to evaluate by selecting each parameter value using a separate MAB (like using only local MABs with NMC). Two instances of this agent are considered, one that selects the parameter values from the local MABs using UCB1 with exploration constant $C_l = 0.7$ and one that selects parameter values using UCB1 with exploration constant $C_l = 1$. Results are reported in Table 6.10. The last column of the table reports for each game the highest win percentage from Table 6.5, therefore the highest that was achieved by any of the instances of the AP agent that tune four parameters. The overall win percentage of both instances of AP_{LOCAL} against AP is lower than the win percentage of all the tuning agents that achieve the highest win percentage in at least one game of Table 6.10 (i.e. AP_{NMC}, AP_{LSI}, AP_{EA} and AP_{NTBEA}). It was also observed that for most games at least one of the allocation strategies that exploit parameter interdependency can significantly outperform the strategy that does not. This can be seen as a confirmation that there is a dependency and it should be exploited.

6.4.6 Tuning Six Parameters with Different Time Constraints

This series of experiments verifies how different time constraints affect the performance of on-line parameter tuning for six parameters, C, ϵ_{MAST}, K, *ref*, VO and T. The agents AP_{NMC}, AP_{LSI}, AP_{EA} and AP_{NTBEA} that tune six parameters are matched against AP for increasing time constraints. Figure 6.7 shows for each agent how the average win percentage over all the tested games changes when using longer start- and play-clock. Results for $1s$ differ from the ones presented in Table 6.6

Table 6.10: Win percentage of the on-line tuning AP_{LOCAL} agent that does not consider interdependency among parameters, and of the on-line tuning AP agents that achieve the highest win percentage against off-line tuned AP. The on-line tuning agents are tuning four parameters.

Game	**AP_{LOCAL}**		**Best on-line tuning AP**
	$C_l = 0.7$	$C_l = 1$	
3D Tic Tac Toe	37.2(±4.03)	34.8(±3.95)	**48.7**(±4.15)
Breakthrough	54.8(±4.37)	53.4(±4.38)	**60.6**(±4.29)
Knightthrough	70.2(±4.01)	66.4(±4.14)	**74.2**(±3.84)
Chinook	30.0(±3.56)	26.1(±3.47)	**52.3**(±4.05)
Chin.Checkers3P	30.6(±4.03)	31.3(±4.05)	**42.7**(±4.32)
Checkers	34.3(±3.95)	33.1(±3.82)	**42.4**(±4.00)
Connect Five	30.6(±3.23)	27.1(±3.07)	**40.3**(±3.27)
Quad	30.3(±3.79)	34.6(±3.90)	**75.6**(±3.56)
Sheep and Wolf	45.4(±4.37)	48.8(±4.39)	**49.0**(±4.39)
TTCC4 2P	34.8(±4.07)	38.3(±4.14)	**49.7**(±4.20)
TTCC4 3P	41.3(±4.09)	45.6(±4.19)	**47.3**(±4.24)
Connect Four	36.7(±4.03)	35.6(±3.98)	**59.1**(±4.18)
Pentago	37.6(±4.11)	37.0(±4.04)	**48.8**(±4.22)
Reversi	33.1(±4.05)	38.5(±4.17)	**45.1**(±4.33)
Avg. Win%	39.1(±1.09)	39.3(±1.09)	**52.5**(±1.12)

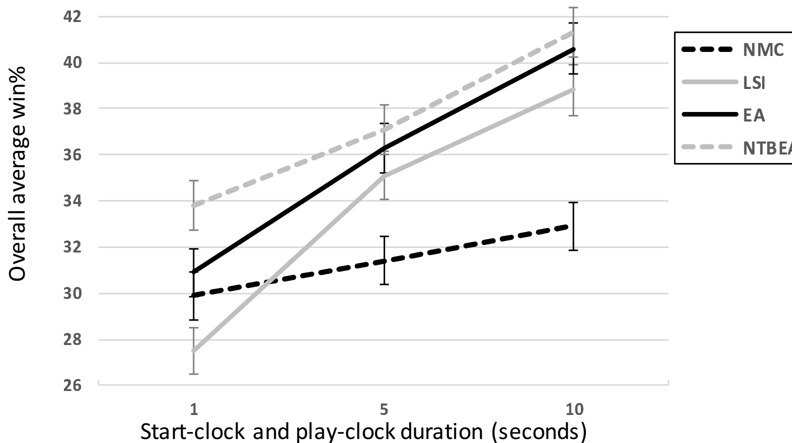


Figure 6.7: Win percentage of AP_{NMC}, AP_{LSI}, AP_{EA} and AP_{NTBEA} tuning six parameters with different time constraints.

Table 6.11: Win percentage of AP and AP_{NTBEA} (1*s* start- and play-clock) against CadiaPlayer (10*s* start- and play-clock).

Game	AP	AP_{NTBEA}		
		2 param.	**4 param.**	**6 param.**
3D Tic Tac Toe	**92.1**(±2.36)	91.9(±2.34)	90.4(±2.55)	86.7(±2.89)
Breakthrough	63.2(±4.23)	61.8(±4.26)	**68.0**(±4.09)	45.8(±4.37)
Knightthrough	50.8(±4.39)	52.2(±4.38)	**74.8**(±3.81)	45.0(±4.37)
Chinook	82.8(±3.22)	**88.0**(±2.74)	81.3(±3.28)	63.4(±4.10)
Checkers	90.6(±2.32)	**91.2**(±2.28)	87.6(±2.71)	52.6(±4.12)
Connect Five	**70.4**(±3.18)	68.2(±3.29)	45.5(±3.78)	51.9(±3.95)
Quad	98.8(±0.96)	99.2(±0.78)	**99.4**(±0.68)	93.0(±2.24)
Sheep and Wolf	56.8(±4.35)	**60.4**(±4.29)	51.6(±4.38)	50.0(±4.39)
Connect Four	68.2(±3.90)	**69.7**(±3.92)	63.2(±4.06)	48.0(±4.24)
Pentago	73.0(±3.80)	**78.1**(±3.52)	71.3(±3.80)	62.6(±4.10)
Avg. Win%	74.7(±1.16)	**76.1**(±1.14)	73.3(±1.19)	59.9(±1.32)

because the experiment ran on a different server, where the agents could perform fewer simulations per second. With more time all agents increase their performance, even if none of them can reach the performance of the off-line tuned agent. Among the on-line tuning agents AP_{NMC} is the one that benefits the least from the increase in thinking time, while AP_{LSI} is the one that benefits the most. Overall, AP_{NTBEA} seems to be the best performing agent, though for 5*s* and 10*s* its confidence interval overlaps with the one of AP_{EA}.

6.4.7 Best On-line Tuning Agent vs CadiaPlayer

In this series of experiments the off-line tuned agent AP and the best on-line tuning agent AP_{NTBEA} are matched against CadiaPlayer. Three versions of AP_{NTBEA} are considered, the ones that tunes two (K and *ref*), four (C, ϵ_{MAST}, K and *ref*) and six (C, ϵ_{MAST}, K, *ref*, VO and T) parameters. Table 6.11 shows the obtained results. Four games (*Chinese Checkers* with 3 players, *TTCC4* with 2 and 3 players, and *Reversi*) are excluded from the experiments because CadiaPlayer encountered some errors while playing them. Results show that both AP and AP_{NTBEA} that tunes two, four and six parameters are better than CadiaPlayer on average. For most of the games at least one among AP_{NTBEA} that tunes two or four parameters performs better than AP. The AP_{NTBEA} instance that tunes six parameters, instead, performs always worse than AP. In line with previous results, AP_{NTBEA} that tunes four parameters performs overall worse than AP_{NTBEA} that tunes only two parameters, and AP_{NTBEA} that tunes six parameters performs overall worse than AP_{NTBEA} that tunes four parameters. However, the difference in performance between tuning two or four parameters is not very large, while the gap in performance between tuning four or six parameters is much larger. Overall, AP_{NTBEA} that tunes two parameters seems to be the one that performs best against CadiaPlayer.

6.4.8 On-line Parameter Tuning in Real-time Settings

This series of experiments evaluates on-line parameter tuning in the real-time domain of the GVG-AI project. More precisely, the MAB and NTBEA allocation strategies are used to tune on-line the parameters of the MCTS-based MaastCTS2 agent. Table 6.12 shows the results obtained by testing MaastCTS2 with default fixed parameter values (MP), with sub-optimal fixed parameter values ($MP_{\text{SUB-OPT}}$), tuned with the MAB strategy (MP_{MAB}) and tuned with the NTBEA strategy (MP_{NTBEA}) with the game-tick duration set to $40ms$ (i.e. the default time settings of the GVG-AI competition). The on-line tuning agents have been tested for two (C and W), three (C, W and ϵ_{NST}) and five (C, W, ϵ_{NST}, N and L) tuned parameters. Looking at the overall win percentage, all agent instances seem quite close in performance. Even the performance of $MP_{\text{SUB-OPT}}$ is not as low as it would be expected for an agent that is using sub-optimal parameters. While tuning a small number of parameters seemed beneficial in the Stanford GGP project domain, it does not make any difference in the GVG-AI project domain. Also increasing the number of tuned parameters does not impact the performance as much as it does on the games of the Stanford GGP project. The performance of MP_{NTBEA} stays overall unchanged independently of how many parameters are tuned. The performance of MP_{MAB}, given its high overhead, would be expected to suffer from a significant performance decrease in most of the games, especially when tuning many parameters. However, only for five tuned parameters its overall performance slightly decreases, and only in six games the performance is significantly worse than the one of MP (Bait, Chase, Crossfire, Escape, Missile Command and Roguelike).

Modality is the only game for which some instances of the on-line tuning agents show a significant increase in the performance. The win rate of $MP_{\text{SUB-OPT}}$ for this game suggests that some of the default parameter values are actually sub-optimal, and the parameter values that are expected to be sub-optimal are actually performing better. Parameter tuning is able to find better values for parameters that are set to a sub-optimal value. For Modality, it seems that tuning the parameter ϵ_{NST} causes the performance increase. Also interesting is to look at the performance of on-line tuning on the games where $MP_{\text{SUB-OPT}}$ performs significantly worse than MP: Camel Race, Missile Command, Plaque Attack, Roguelike and Wait For Breakfast. For these games almost all instances of MP_{MAB} and all instances of MP_{NTBEA} manage to reach a close performance to the one of MP. This suggests that it is still beneficial to tune parameters on-line rather than keeping them fixed to values that might be sub-optimal for the game being played.

However, in GVG-AI on-line parameter tuning does not seem to have in general an impact on the performance as significant as for the Stanford GGP project. One explanation for this might be that the effect of the tuned parameters on the search is only limited for the considered games. Moreover, all the considered games are single-player. It could be that parameter tuning affects the search more for two- or multi-player games, where the opponents are actually modeled in the search tree. Another reason could be the low number of simulations that the agents can perform in $40ms$. Table 6.13 shows the average median number of simulations per tick of the MP agent. Moreover, for all the considered instances of the on-line tuning agent,

Table 6.12: Win percentage of MaastCTS2 with fixed parameter values (MP), with sub-optimal fixed parameter values ($MP_{SUB\text{-}OPT}$), tuned on-line with the MAB strategy (MP_{MAB}) and tuned on-line with the NTBEA strategy (MP_{NTBEA}), with game tick set to $40ms$. MP_{MAB} and MP_{NTBEA} tune two three and five parameters.

Game	MP	$MP_{SUB\text{-}OPT}$	2 parameters		3 parameters		5 parameters	
			MP_{MAB}	MP_{NTBEA}	MP_{MAB}	MP_{NTBEA}	MP_{MAB}	MP_{NTBEA}
Aliens	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	99.8(±0.39)	100.0(±0.00)	99.6(±0.55)	100.0(±0.00)
Bait	31.8(±4.09)	25.8(±3.84)	32.8(±4.12)	31.2(±4.07)	31.2(±4.07)	32.0(±4.09)	22.0(±3.63)	31.6(±4.08)
Butterflies	98.6(±1.03)	99.0(±0.87)	99.4(±0.68)	98.8(±0.96)	99.0(±0.87)	99.8(±0.39)	99.4(±0.68)	100.0(±0.00)
Camel Race	44.4(±4.36)	33.8(±4.15)	41.0(±4.32)	40.6(±4.31)	39.4(±4.29)	41.6(±4.32)	39.2(±4.28)	42.4(±4.34)
Chase	28.0(±3.94)	28.2(±3.95)	29.2(±3.99)	24.4(±3.77)	26.8(±3.89)	28.2(±3.95)	20.2(±3.52)	26.8(±3.89)
Chopper	99.8(±0.39)	99.2(±0.78)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	99.6(±0.55)	99.6(±0.55)
Crossfire	31.8(±4.09)	27.6(±3.92)	29.6(±4.01)	29.8(±4.01)	28.8(±3.97)	29.4(±4.00)	17.2(±3.31)	28.4(±3.96)
Dig Dug	1.6(±1.10)	0.8(±0.78)	1.4(±1.03)	0.8(±0.78)	1.6(±1.10)	1.8(±1.17)	1.8(±1.17)	1.2(±0.96)
Escape	93.4(±2.18)	93.2(±2.21)	90.6(±2.56)	91.0(±2.51)	92.4(±2.33)	91.0(±2.51)	80.4(±3.48)	92.2(±2.35)
Hungry Birds	100.0(±0.00)	99.8(±0.39)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)
Infection	100.0(±0.00)	99.8(±0.39)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	99.8(±0.39)	100.0(±0.00)
Intersection	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)
Lemmings	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)
Missile Command	96.8(±1.54)	89.6(±2.68)	97.0(±1.50)	97.2(±1.45)	94.4(±2.02)	96.6(±1.59)	92.6(±2.30)	94.6(±1.98)
Modality	25.6(±3.83)	43.4(±4.35)	25.0(±3.80)	25.0(±3.80)	40.4(±4.31)	41.4(±4.32)	32.0(±4.09)	41.0(±4.32)
Plaque Attack	94.8(±1.95)	90.0(±2.63)	95.8(±1.76)	96.6(±1.59)	94.2(±2.05)	94.8(±1.95)	94.2(±2.05)	95.8(±1.76)
Roguelike	4.6(±1.84)	1.8(±1.17)	5.4(±1.98)	4.0(±1.72)	3.0(±1.50)	4.0(±1.72)	0.8(±0.78)	3.2(±1.54)
Sea Quest	58.4(±4.32)	59.6(±4.31)	56.8(±4.35)	53.4(±4.38)	57.6(±4.34)	50.2(±4.39)	53.8(±4.37)	54.6(±4.37)
Survive Zombies	42.4(±4.34)	40.2(±4.30)	42.0(±4.33)	41.4(±4.32)	40.4(±4.31)	42.2(±4.33)	38.2(±4.26)	40.8(±4.31)
Wait For Breakfast	99.0(±0.87)	83.4(±3.26)	98.6(±1.03)	98.4(±1.10)	99.0(±0.87)	99.0(±0.87)	98.0(±1.23)	98.0(±1.23)
Avg. Win%	62.6(±0.95)	60.8(±0.96)	62.2(±0.95)	61.6(±0.95)	62.4(±0.95)	62.6(±0.95)	59.4(±0.96)	62.5(±0.95)

Table 6.13: Variation (%) of MCTS iterations per tick ($40ms$) of MaastCTS2 tuned on-line with the MAB strategy (MP_{MAB}) and tuned on-line with the NTBEA strategy (MP_{NTBEA}) with respect to MaastCTS2 with fixed parameter values (MP). The variation is reported for MP_{MAB} and MP_{NTBEA} that tune two, three and five parameters.

Game	**MP**	**2 parameters**		**3 parameters**		**5 parameters**	
		MP_{MAB}	MP_{NTBEA}	MP_{MAB}	MP_{NTBEA}	MP_{MAB}	MP_{NTBEA}
Aliens	14.70	-4.7%	-6.5%	-5.7%	-5.5%	-41.0%	-1.6%
Bait	91.18	-11.7%	-31.9%	-48.1%	-22.5%	-88.7%	-36.8%
Butterflies	12.13	-17.9%	-20.9%	-18.5%	-20.2%	-41.5%	-4.5%
Camel Race	8.84	-23.4%	-25.7%	-23.6%	-25.3%	-39.9%	-3.8%
Chase	11.05	-10.2%	-12.2%	-9.3%	-11.6%	-42.5%	-6.6%
Chopper	5.38	-2.6%	-5.2%	-0.4%	-2.2%	-40.3%	-8.2%
Crossfire	6.53	-14.9%	-16.2%	-13.9%	-15.2%	-45.3%	-9.2%
Dig Dug	4.98	-11.2%	-14.7%	-9.8%	-15.3%	-44.4%	-14.7%
Escape	33.83	-15.8%	-58.7%	-30.7%	-56.9%	-75.7%	-53.0%
Hungry Birds	18.35	-11.7%	-13.7%	-12.2%	-12.8%	-44.5%	-8.1%
Infection	10.58	-7.9%	-7.7%	-8.1%	-6.0%	-46.8%	-4.0%
Intersection	15.62	-6.4%	-7.8%	-10.2%	-9.6%	-59.4%	-19.9%
Lemmings	11.09	-2.1%	-1.6%	-4.6%	-4.8%	-68.1%	-53.9%
Missile Command	17.70	-20.0%	-23.4%	-21.9%	-23.0%	-51.9%	-8.0%
Modality	302.56	-15.6%	-40.4%	-51.2%	-49.9%	-95.7%	-54.5%
Plaque Attack	5.97	-12.7%	-15.7%	-16.2%	-16.8%	-45.6%	-5.2%
Roguelike	4.43	-14.0%	-22.6%	-17.8%	-17.2%	-47.0%	-20.5%
Sea Quest	20.33	-9.9%	-11.4%	-13.6%	-12.3%	-57.1%	-5.1%
Survive Zombies	9.52	-7.1%	-8.7%	-7.0%	-8.9%	-44.0%	-5.1%
Wait For Breakfast	28.86	-10.7%	-13.7%	-9.3%	-8.0%	-58.0%	-8.1%

Table 6.14: Win percentage of MaastCTS2 with fixed parameter values (MP), with sub-optimal parameter values ($MP_{SUB\text{-}OPT}$), tuned on-line with the MAB strategy (MP_{MAB}) and tuned on-line with the NTBEA strategy (MP_{NTBEA}), with game tick set to 100ms. MP_{MAB} and MP_{NTBEA} tune two three and five parameters.

Game	**MP**	$\mathbf{MP_{SUB\text{-}OPT}}$	**2 parameters**		**3 parameters**		**5 parameters**	
			$\mathbf{MP_{MAB}}$	$\mathbf{MP_{NTBEA}}$	$\mathbf{MP_{MAB}}$	$\mathbf{MP_{NTBEA}}$	$\mathbf{MP_{MAB}}$	$\mathbf{MP_{NTBEA}}$
Bait	51.8(±4.38)	49.4(±4.39)	48.6(±4.39)	40.8(±4.31)	40.8(±4.31)	34.8(±4.18)	31.2(±4.07)	36.6(±4.23)
Camel Race	95.8(±1.76)	93.8(±2.12)	94.6(±1.98)	92.4(±2.33)	90.2(±2.61)	89.4(±2.70)	85.4(±3.10)	90.8(±2.54)
Chase	56.2(±4.35)	50.0(±4.39)	49.8(±4.39)	52.0(±4.38)	46.8(±4.38)	50.6(±4.39)	43.0(±4.34)	51.6(±4.38)
Crossfire	84.8(±3.15)	80.4(±3.48)	79.8(±3.52)	81.2(±3.43)	79.6(±3.54)	79.8(±3.52)	68.2(±4.09)	81.8(±3.39)
Dig Dug	0.0(±0.00)	0.2(±0.39)	0.2(±0.39)	0.2(±0.39)	0.2(±0.39)	0.2(±0.39)	0.4(±0.55)	0.0(±0.00)
Lemmings	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)
Modality	26.2(±3.86)	25.2(±3.81)	26.4(±3.87)	23.8(±3.74)	40.0(±4.30)	40.6(±4.31)	38.8(±4.28)	40.4(±4.31)
Roguelike	32.6(±4.11)	34.0(±4.16)	32.6(±4.11)	31.8(±4.09)	34.4(±4.17)	36.8(±4.23)	28.4(±3.96)	32.0(±4.09)
Sea Quest	58.6(±4.32)	59.8(±4.30)	57.2(±4.34)	59.8(±4.30)	60.8(±4.28)	61.4(±4.27)	61.8(±4.26)	56.2(±4.35)
Survive Zombies	49.0(±4.39)	49.0(±4.39)	44.4(±4.36)	46.4(±4.38)	46.6(±4.38)	46.0(±4.37)	44.8(±4.36)	45.4(±4.37)
Avg. Win%	45.5(±1.38)	44.2(±1.38)	43.4(±1.37)	42.8(±1.37)	43.9(±1.38)	44.0(±1.38)	40.2(±1.36)	43.5(±1.37)

the table reports the speed variation with respect to MP. The speed of the MP_{MAB} is shown to substantially decrease with the increase of tuned parameters, while the speed of MP_{NTBEA} is shown to be less affected by the number of tuned parameters. This explains the drop in performance of MP_{MAB} when tuning five parameters. However, the number of simulations might be in general too low for all the agents. A low number of simulations might have two implications for on-line parameter tuning: (i) the allocation strategies cannot find optimal values early enough in the game to make a positive difference in the performance, and (ii) even if there are bad values among the feasible ones, the number of simulations controlled by them is not high enough to be detrimental.

To verify how more simulations influence the performance, the same series of experiments has been performed increasing the tick duration to $100ms$. Only the games where, given more time, MaastCTS2 has more room for improvement are considered, and the ones where it is performing close to 100% with a tick of $40ms$ are excluded. Results of this series of experiments are presented in Table 6.14. A longer search time significantly increases the performance of MaastCTS2 in many of the games. With more search time there seems to be no particular difference in performance between MP and $MP_{SUB\text{-}OPT}$ in any of the games. There are also a few games for which more search time seems to make on-line parameter tuning detrimental. For example, for Bait, Camel Race and Chase the win percentage of many of the on-line tuning agents' instances significantly decreases with respect to MP. The overall results with game ticks of $100ms$ are still mostly in line with what is observed for $40ms$. All the agent instances are close in performance, and only when tuning five parameters the decrease in performance of MP_{MAB} with respect to MP becomes statistically significant (although the difference is still not as high as for the Stanford GGP project games). Once again, when ϵ_{NST} is tuned, the agents achieve a much higher win rate than MP in Modality. Differently from what was observed with a tick of $40ms$, with a longer search time the sub-optimal parameter values do not seem to have a positive effect on the performance in Modality. This might mean that optimal parameter values for this game also depend on how much search time is available and on-line parameter tuning is able to adjust them accordingly.

The performance increase in Modality, both with a $40ms$ and a $100ms$ tick duration, suggest that on-line parameter tuning has the potential to be useful even when decisions have to be made in a short amount of time.

6.4.9 Discussion

An important aspect that should be taken into account when tuning MCTS parameters on-line is the choice of such parameters. Not all the parameters that control some aspects of MCTS are suitable to be tuned on-line with the method proposed in this chapter. First of all, the chosen parameters should directly influence each MCTS simulation, and be used to decide how the simulation is performed. Therefore, parameters like decay factors for the collected statistics that are used once per turn or once every few simulations should not be considered. Such parameters can only be evaluated once their effect on the search has taken place. For example, it would be possible to evaluate a statistics decay factor that is applied after every

move, only at the end of the turn or even at the end of the complete game, making the number of samples that can be collected to evaluate its possible values too low to perform on-line tuning.

Second, parameters that influence the simulation, and thus how the game tree is visited, should do so by changing how actions are selected in each state, and not by changing which parts of the tree are allowed to be visited. An example of parameter that would change the portion of the tree that can be visited is the simulation depth. Changing this parameter for each simulation would cause the same path in the tree to have different payoffs every time it is visited with a different depth. Depending on the depth limit, the visit of a path will end at different states, which also have different payoffs. This might cause the parameter values to be wrongly evaluated, given that these payoffs are used as evaluation by the on-line tuning strategy.

This has been confirmed by experiments on the single-player planning track of the GVG-AI project. These experiments apply on-line parameter tuning to the sampleMCTS agent (see Subsection 3.2.4), considering the simulation depth as a parameter to be tuned on-line with values between 1 and 15. Results showed that, for most of the games, the most used value is 1, although an instance of the sampleMCTS agent that uses 1 as fixed default value for the depth is usually performing poorly. This is an example of how on-line parameter tuning might wrongly evaluate a parameter. What happens is that, the deeper an agent visits the game tree the more likely it is to find terminal states. Therefore, in the games where terminal states mostly correspond to losses, the on-line tuning mechanism will compute a lower evaluation the higher the depth value is. This phenomenon is further accentuated by the heuristic used by the agent to evaluate a state (Formula 3.1), which gives a high penalty to losing states and a high reward to winning states. In this way, a depth of 1 is preferred by the agent, even if with a higher depth value it would be able to detect losing states earlier in the search and possibly avoid them.

6.5 Chapter Conclusions and Future Research

This chapter presented an on-line tuning method for search-control parameters that enables MCTS to be self-adaptive during game play. The performance of this method was evaluated on the Stanford GGP project and on the GVG-AI project. Seven different allocation strategies were introduced and tested for parameter tuning: MAB, HE, NMC, LSI, EA, NTBEA and CMA-ES.

Comparing the on-line tuning approaches, NTBEA seems to have the best performance overall, but EA is also quite close. NTBEA performs well on most of the games and for different numbers of tuned parameters. This is likely due to the fact that it merges the use of evolutionary computation with the multi-armed bandit approach. It also seems that a discrete domain for the parameters is sufficient to achieve a good performance. It could be expected that, given its continuous parameter domains, CMA-ES could converge to more accurate optimal values for the parameters, which the discretization of the domain might exclude. However, results showed that it was not performing better than most of the discrete allocation strategies, especially for two and four tuned parameters.

Results for the Stanford GGP project show that, when tuning two parameters, with most of the allocation strategies the agents can reach at least the same performance or surpass the off-line tuned agent. When tuning four parameters for the more advanced agent, the overall performance of the on-line tuning agent is lower than the off-line tuned agent, but still quite close for some of the allocation strategies. This is especially remarkable because only a single run of a game is used to tune the parameters, instead of a few hundred. On-line tuning of six parameters is much harder, especially when the thinking time is low. Independently of the number of tuned parameters, agents that use on-line parameter tuning with NTBEA were also shown to perform overall better than agents with random fixed parameter settings. It may be concluded that the proposed approach is useful when off-line parameter tuning is infeasible, or in a context like GGP, where parameters cannot be tuned in advance for each game, or when off-line tuning incurs in the risk of overfitting the values to the set of games selected for the purpose of tuning.

Results also show that both the off-line tuned and the on-line tuning agents using NTBEA perform overall better than an agent with a different implementation of MCTS, CadiaPlayer. Moreover, against CadiaPlayer the agent that tunes two parameter on-line with NTBEA has a better performance than the off-line tuned agent. Thus, it may be concluded that on-line parameter tuning is robust against different types of opponents. This also suggests that on-line parameter tuning can adapt to the opponent style of play and could be a valid approach to create agents that change their behavior according to the level of their (human) opponent.

Results for the GVG-AI project suggest that it is harder to tune parameters on-line with much shorter time settings, even when the number of tuned parameters is small. However, it may still be better to tune parameters on-line when fixed parameter settings might be sub-optimal, such as was observed in the game Modality.

In future research, it might be interesting to investigate other evolutionary strategies for continuous domains, such as *Differential Evolution* (Storn and Price, 1997), to see if they can improve with respect to the performance of CMA-ES. Moreover, given the good performance of NTBEA, future work could look into using other parameter optimization methods that are based on a model of the parameters landscape. An example is Sequential Model-based Algorithm Configuration (SMAC) (Hutter, Hoos, and Leyton-Brown, 2011), which builds explicit regression models to predict the performance of parameters. This method was shown to be comparable to NTBEA when tuning the parameters of an agent for the Planet Wars game (Lucas *et al.*, 2019).

Finally, it would be interesting to see if the devised on-line parameter tuning method can be successfully applied to other domains as well. In addition, the fact that the self-adaptive agents are not able to choose which and how many parameters to tune is a limitation of this work. These choices can be seen as extra parameters of the agent and for the experiments in this chapter their values were selected manually by off-line testing. Future work should design agents that consider these choices as part of the on-line automatic adaptation. Moreover, performing this decision on-line could help automatically reduce the size of the combinatorial search space by excluding less relevant parameters.

It could also be interesting to investigate how much the randomization introduced

by on-line parameter tuning is actually influencing the search. The constantly changing parameter values because of on-line tuning might already make a difference in the search, whether or not they are optimal values. One of the allocation strategies, LSI, actually has a highly random component. Parameters in the first phase of this strategy, the generation phase, are evaluated uniformly at random. Although not the best over all the allocation strategies, LSI still performs well on a few games, among which the ones with the most notable performance are Quad and Connect Four. The randomization aspect introduced by on-line parameter tuning is further investigated in Chapter 7.

Chapter 7

Comparing Randomization Strategies for Search-Control Parameters in MCTS

This chapter is based on:

- Sironi, Chiara F., and Winands, Mark H.M. (2018b). Analysis of Self-Adaptive Monte Carlo Tree Search in General Video Game Playing. *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 397–400.
- Sironi, Chiara F., and Winands, Mark H.M. (2019). Comparing Randomization Strategies for Search-Control Parameters in MCTS. *2019 IEEE Conference on Games (COG)*.

Previous research has shown that adding randomization to certain components of the search might increase its diversification and improve the performance. Different approaches have been proposed to add randomization to the search, such as adding a random term to the state evaluation function used by the search algorithm (Beal and Smith, 1994), or adding some randomness when choosing actions during the selection or the play-out phase of MCTS (Bošanský *et al.*, 2016; Chen, 2012).

In a domain that tackles many games with different characteristics, like General Game Playing (GGP), trying to diversify the search might be a good strategy. This is also suggested by the results presented in Chapter 6 for on-line parameter tuning. These results showed that on-line parameter tuning is beneficial for MCTS, especially if the number of tuned parameters is low and if the agents can perform a sufficient number of simulations to evaluate the combinations of values. Part of the explanation for the performance of on-line parameter tuning might be that changing parameter settings helps the agents diversify the search process. Therefore, they explore more parts of the tree which would not be explored much when parameter values are fixed. There might be games for which this diversification is beneficial.

A way to verify whether this is true would be trying to randomize parameter values on-line. This would remove from the agents the ability to make an informed choice when selecting the values, while still letting the parameters change over time. This chapter answers the fourth research question by comparing four different strategies that randomize search-control parameters for MCTS in GGP. These strategies randomize parameters once per game run, once per turn, once per simulation and once per visited state, respectively. The randomization strategy that performs best is further analyzed to verify how it influences the performance with respect to fixed parameter values. Moreover, this randomization strategy is compared with on-line parameter tuning both directly and by evaluating the performance against an agent with fixed sub-optimal parameter values and against a benchmark GGP agent, CadiaPlayer (Björnsson and Finnsson, 2009). Finally, a comparison of the best performing randomization strategy with fixed parameter values and on-line parameter tuning is performed also in a real-time GGP environment.

The chapter is organized as follows. First, Section 7.1 discusses related work. Next, the parameter randomization strategies are described in Section 7.2. Subsequently, Section 7.3 reports the results of the performed experiments. Finally, in Section 7.4 the conclusion is given and future work is discussed.

7.1 Related Work

Previous research has shown how tree search might benefit from adding randomization to some of its aspects. A first example can be found in the work of Beal and Smith (1994), which shows the effects of using random numbers as intermediate state evaluations when performing minimax search in Chess. An agent using only random evaluations for intermediate states is shown to outperform the same agent that uses 0 as heuristic instead. Adding a random term to existing heuristic functions is shown to be beneficial as well. This effect is explained by considering that random evaluations are able to capture some aspects of the structure of the tree, biasing the search toward states where the player has more mobility.

Bošanský *et al.* (2016) proposed the use of a random tie-breaking rule when MCTS has to select among multiple actions of a player that have the same UCT value. They test an UCT agent that, for each role, selects an action randomly among those that have the UCT value within a predefined small offset from the highest UCT value. Results on a set of simultaneous move games show that this agent converges to a better approximation of the optimal strategy with respect to the agent that uses a deterministic tie-breaking rule instead. This happens because with a tie-breaking rule the agent is mixing the strategies that it uses to sample the actions, instead of sampling always according to the same mixed strategy.

Chen (2012) proposed two randomization techniques for MCTS in the game of Go. The first technique consists in randomizing a set of parameters that control the selection phase. During a simulation, each of these parameters is randomized in a predefined range of values before selecting a move in each of the visited tree nodes. The second technique is used to add randomization to the play-out phase of MCTS by hierarchically randomizing the order of a set of predefined move generators

before selecting a move in each state visited during the play-out. These randomization techniques diversify the sampling of the actions and are shown to improve the performance of the MCTS agent for different search budgets and sizes of the Go board.

7.2 Search-Control Parameter Randomization

In order to implement search-control parameter randomization for a tree search algorithm, the following three steps are performed:

- Identify a finite set of d parameters, $P = \{P_1, ..., P_d\}$, that will be randomized.
- For each parameter $P_i \in P$, define the finite set of m_i different values that the parameter can assume, $\mathcal{V}_i = \{v_{i,1}, ..., v_{i,m_i}\}$.
- Design a *randomization strategy* that decides when and how to randomize the parameters.

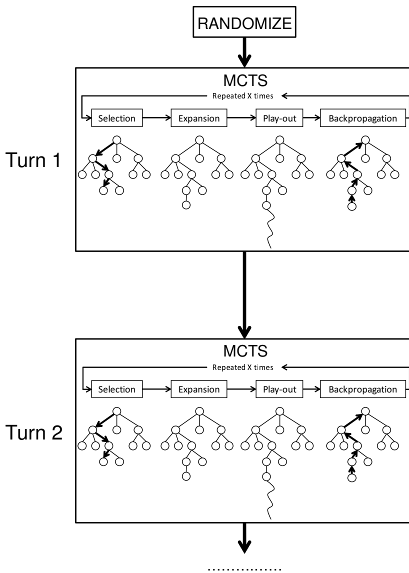
Note that also a continuous domain for the parameters could be considered. However, this thesis focuses only on a discrete domain because results presented in Chapter 6 showed that considering a continuous domain was not adding any benefit to the performance of on-line parameter tuning.

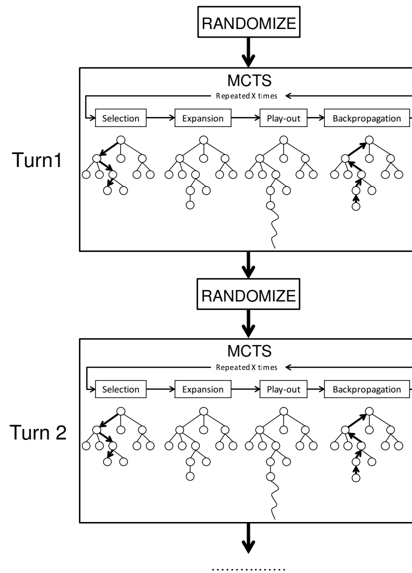
Four different strategies to randomize search-control parameters are considered in this chapter: *per game run*, *per turn*, *per simulation* and *per state*. All of them randomize the values of the selected parameters for each role in the game separately. This choice has been made to be consistent with the on-line parameter tuning mechanism evaluated in Chapter 6, which was designed to tune parameters independently for each role. The randomization strategies are described below and an overview of when randomization takes place for each of them is given in Figure 7.1.

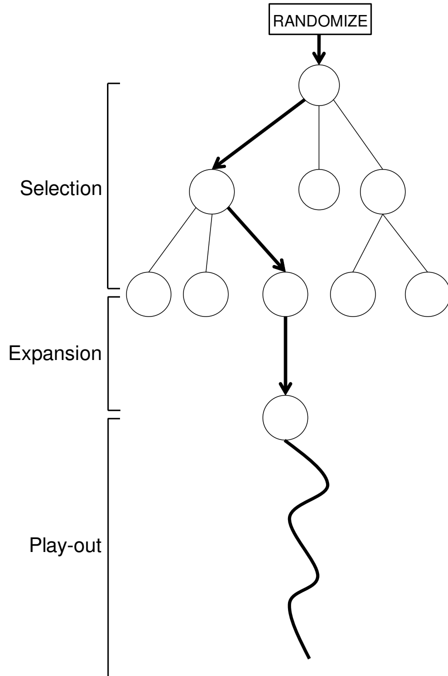
Per game run. Before the start of the search for an entire run of the game, this strategy sets for each role each considered parameter to a random value in its set of feasible values. The combinations of parameters for each role are then kept fixed for the search performed in each turn, until the run of the game is over.

Per turn. For each role in the game, this strategy sets a random combination of feasible values for the considered parameters before starting the search for each game turn.

Per simulation. This strategy sets for each role a new random combination of feasible parameter values before the start of each new MCTS simulation. This strategy is the one that most resembles on-line parameter tuning, which is also modifying the combination of parameter values for each role before each simulation. The difference is that on-line parameter tuning uses the previously learned information to bias the selection of parameter values.







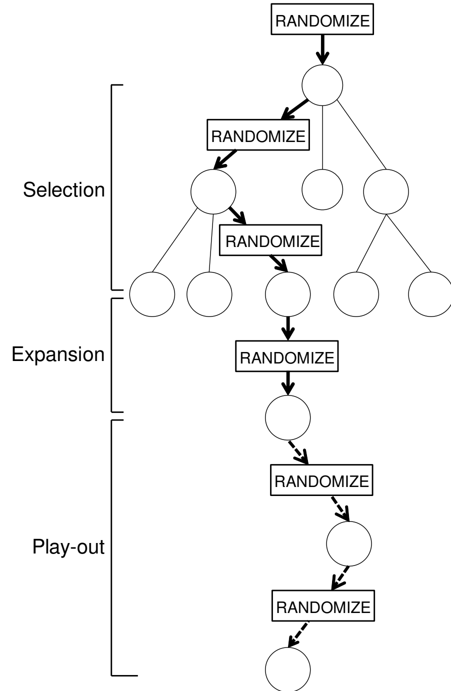


Figure 7.1: Randomization strategies.

Per state. Every time a state is visited during a simulation, both during the selection and the play-out phase, this strategy randomizes for each role the values of the parameters used to perform the search in the state. This strategy is similar to the one proposed by Chen (2012), although Chen assigns the same random parameter values to all the roles. As previously mentioned, this chapter considers parameter randomization for all roles independently to be consistent with the on-line parameter tuning mechanism analyzed in Chapter 6.

7.3 Empirical Evaluation

This section presents an analysis of parameter randomization for MCTS, by performing multiple series of experiments. Subsection 7.3.1 describes the experimental setup. Results obtained by performing experiments on the Stanford GGP project are given in Subsections 7.3.2, 7.3.3, 7.3.4 and 7.3.5. An analysis of the search tree built by different agent instances is presented in Subsection 7.3.6. Finally, experimental results on the GVG-AI project are reported in Subsection 7.3.7.

7.3.1 Setup

Like on-line search-control parameter tuning, on-line search-control parameter randomization has been tested on MCTS agents both for the Stanford GGP project and for the GVG-AI project. First, a thorough evaluation of parameter randomization is performed on the Stanford GGP project, on which the four parameter randomization strategies are compared. Subsequently, the randomization strategy that performed best is further tested and compared with on-line parameter tuning also on the GVG-AI project. When comparing parameter randomization with online parameter tuning, only the NTBEA strategy is considered because it is the one that seemed to perform best in the previous chapter. The settings for NTBEA are the same as in Subsection 6.4.1. This subsection presents the experimental setup for both the Stanford GGP project, and the GVG-AI project.

Stanford GGP Project Setup

The experiments presented in this chapter for the Stanford GGP project use as baseline the same agent instance of Chapter 6, AP, which implements MCTS with the GRAVE selection strategy and the MAST play-out strategy. Also the parameters that can be randomized/tuned on-line are the same: the UCT exploration constant C, the probability used by MAST, ϵ_{MAST}, the equivalence parameter K and the visit threshold *ref* used by GRAVE, the value offset for random tie-breaking in UCT, VO, and the visit threshold T used to decide whether to use the play-out strategy for a node during the selection phase of MCTS. The default values and the discrete domains for these parameters are the same given in Table 6.1 and in this chapter are reported again for completeness in Table 7.1, together with the sub-optimal value of each parameter used in the series of experiments presented in Subsection 7.3.5. When randomizing or tuning the parameters on-line, the agent instances are represented with a subscript indicating the type of randomization or allocation strategy used,

Table 7.1: Default values, discrete domains and sub-optimal values of the parameters considered in the experiments on the Stanford GGP project.

Param.	**Default value**	**Discrete domain**	**Sub-optimal value**
C	0.2	$\{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$	0.9
ϵ_{MAST}	0.4	$\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$	0.0
K	250	$\{0, 10, 50, 100, 250, 500, 750, 1\,000, 2\,000, \infty\}$	∞
ref	50	$\{0, 50, 100, 250, 500, 1\,000, 10\,000, \infty\}$	∞
VO	0.01	$\{0.001, 0.005, 0.01, 0.015, 0.02, 0.025\}$	0.025
T	0	$\{0, 5, 10, 20, 30, 40, 50, 100, 200, \infty\}$	200

respectively. The four randomization strategies are identified, in order, as GAME-RND, TURN-RND, SIM-RND and STATE-RND.

In one of the series of experiments, the last available version of CadiaPlayer[1] (Finnsson, 2012b) is used as a benchmark to compare the performance of the best parameter randomizing agent instance with the one of the on-line tuning and the off-line tuned agent instances.

All the experiments presented in the next subsections for the Stanford GGP project are performed on the same set of 14 games used in Chapter 6 (Schreiber, 2016): 3D Tic Tac Toe, Breakthrough, Knightthrough, Chinook, Chinese Checkers with 3 players, Checkers, Connect 5, Quad (the version played on a 7×7 board), Sheep and Wolf, Tic-Tac-Chess-Checkers-Four (TTCC4) with 2 and 3 players, Connect Four, Pentago and Reversi. Each experiment matches two agent types at a time against each other, ensuring that each of them is assigned to each role in the game the same number of times over all the game runs. For 3-player games, all possible assignments of agent types to the roles are considered, except the two configurations that assign the same type to each role. All configurations are run the same number of times until each agent type has played at least 500 games in total. All agent instances use the software implementation of the PropNet. A new PropNet is created before each game run and the same structure is given to both the involved agent instances to prevent any of them from having an advantage due to a faster structure. For each agent, start- and play-clock are set to $1s$, except for CadiaPlayer, which uses $10s$ start- and play-clock in order to reach a number of simulations similar to the other agents.

Different series of experiments have been performed, one that compares all the randomization strategies, one that compares randomization of single parameters with fixed parameter values, one that compares parameter randomization with on-line parameter tuning directly, and one that compares parameter randomization and on-line parameter tuning with fixed parameter values, both directly and against different types of opponents (i.e. an agent with sub-optimal values and CadiaPlayer). Experimental results always report the average win percentage of one of the two

[1]Version of 18-11-2012. Downloaded from `http://cadia.ru.is/wiki/public:cadiaplayer:main`

Table 7.2: Default values, discrete domains and sub-optimal values of the parameters considered in the experiments on the GVG-AI project.

Param.	**Default value**	**Discrete domain**	**Sub-optimal value**
C	0.6	$\{0.2, 0.4, 0.5, 0.6, 0.7, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0\}$	2.0
ϵ_{NST}	0.5	$\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$	0.0
W	1	$\{0.1, 0.25, 0.5, 1, 3, 5, 7.5, 10, 20, 50\}$	0.1
N	7	$\{1, 3, 5, 7, 10, 15, 20, 30, 50\}$	50
L	3	$\{1, 2, 3, 4, 5\}$	1

involved agent types with a 95%-confidence interval. The average win percentage of an agent type for a game is computed by assigning 1 point to the agent type that achieved the highest score and 0 to the other. If they both achieve the same score, they are given 0.5 points each. Bold results indicate the agent type with the highest win rate for the corresponding game and number of randomized/tuned parameters. Experiments were performed on a Linux server consisting of 64 AMD Opteron 6274 2.2-GHz cores, except the ones presented in Tables 7.5 and 7.6, which were performed on a Linux server consisting of 48 AMD Opteron 6344 2.6-GHz cores.

GVG-AI Project Setup

As for testing on-line parameter tuning in a real-time domain in Chapter 6, parameter randomization has been implemented in the GVG-AI framework for the single-player planning track agent MaastCTS2. The same parameters reported in Subsection 6.4.1 are considered to be randomized or tuned: the UCT exploration constant C, the probability used by NST, ϵ_{NST}, the *Progressive History* weight W, and the visit threshold N and the maximum N-Gram length L used by NST. The default values, the discrete domain and the sub-optimal values for these parameters are also the same as in Chapter 6, and are reported in Table 7.2.

The agent is tested on the same set of 20 heterogeneous single-player games used in Chapter 6 and taken from the GVG-AI framework (Perez-Liebana *et al.*, 2016; Perez-Liebana, 2018): Aliens, Bait, Butterflies, Camel Race, Chase, Chopper, Crossfire, Dig Dug, Escape, Hungry Birds, Infection, Intersection, Lemmings, Missile Command, Modality, Plaque Attack, Roguelike, Sea Quest, Survive Zombies, and Wait for Breakfast. In each experiment, the considered agent is tested equally on all 5 levels of these games until 500 samples per game have been collected.

Two series of experiments have been performed, one for which the agent has $40ms$ per game tick to choose an action and one for which the agent has $100ms$ per game tick. Contrary to the setting of the GVG-AI Single-Playing Planning competition, in these experiments if the agent exceeds the time limit per game tick it will not be disqualified and can still apply its selected move. Experimental results always report the average win percentage of the tested agent type with a 95%-confidence interval. All the experiments for the GVG-AI project were performed on a Linux server consisting of 64 AMD Opteron 6274 2.2-GHz cores.

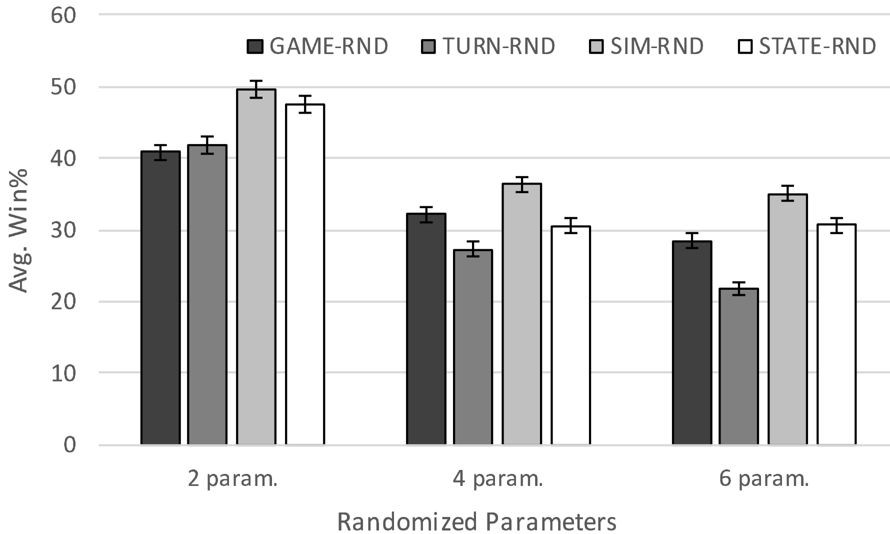


Figure 7.2: Win percentage of the agents with the parameter randomization strategies against the agent with fixed default parameter values.

7.3.2 Comparison of Parameter Randomization Strategies

This series of experiments evaluates the randomization strategies by directly matching the agent instances that implement them against each other and by comparing them when matched against the agent instance with fixed default parameters. Table 7.3 shows the results obtained by performing a round-robin tournament with all the instances of the AP agent that use one of the parameter randomization strategies (i.e. $\mathrm{AP}_{\text{GAME-RND}}$, $\mathrm{AP}_{\text{TURN-RND}}$, $\mathrm{AP}_{\text{SIM-RND}}$ and $\mathrm{AP}_{\text{GAME-RND}}$). Each block of the table corresponds to a different number of parameters being randomized, each row corresponds to an agent, each column to a game. Results represent the average win percentage of the row agent against all other agents. The agents consider K and *ref* when randomizing two parameters, K, *ref*, C and ϵ_{MAST} when randomizing four parameters and K, *ref*, C, ϵ_{MAST}, VO and T when randomizing six.

As can be seen, for most of the games, independently of the number of parameters being considered, the agent that randomizes parameters per simulation seems to be the best performing one. This does not contradict the findings of Chen (2012). Although the author mentions that randomizing parameters per simulation did not perform as well as per state, the comparison was performed only in the game of Go. When tested on more games, there are still a few games where randomization per state performs significantly better (e.g. Quad for two parameters, Connect Five for four, and Knightthrough for six), but randomizing per simulation seems overall best.

The performance of the randomizing agents against the agent with default fixed values is shown in Fig. 7.2. For different number of parameters, the figure reports the average win percentage of each of the randomizing agents over all the tested games.

Table 7.3: Comparison of parameter randomization strategies against each other for different numbers of randomized parameters.

2 parameters

	3D Tic Tac Toe	Breakthrough	Knightthrough	Chinook	Chin.Checkers3P	Checkers	Connect Five
$\text{AP}_{\text{GAME-RND}}$	46.8(±2.38)	49.7(±2.53)	**53.3**(±2.53)	44.9(±2.33)	49.5(±2.52)	38.7(±2.35)	45.3(±1.92)
$\text{AP}_{\text{TURN-RND}}$	46.8(±2.39)	48.1(±2.53)	51.5(±2.53)	40.7(±2.29)	**51.1**(±2.52)	43.4(±2.40)	42.2(±1.90)
$\text{AP}_{\text{SIM-RND}}$	**59.5**(±2.31)	**52.9**(±2.53)	51.1(±2.53)	**58.1**(±2.33)	49.9(±2.52)	58.8(±2.36)	54.6(±1.82)
$\text{AP}_{\text{STATE-RND}}$	46.9(±2.35)	49.4(±2.53)	44.1(±2.51)	56.4(±2.33)	49.5(±2.52)	**59.1**(±2.35)	**57.8**(±1.85)

	Quad	Sheep and Wolf	TTCC4 2P	TTCC4 3P	Connect Four	Pentago	Reversi
$\text{AP}_{\text{GAME-RND}}$	38.4(±2.32)	47.3(±2.53)	41.5(±2.42)	49.6(±2.45)	41.3(±2.38)	48.0(±2.49)	44.4(±2.40)
$\text{AP}_{\text{TURN-RND}}$	37.5(±2.31)	49.0(±2.53)	45.6(±2.46)	48.5(±2.46)	45.8(±2.41)	48.1(±2.48)	46.0(±2.41)
$\text{AP}_{\text{SIM-RND}}$	56.2(±2.37)	**53.5**(±2.52)	**60.5**(±2.39)	50.0(±2.45)	**60.8**(±2.34)	**52.2**(±2.49)	**57.0**(±2.39)
$\text{AP}_{\text{STATE-RND}}$	**67.9**(±2.20)	50.1(±2.53)	52.3(±2.45)	**51.9**(±2.46)	52.1(±2.42)	51.7(±2.49)	52.5(±2.39)

4 parameters

	3D Tic Tac Toe	Breakthrough	Knightthrough	Chinook	Chin.Checkers3P	Checkers	Connect Five
$\text{AP}_{\text{GAME-RND}}$	43.9(±2.40)	49.5(±2.53)	**55.9**(±2.51)	50.3(±2.41)	50.7(±2.52)	52.4(±2.40)	46.4(±2.17)
$\text{AP}_{\text{TURN-RND}}$	34.9(±2.33)	45.5(±2.52)	50.7(±2.53)	42.7(±2.38)	41.4(±2.48)	30.4(±2.21)	32.6(±2.02)
$\text{AP}_{\text{SIM-RND}}$	**68.2**(±2.22)	**61.1**(±2.47)	51.7(±2.53)	**63.9**(±2.32)	**59.0**(±2.48)	**59.8**(±2.36)	54.9(±2.11)
$\text{AP}_{\text{STATE-RND}}$	53.0(±2.39)	43.9(±2.51)	41.7(±2.50)	43.1(±2.41)	48.9(±2.52)	57.4(±2.40)	**66.1**(±2.02)

	Quad	Sheep and Wolf	TTCC4 2P	TTCC4 3P	Connect Four	Pentago	Reversi
$\text{AP}_{\text{GAME-RND}}$	44.3(±2.45)	52.6(±2.53)	48.9(±2.51)	49.2(±2.48)	47.2(±2.44)	51.5(±2.45)	49.3(±2.49)
$\text{AP}_{\text{TURN-RND}}$	40.3(±2.42)	50.1(±2.53)	46.8(±2.51)	46.1(±2.48)	46.1(±2.45)	39.9(±2.40)	48.1(±2.49)
$\text{AP}_{\text{SIM-RND}}$	**72.3**(±2.19)	**52.9**(±2.53)	**58.1**(±2.47)	**53.5**(±2.48)	**61.5**(±2.38)	**62.4**(±2.36)	**51.4**(±2.49)
$\text{AP}_{\text{STATE-RND}}$	43.0(±2.45)	44.5(±2.52)	46.2(±2.51)	51.2(±2.50)	45.1(±2.45)	46.1(±2.45)	51.3(±2.50)

6 parameters

	3D Tic Tac Toe	Breakthrough	Knightthrough	Chinook	Chin.Checkers3P	Checkers	Connect Five
$\text{AP}_{\text{GAME-RND}}$	44.3(±2.41)	52.5(±2.53)	44.5(±2.52)	52.6(±2.42)	45.4(±2.51)	43.2(±2.40)	40.7(±2.16)
$\text{AP}_{\text{TURN-RND}}$	28.0(±2.21)	45.1(±2.52)	41.3(±2.49)	37.3(±2.35)	40.0(±2.47)	25.3(±2.10)	30.4(±2.06)
$\text{AP}_{\text{SIM-RND}}$	**71.8**(±2.15)	**53.6**(±2.52)	54.1(±2.52)	54.7(±2.41)	**57.8**(±2.49)	**67.2**(±2.26)	**67.9**(±2.01)
$\text{AP}_{\text{STATE-RND}}$	55.9(±2.38)	48.7(±2.53)	**60.1**(±2.48)	**55.4**(±2.43)	56.8(±2.50)	64.3(±2.33)	61.0(±2.14)

	Quad	Sheep and Wolf	TTCC4 2P	TTCC4 3P	Connect Four	Pentago	Reversi
$\text{AP}_{\text{GAME-RND}}$	38.6(±2.40)	49.8(±2.53)	49.5(±2.52)	48.1(±2.49)	42.5(±2.43)	46.4(±2.46)	43.2(±2.48)
$\text{AP}_{\text{TURN-RND}}$	34.1(±2.34)	46.5(±2.53)	39.0(±2.46)	45.3(±2.48)	37.8(±2.39)	36.1(±2.37)	42.2(±2.48)
$\text{AP}_{\text{SIM-RND}}$	**75.3**(±2.11)	**55.6**(±2.52)	**60.4**(±2.47)	53.1(±2.50)	**67.2**(±2.28)	**64.0**(±2.37)	57.2(±2.47)
$\text{AP}_{\text{STATE-RND}}$	52.0(±2.47)	48.1(±2.53)	51.1(±2.53)	**53.5**(±2.50)	52.5(±2.46)	53.5(±2.47)	**57.5**(±2.47)

Detailed results per game are presented in Appendix F. Once again, the agent that randomizes parameters per simulation is the one showing the best performance overall when compared to the other randomizing agents. However, none of the agents seems to be overall better than the agent that uses fixed default values. Moreover, the results suggest that the performance of the randomizing agents drops with the increase in the number of randomized parameters.

7.3.3 Randomization per Simulation vs Fixed Parameters

These series of experiments further analyze the randomization strategy that performed overall best in the previous series of experiments: randomization per simulation. The purpose is to verify if randomization of parameter values during the search brings any contribution to the performance of the MCTS agent with respect to keeping parameter values fixed for the whole game. Each series of experiments focuses on one single parameter, considering the agents that keep the parameter fixed to one of its feasible values and the agent that randomizes such parameter per simulation ($\text{AP}_{\text{SIM-RND}}$). All other parameters for $\text{AP}_{\text{SIM-RND}}$ are set to their default values. These agents are matched against each other in a round-robin tournament and, for each game, the average win percentage of each agent against all other agents is reported in the results.

Tables 7.4, 7.5 and 7.6 show the results of such experiments for the parameter C, ϵ_{MAST} and K, respectively. Being quite time-consuming, these series of experiments have been performed only for the parameters that seemed to be more relevant for the search. First of all, it is interesting to notice that for a few games randomizing the parameter values during the search achieves one of the highest win percentages. This can be observed in Quad and Pentago for the parameter C, in TTCC4 with 3 players and Pentago for the parameter ϵ_{MAST} and in about half of the games for the parameter K. Randomization seems to be particularly effective for this parameter.

In general, for many of the games randomization per simulation seems to perform better than a few of the fixed values of the parameter. Moreover, there are a few games where randomizing parameter values, even if not the best choice, still performs better than the default fixed value. For some games, this happens because the default value, despite being optimal over all the set of games on which it was tuned, is actually not optimal for the specific game. Examples are Quad, Connect Four and Pentago for C, Quad, Sheep and Wolf, TTCC4 with 3 players and Connect Four for ϵ_{MAST}, and Quad, TTCC4 with 2 players, Connect Four and Pentago for K.

It is also interesting to notice that part of the results presented in this subsection can be compared to some of the results presented in Chapter 5 to verify their consistency, and, in some cases, give more insights. For example, results in Table 7.5 can be compared with the ones in Table 5.6. First of all, results in Table 7.5 seem to confirm that combining the GRAVE selection strategy with the MAST play-out strategy has in general a positive effect on the search. Secondly, the agents that use $\epsilon_{\text{MAST}} = 0.4$ and $\epsilon_{\text{MAST}} = 1.0$ correspond to the agents $\text{P}_{\text{GRAVE-MAST}}$ and P_{GRAVE}, respectively (note that MAST with $\epsilon_{\text{MAST}} = 1.0$ corresponds to a random play-out strategy). Therefore, their results can be directly compared with the ones presented in the third column of Table 5.6. For the games that are present in both tables, Ta-

Table 7.4: Comparing all feasible values of C with value randomization of C per simulation.

	3D Tic Tac Toe	Breakthrough	Knightthrough	Chinook	Chin.Checkers3P	Checkers	Connect Five
$C = 0.1$	51.5(±1.38)	66.0(±1.38)	63.9(±1.40)	42.2(±1.35)	48.2(±1.45)	54.2(±1.37)	45.7(±1.07)
$C = 0.2$	**55.6**(±1.36)	**78.2**(±1.21)	**69.0**(±1.35)	63.8(±1.31)	**58.9**(±1.43)	**65.8**(±1.31)	**53.0**(±1.05)
$C = 0.3$	54.9(±1.36)	71.0(±1.33)	62.5(±1.41)	**66.9**(±1.30)	57.8(±1.44)	61.3(±1.34)	51.9(±1.04)
$C = 0.4$	52.5(±1.35)	58.3(±1.44)	54.0(±1.46)	63.4(±1.32)	54.7(±1.45)	56.4(±1.37)	52.7(±1.05)
$C = 0.5$	50.8(±1.35)	45.6(±1.46)	47.6(±1.46)	56.5(±1.36)	51.5(±1.45)	50.7(±1.38)	52.7(±1.04)
$C = 0.6$	50.0(±1.37)	39.0(±1.43)	42.4(±1.44)	47.6(±1.37)	48.5(±1.45)	47.2(±1.38)	51.9(±1.06)
$C = 0.7$	47.5(±1.36)	35.8(±1.40)	38.6(±1.42)	39.3(±1.34)	45.1(±1.45)	41.3(±1.36)	49.6(±1.10)
$C = 0.8$	46.8(±1.36)	33.2(±1.38)	37.9(±1.42)	35.2(±1.30)	44.8(±1.45)	38.7(±1.35)	47.1(±1.09)
$C = 0.9$	43.7(±1.35)	30.0(±1.34)	41.0(±1.44)	33.0(±1.28)	41.5(±1.43)	36.9(±1.34)	46.1(±1.09)
$AP_{SIM\text{-}RND}$	46.8(±1.35)	42.8(±1.45)	43.0(±1.45)	52.1(±1.38)	49.0(±1.45)	47.6(±1.38)	49.3(±1.07)
	Quad	Sheep and Wolf	TTCC4 2P	TTCC4 3P	Connect Four	Pentago	Reversi
$C = 0.1$	24.0(±1.16)	**59.6**(±1.43)	39.1(±1.39)	49.6(±1.42)	29.5(±1.26)	34.6(±1.35)	**61.2**(±1.40)
$C = 0.2$	37.5(±1.34)	55.3(±1.45)	61.5(±1.39)	**52.1**(±1.41)	48.5(±1.40)	51.6(±1.41)	59.9(±1.41)
$C = 0.3$	53.2(±1.39)	51.7(±1.46)	**62.7**(±1.38)	51.1(±1.42)	54.5(±1.39)	**54.6**(±1.39)	54.1(±1.43)
$C = 0.4$	59.9(±1.36)	49.7(±1.46)	58.3(±1.40)	51.0(±1.42)	**56.4**(±1.39)	53.5(±1.38)	50.1(±1.44)
$C = 0.5$	59.5(±1.37)	47.8(±1.46)	52.0(±1.43)	51.2(±1.42)	54.5(±1.40)	50.9(±1.39)	49.0(±1.44)
$C = 0.6$	59.2(±1.38)	48.2(±1.46)	48.9(±1.43)	48.8(±1.42)	54.2(±1.40)	51.1(±1.39)	46.4(±1.44)
$C = 0.7$	55.5(±1.40)	46.4(±1.46)	45.7(±1.43)	49.7(±1.42)	50.7(±1.41)	51.7(±1.39)	44.0(±1.43)
$C = 0.8$	48.5(±1.41)	46.9(±1.46)	42.0(±1.42)	49.7(±1.42)	49.5(±1.41)	51.2(±1.39)	44.2(±1.43)
$C = 0.9$	40.7(±1.39)	45.3(±1.45)	40.8(±1.41)	46.8(±1.42)	49.7(±1.41)	46.4(±1.39)	45.3(±1.43)
$AP_{SIM\text{-}RND}$	**62.0**(±1.35)	49.0(±1.46)	49.1(±1.43)	50.0(±1.43)	52.5(±1.40)	54.5(±1.38)	45.8(±1.43)

Table 7.5: Comparing all feasible values of ϵ_{MAST} with value randomization of ϵ_{MAST} per simulation.

	3D Tic Tac Toe	Breakthrough	Knightthrough	Chinook	Chin.Checkers3P	Checkers	Connect Five
$\epsilon_{\text{MAST}} = 0.0$	43.0(±1.25)	**64.2**(±1.27)	**73.7**(±1.16)	38.8(±1.17)	32.3(±1.23)	42.2(±1.22)	37.8(±1.00)
$\epsilon_{\text{MAST}} = 0.1$	47.0(±1.25)	59.3(±1.30)	67.4(±1.24)	58.8(±1.20)	45.8(±1.31)	58.4(±1.23)	46.6(±0.98)
$\epsilon_{\text{MAST}} = 0.2$	46.3(±1.25)	57.3(±1.31)	63.2(±1.27)	**64.7**(±1.16)	50.8(±1.32)	64.2(±1.19)	50.4(±0.98)
$\epsilon_{\text{MAST}} = 0.3$	48.2(±1.25)	57.4(±1.31)	57.3(±1.31)	63.4(±1.17)	55.1(±1.31)	**64.8**(±1.19)	51.1(±0.97)
$\epsilon_{\text{MAST}} = 0.4$	48.4(±1.25)	56.6(±1.31)	54.7(±1.32)	61.1(±1.19)	55.5(±1.31)	61.2(±1.21)	54.1(±0.96)
$\epsilon_{\text{MAST}} = 0.5$	49.8(±1.26)	55.2(±1.31)	53.7(±1.32)	58.8(±1.20)	**57.4**(±1.30)	56.7(±1.23)	56.3(±0.97)
$\epsilon_{\text{MAST}} = 0.6$	53.1(±1.25)	53.0(±1.32)	50.9(±1.32)	54.3(±1.22)	54.5(±1.31)	51.1(±1.24)	57.6(±0.95)
$\epsilon_{\text{MAST}} = 0.7$	55.4(±1.25)	51.3(±1.32)	47.5(±1.32)	50.2(±1.22)	53.9(±1.31)	46.5(±1.25)	**58.5**(±0.97)
$\epsilon_{\text{MAST}} = 0.8$	**59.7**(±1.23)	46.8(±1.32)	40.9(±1.30)	43.2(±1.21)	50.9(±1.32)	42.8(±1.23)	58.1(±1.00)
$\epsilon_{\text{MAST}} = 0.9$	57.4(±1.25)	37.3(±1.28)	31.5(±1.23)	36.3(±1.18)	48.2(±1.32)	39.3(±1.22)	53.9(±1.05)
$\epsilon_{\text{MAST}} = 1.0$	44.0(±1.29)	22.9(±1.11)	16.7(±0.98)	25.4(±1.06)	42.5(±1.30)	35.8(±1.19)	36.2(±1.13)
$AP_{\text{SIM-RND}}$	47.7(±1.26)	38.7(±1.29)	42.6(±1.31)	44.9(±1.20)	53.0(±1.31)	36.8(±1.20)	39.3(±1.01)
	Quad	Sheep and Wolf	TTCC4 2P	TTCC4 3P	Connect Four	Pentago	Reversi
$\epsilon_{\text{MAST}} = 0.0$	18.6(±0.98)	38.7(±1.29)	40.5(±1.26)	41.6(±1.25)	26.9(±1.11)	46.1(±1.27)	43.2(±1.28)
$\epsilon_{\text{MAST}} = 0.1$	26.2(±1.10)	41.4(±1.30)	60.7(±1.25)	47.3(±1.27)	29.9(±1.15)	49.2(±1.27)	54.0(±1.30)
$\epsilon_{\text{MAST}} = 0.2$	33.7(±1.19)	42.2(±1.31)	67.0(±1.20)	50.2(±1.27)	35.6(±1.20)	51.0(±1.28)	60.5(±1.27)
$\epsilon_{\text{MAST}} = 0.3$	39.7(±1.23)	44.7(±1.31)	**67.3**(±1.19)	52.2(±1.27)	39.5(±1.23)	52.1(±1.28)	**61.3**(±1.27)
$\epsilon_{\text{MAST}} = 0.4$	47.6(±1.26)	47.1(±1.32)	62.9(±1.23)	53.2(±1.27)	45.4(±1.25)	**52.5**(±1.28)	59.1(±1.28)
$\epsilon_{\text{MAST}} = 0.5$	53.8(±1.26)	48.8(±1.32)	58.2(±1.26)	**54.0**(±1.27)	51.7(±1.25)	51.4(±1.28)	56.9(±1.29)
$\epsilon_{\text{MAST}} = 0.6$	61.5(±1.23)	51.9(±1.32)	50.4(±1.28)	52.7(±1.27)	56.8(±1.25)	49.3(±1.27)	52.9(±1.30)
$\epsilon_{\text{MAST}} = 0.7$	67.0(±1.18)	55.1(±1.31)	43.1(±1.27)	51.5(±1.27)	62.3(±1.22)	49.7(±1.27)	49.7(±1.30)
$\epsilon_{\text{MAST}} = 0.8$	**69.4**(±1.17)	56.3(±1.31)	37.9(±1.25)	49.2(±1.27)	65.0(±1.20)	48.9(±1.28)	45.4(±1.29)
$\epsilon_{\text{MAST}} = 0.9$	68.4(±1.19)	60.4(±1.29)	30.8(±1.19)	48.4(±1.28)	**67.7**(±1.18)	49.2(±1.28)	38.2(±1.26)
$\epsilon_{\text{MAST}} = 1.0$	57.2(±1.29)	**60.8**(±1.29)	25.8(±1.13)	45.6(±1.27)	66.5(±1.19)	48.1(±1.28)	29.2(±1.18)
$AP_{\text{SIM-RND}}$	56.8(±1.25)	52.4(±1.32)	55.2(±1.27)	**54.0**(±1.27)	52.9(±1.26)	**52.5**(±1.27)	49.6(±1.30)

Table 7.6: Comparing all feasible values of K with value randomization of K per simulation.

	3D Tic Tac Toe	Breakthrough	Knightthrough	Chinook	Chin.Checkers3P	Checkers	Connect Five
$K = 0$	47.8(±1.32)	46.2(±1.38)	57.8(±1.37)	40.1(±1.26)	51.1(±1.38)	33.1(±1.26)	48.6(±1.06)
$K = 10$	51.2(±1.33)	50.0(±1.39)	**59.9**(±1.36)	40.9(±1.27)	51.4(±1.38)	41.5(±1.31)	52.6(±1.04)
$K = 50$	53.2(±1.31)	53.6(±1.38)	56.8(±1.37)	45.3(±1.28)	**52.4**(±1.38)	55.7(±1.31)	54.5(±1.01)
$K = 100$	53.2(±1.31)	54.2(±1.38)	54.5(±1.38)	51.9(±1.28)	52.2(±1.38)	61.7(±1.28)	55.0(±1.01)
$K = 250$	56.4(±1.30)	56.8(±1.37)	53.8(±1.38)	55.8(±1.27)	52.0(±1.38)	**66.6**(±1.24)	**58.0**(±0.98)
$K = 500$	56.5(±1.30)	**56.9**(±1.37)	51.9(±1.39)	57.4(±1.26)	**52.4**(±1.38)	63.5(±1.26)	57.8(±0.98)
$K = 750$	**57.1**(±1.30)	55.2(±1.38)	51.0(±1.39)	57.2(±1.26)	52.0(±1.38)	58.7(±1.30)	55.2(±0.98)
$K = 1000$	56.5(±1.30)	54.0(±1.38)	50.5(±1.39)	55.5(±1.27)	51.9(±1.38)	53.9(±1.31)	53.3(±0.99)
$K = 2000$	53.6(±1.31)	52.7(±1.38)	50.4(±1.39)	54.6(±1.27)	49.7(±1.38)	42.7(±1.31)	47.9(±0.99)
$K = \infty$	8.9(±0.77)	16.7(±1.03)	16.6(±1.03)	29.3(±1.16)	34.4(±1.31)	8.4(±0.73)	10.2(±0.62)
$AP_{\text{SIM-RND}}$	55.7(±1.30)	53.6(±1.38)	46.8(±1.38)	**62.0**(±1.24)	50.3(±1.38)	64.3(±1.26)	56.9(±0.98)
	Quad	Sheep and Wolf	TTCC4 2P	TTCC4 3P	Connect Four	Pentago	Reversi
$K = 0$	49.0(±1.31)	**51.6**(±1.39)	28.4(±1.23)	49.9(±1.34)	41.6(±1.30)	37.4(±1.31)	53.4(±1.36)
$K = 10$	50.2(±1.32)	49.9(±1.39)	33.0(±1.27)	51.5(±1.34)	44.2(±1.31)	38.9(±1.32)	58.3(±1.34)
$K = 50$	50.8(±1.31)	50.8(±1.39)	43.6(±1.34)	**53.3**(±1.33)	48.4(±1.32)	44.6(±1.34)	61.1(±1.33)
$K = 100$	50.8(±1.31)	50.4(±1.39)	50.7(±1.34)	50.6(±1.33)	50.0(±1.31)	48.7(±1.35)	**61.5**(±1.32)
$K = 250$	51.7(±1.31)	50.6(±1.39)	59.5(±1.31)	51.7(±1.33)	54.3(±1.31)	55.7(±1.33)	57.6(±1.35)
$K = 500$	53.4(±1.31)	50.2(±1.39)	63.2(±1.28)	52.2(±1.33)	55.9(±1.31)	58.0(±1.32)	52.0(±1.36)
$K = 750$	54.6(±1.31)	50.0(±1.39)	**63.9**(±1.28)	51.1(±1.33)	56.8(±1.31)	60.1(±1.30)	48.9(±1.36)
$K = 1000$	55.7(±1.30)	49.6(±1.39)	63.4(±1.28)	50.9(±1.33)	57.1(±1.31)	60.5(±1.31)	44.0(±1.36)
$K = 2000$	57.3(±1.30)	49.8(±1.39)	59.0(±1.32)	49.2(±1.34)	56.7(±1.30)	60.2(±1.31)	38.7(±1.33)
$K = \infty$	10.6(±0.83)	45.5(±1.38)	21.9(±1.13)	39.1(±1.31)	25.3(±1.15)	23.9(±1.15)	19.5(±1.08)
$AP_{\text{SIM-RND}}$	**65.9**(±1.25)	**51.6**(±1.39)	63.5(±1.29)	50.4(±1.34)	**59.8**(±1.30)	**62.0**(±1.30)	55.0(±1.36)

ble 7.5 confirms that using the MAST play-out strategy with $\epsilon_{\text{MAST}} = 0.4$ is better than using a random play-out strategy, except for Quad and Sheep and Wolf.

A similar comparison can be performed for the agents that use $K = 0$ and $K = 250$ in Table 7.6. These two agents correspond to the agents $\text{P}_{\text{UCT-MAST}}$ and $\text{P}_{\text{GRAVE-MAST}}$ analyzed in Table 5.5, respectively ($K = 0$ means deactivating the GRAVE strategy, using only UCT). The only difference is that all agents tested in Table 7.6 have the C constant set to 0.2, while the C constant for $\text{P}_{\text{UCT-MAST}}$ in Table 5.5 is set to 0.7. For most of the games that are present in both tables, results reported in Table 7.6 seem to confirm the ones reported in Table 5.5, showing that the GRAVE selection strategy with $K = 250$ has at least an equal performance to the UCT selection strategy when combined with MAST. An exception are the games of Knightthrough and Quad, for which results in the two tables seem to differ. For Knightthrough, Table 7.6 suggests that, when combined with MAST, UCT (i.e. $K = 0$) is better than GRAVE with $K = 250$, but in Table 5.5 $\text{P}_{\text{GRAVE-MAST}}$ is shown to be significantly better than $\text{P}_{\text{UCT-MAST}}$. For Quad, instead, Table 7.6 suggests that there is not much difference between combining with MAST the UCT selection strategy or the GRAVE selection strategy with $K = 250$, but Table 5.5 shows $\text{P}_{\text{GRAVE-MAST}}$ to perform significantly worse than $\text{P}_{\text{UCT-MAST}}$. This difference is caused by the fact that the agent that uses the UCT selection strategy have a different value for the C constant in the two tables. Results suggest that, for Knightthrough, $C = 0.7$ is a sub-optimal value for the agent that uses UCT and MAST, therefore the agent that uses GRAVE performs better against it than against the UCT-MAST agent with $C = 0.2$. Conversely, $C = 0.2$ is sub-optimal for the UCT-MAST agent in Quad, causing the performance of this agent to worsen against the agent that uses GRAVE-MAST.

Table 7.6 also suggest that, while being among the best values of K for most of the games, the default value of $K = 250$ is actually sub-optimal for the game of Knightthrough. This would contribute to explain why many instances of the on-line tuning agents in Tables 6.4, 6.5 and 6.6 show to benefit the agent for this game more than for most of the other games. On-line parameter tuning is able to find better values for the parameter than the sub-optimal one. For further confirmation, the AP_{NTBEA} agent instance that tunes four parameters has been matched on Knightthrough for 1 000 game runs against the non-tuning instance of the AP agent that uses the value of $K = 10$ (i.e. the one that in Table 7.6 shows the best performance for the game). Moreover, 500 more game runs have been performed matching on Knightthrough the AP_{NTBEA} agent against the AP agent that uses the default value of $K = 250$, to obtain a total of 1 000 game runs. In this case, the win percentage of AP_{NTBEA} decreases from $66.6(\pm 2.92)$ when the opponent's K is set to a sub-optimal value, to $59.6(\pm 3.04)$, when the opponent's K is set to the best among the tested values. The significant difference in win percentage shows that it is beneficial to tune parameters on-line when the parameters might otherwise be set to a sub-optimal fixed value. However, the fact that the on-line parameter tuning agent is still performing better than the non-tuning agent could indicate that there are other factors at play. For example, for Knightthrough optimal parameter values might depend on the phase of the game. On-line parameter tuning might be able to find the best parameter values for each game phase.

Table 7.7: Parameter randomization against parameter tuning.

Game	**$AP_{SIM\text{-}RND}$**		
	2 param.	**4 param.**	**6 param.**
3D Tic Tac Toe	47.1(±4.01)	52.1(±4.07)	51.7(±4.16)
Breakthrough	39.8(±4.29)	8.8(±2.49)	12.6(±2.91)
Knightthrough	40.8(±4.31)	10.6(±2.70)	15.6(±3.18)
Chinook	50.8(±4.08)	19.7(±3.24)	36.9(±4.11)
Chin.Checkers3P	44.6(±4.34)	47.2(±4.36)	56.3(±4.33)
Checkers	49.7(±4.10)	25.9(±3.63)	62.5(±4.04)
Connect Five	48.1(±3.15)	66.1(±3.32)	66.7(±3.53)
Quad	55.2(±4.17)	65.8(±3.99)	93.0(±2.06)
Sheep and Wolf	48.8(±4.39)	52.2(±4.38)	51.4(±4.39)
TTCC4 2P	50.3(±4.22)	22.7(±3.61)	34.5(±4.12)
TTCC4 3P	50.9(±4.27)	53.9(±4.26)	59.1(±4.22)
Connect Four	49.1(±4.21)	59.7(±4.19)	67.6(±3.92)
Pentago	47.7(±4.15)	54.7(±4.04)	57.5(±4.12)
Reversi	48.1(±4.31)	42.1(±4.26)	57.4(±4.27)
Avg. Win%	47.9(±1.11)	41.5(±1.11)	51.6(±1.14)

Quad is another example that shows that the benefit of on-line parameter tuning does not only depend on having otherwise sub-optimal fixed values. Looking at Table 7.6, also for this game $K = 250$ seems sub-optimal, but in Tables 6.5 and 6.6 only the agent using the LSI allocation strategy has a particularly good performance on it. As opposed to all other strategies, which try from the start to exploit good value combinations for the parameters, LSI in the initial phase is evaluating all parameter values uniformly at random. This suggests that the way the samples are allocated to evaluate the parameter combinations is also playing a role in the success of on-line parameter tuning.

7.3.4 Randomization per Simulation vs Parameter Tuning

This series of experiments compares the agent that uses parameter randomization per simulation, $AP_{SIM\text{-}RND}$ with the one that tunes parameter values on-line with the NTBEA allocation strategy, AP_{NTBEA}. Table 7.7 shows the results obtained by matching the two agents against each other for two (K and *ref*), four (K, *ref*, C and ϵ_{MAST}) and six (K, *ref*, C, ϵ_{MAST}, VO and T) randomized/tuned parameters. Regarding the overall performance, for two and four parameters on-line tuning seems to perform better than randomization, while when the tuned parameters increase to six, randomization performs in general as well as on-line tuning. This is probably because with six parameters the number of possible value combinations becomes too high. With short time settings the agent does not have sufficient time to converge to good combinations, therefore evaluating combinations almost randomly. Among the tested ones, four seems to be the most interesting number of parameters to compare

randomization and on-line tuning against the default values. With two and six parameters the performance of the two agents is close for many of the games, while with four there is a clearer distinction between games for which tuning performs best and games for which randomization performs best.

Looking at specific games, interesting results are the ones for Knightthrough and Breakthrough. For these two games, for each number of considered parameters, on-line tuning seems more effective than randomization. In addition, for Chinook and TTCC4 with 2 players on-line tuning performs much better than randomization when the number of tuned parameters is four or six. On the contrary, in Quad randomization achieves a much higher performance than on-line tuning for all number of parameters. Moreover, in Connect Five and Connect Four, parameter randomization shows a better performance than on-line tuning for four and six parameters, and in Checkers its performance drops when going from two to four parameters, but becomes better than on-line tuning with six parameters. In general, it seems that on-line parameter tuning performs better than parameter randomization on games that present only narrow win paths and many losing paths, like Knightthrough and Breakthrough. On such games, if the search is too diversifies by randomized parameter values many losing paths will be encountered, making it more difficulties to focus on winning paths. Parameter randomization, instead, seems to work better on games with more winning paths. For example, in Quad, Connect Four and Connect Five the aim of the player is to place the pieces to form a certain shape (i.e. squares or lines), and there are usually many ways of doing so.

7.3.5 Comparison of Default Parameter Values, Parameter Randomization and On-line Parameter Tuning

These series of experiments compare parameter randomization and parameter tuning with the fixed default values. First of all, the agent that randomizes parameter values, $\text{AP}_{\text{SIM-RND}}$, and the one that tunes them on-line, AP_{NTBEA}, are matched directly against the agent that uses the fixed default parameter values, AP. Subsequently, all these three agents are compared by matching them against two different types of opponents: an agent that uses fixed sub-optimal values for the parameters and a successful GGP agent, CadiaPlayer. The agents consider K and *ref* when randomizing/tuning two parameters, K, *ref*, C and ϵ_{MAST} when randomizing/tuning four parameters and K, *ref*, C, ϵ_{MAST}, VO and T when randomizing/tuning six parameters.

Results obtained by matching $\text{AP}_{\text{SIM-RND}}$ and AP_{NTBEA} against AP are shown in Table 7.8. Results are reported for two, four and six randomized/tuned parameters. These results are in line to the results presented in Subsection 7.3.4. When matched against an opponent that uses generally good fixed default values, the difference in the overall performance between $\text{AP}_{\text{SIM-RND}}$ and AP_{NTBEA} is similar to the difference in performance that was observed when the two agents were matched against each other directly. Moreover, for two parameters both agents seem to have at least the same performance of AP in many of the tested games. For four parameters the performance of AP_{NTBEA} is still quite close to the one of AP, while the performance of $\text{AP}_{\text{SIM-RND}}$ drops for most of the games. For six parameters

Table 7.8: Win percentage of $AP_{SIM\text{-}RND}$ and AP_{NTBEA} against the agent with default parameter values, AP.

Game	2 parameters		4 parameters		6 parameters	
	$AP_{SIM\text{-}RND}$	AP_{NTBEA}	$AP_{SIM\text{-}RND}$	AP_{NTBEA}	$AP_{SIM\text{-}RND}$	AP_{NTBEA}
3D Tic Tac Toe	**46.4**(±4.12)	46.0(±4.11)	39.4(±4.00)	**39.5**(±4.08)	**40.6**(±4.01)	38.6(±4.05)
Breakthrough	40.4(±4.31)	**48.6**(±4.39)	11.0(±2.75)	**55.2**(±4.36)	9.4(±2.56)	**31.6**(±4.08)
Knightthrough	44.2(±4.36)	**46.8**(±4.38)	20.6(±3.55)	**68.2**(±4.09)	19.8(±3.50)	**50.2**(±4.39)
Chinook	61.8(±3.99)	**63.5**(±3.95)	23.6(±3.61)	**51.0**(±4.05)	19.8(±3.34)	**31.6**(±3.94)
Chin.Checkers3P	45.6(±4.35)	**51.0**(±4.37)	34.7(±4.16)	**42.7**(±4.32)	**33.7**(±4.13)	28.0(±3.92)
Checkers	**48.8**(±4.08)	47.6(±4.13)	17.9(±3.17)	**40.4**(±4.06)	20.4(±3.29)	**20.9**(±3.42)
Connect Five	43.9(±3.12)	**45.7**(±3.05)	**36.5**(±3.22)	28.6(±3.07)	**45.7**(±3.21)	33.2(±3.26)
Quad	**65.0**(±3.93)	60.1(±4.05)	**72.8**(±3.71)	51.9(±4.21)	**72.4**(±3.67)	17.2(±3.13)
Sheep and Wolf	52.0(±4.38)	**52.2**(±4.38)	**49.8**(±4.39)	44.8(±4.36)	**50.0**(±4.39)	46.2(±4.37)
TTCC4 2P	49.5(±4.27)	**51.5**(±4.19)	20.6(±3.47)	**49.7**(±4.20)	19.0(±3.40)	**33.6**(±4.03)
TTCC4 3P	**48.7**(±4.26)	48.4(±4.26)	**46.1**(±4.28)	43.1(±4.18)	**40.8**(±4.23)	39.4(±4.15)
Connect Four	50.9(±4.17)	**55.6**(±4.18)	**55.4**(±4.13)	46.8(±4.20)	**48.0**(±4.23)	30.6(±3.89)
Pentago	53.7(±4.19)	**55.3**(±4.21)	**50.2**(±4.22)	43.5(±4.15)	**42.6**(±4.13)	42.1(±4.18)
Reversi	42.8(±4.29)	**46.9**(±4.33)	31.0(±3.99)	**45.1**(±4.33)	28.7(±3.92)	**33.5**(±4.07)
Avg. Win%	49.6(±1.12)	**51.4**(±1.12)	36.4(±1.08)	**46.5**(±1.12)	**35.1**(±1.07)	34.0(±1.07)

Table 7.9: Win percentage of AP, $AP_{SIM\text{-}RND}$ and AP_{NTBEA} against the agent with sub-optimal fixed parameter values.

Game	**2 parameters**		
	AP	$\mathbf{AP_{SIM\text{-}RND}}$	$\mathbf{AP_{NTBEA}}$
3D Tic Tac Toe	**94.9**(±1.90)	92.4(±2.22)	93.4(±2.03)
Breakthrough	**97.8**(±1.29)	95.6(±1.80)	97.0(±1.50)
Knightthrough	96.6(±1.59)	92.4(±2.33)	**96.8**(±1.54)
Chinook	73.1(±3.55)	**83.5**(±3.07)	80.9(±3.21)
Chin.Checkers3P	67.9(±4.08)	68.5(±4.06)	**72.0**(±3.92)
Checkers	94.8(±1.80)	95.3(±1.76)	**95.7**(±1.68)
Connect Five	**94.7**(±1.54)	92.8(±1.66)	94.5(±1.40)
Quad	91.0(±2.40)	**96.8**(±1.33)	95.9(±1.64)
Sheep and Wolf	**58.4**(±4.32)	56.6(±4.35)	57.0(±4.34)
TTCC4 2P	90.0(±2.60)	92.0(±2.36)	**93.5**(±2.15)
TTCC4 3P	**68.4**(±3.97)	65.0(±4.08)	67.7(±3.96)
Connect Four	93.1(±2.09)	**95.7**(±1.70)	94.2(±1.94)
Pentago	84.1(±3.15)	**89.4**(±2.64)	87.7(±2.83)
Reversi	**85.3**(±3.05)	81.2(±3.35)	83.4(±3.23)
Avg. Win%	85.0(±0.81)	85.5(±0.80)	**86.4**(±0.78)

Game	**4 parameters**		
	AP	$\mathbf{AP_{SIM\text{-}RND}}$	$\mathbf{AP_{NTBEA}}$
3D Tic Tac Toe	**87.1**(±2.67)	80.5(±3.21)	85.3(±2.86)
Breakthrough	96.4(±1.63)	73.2(±3.89)	**97.8**(±1.29)
Knightthrough	93.0(±2.24)	63.0(±4.24)	**99.0**(±0.87)
Chinook	83.3(±3.08)	63.8(±4.06)	**86.0**(±2.85)
Chin.Checkers3P	**79.6**(±3.52)	70.8(±3.97)	70.6(±3.98)
Checkers	**88.4**(±2.55)	67.4(±3.85)	84.1(±2.95)
Connect Five	**85.5**(±2.55)	83.9(±2.77)	67.0(±3.63)
Quad	79.0(±3.52)	**91.3**(±2.42)	81.8(±3.29)
Sheep and Wolf	**65.4**(±4.17)	**65.4**(±4.17)	62.2(±4.25)
TTCC4 2P	89.4(±2.67)	71.3(±3.95)	**89.5**(±2.68)
TTCC4 3P	69.7(±3.95)	**70.0**(±3.95)	67.1(±4.02)
Connect Four	90.8(±2.40)	**94.3**(±1.99)	87.8(±2.79)
Pentago	**87.7**(±2.83)	85.6(±3.02)	81.6(±3.32)
Reversi	**83.8**(±3.20)	67.8(±4.04)	69.5(±4.00)
Avg. Win%	**84.2**(±0.83)	74.9(±0.99)	80.7(±0.89)

Game	**6 parameters**		
	AP	$\mathbf{AP_{SIM\text{-}RND}}$	$\mathbf{AP_{NTBEA}}$
3D Tic Tac Toe	**98.8**(±0.91)	97.3(±1.38)	96.4(±1.61)
Breakthrough	96.8(±1.54)	83.2(±3.28)	**97.0**(±1.50)
Knightthrough	95.2(±1.88)	70.2(±4.01)	**95.6**(±1.80)
Chinook	**93.5**(±2.01)	86.6(±2.82)	89.1(±2.51)
Chin.Checkers3P	**89.7**(±2.66)	85.3(±3.09)	76.4(±3.71)
Checkers	**99.3**(±0.70)	98.8(±0.83)	96.5(±1.39)
Connect Five	96.1(±1.36)	**97.2**(±1.37)	90.1(±2.32)
Quad	98.3(±1.08)	**99.4**(±0.68)	92.6(±2.21)
Sheep and Wolf	**77.0**(±3.69)	70.8(±3.99)	67.4(±4.11)
TTCC4 2P	**98.2**(±1.17)	91.8(±2.41)	94.2(±2.05)
TTCC4 3P	**85.0**(±3.05)	81.7(±3.34)	79.4(±3.47)
Connect Four	97.3(±1.35)	**98.1**(±1.08)	95.1(±1.84)
Pentago	**93.4**(±2.11)	**93.4**(±2.07)	90.3(±2.56)
Reversi	**95.4**(±1.80)	93.8(±2.10)	92.9(±2.19)
Avg. Win%	**93.8**(±0.55)	89.1(±0.72)	89.5(±0.70)

Table 7.10: Win percentage of AP, $AP_{SIM\text{-}RND}$ and AP_{NTBEA} (1s start- and play-clock) against CadiaPlayer (10s start- and play-clock).

Game	AP	2 parameters		4 parameters		6 parameters	
		$AP_{SIM\text{-}RND}$	AP_{NTBEA}	$AP_{SIM\text{-}RND}$	AP_{NTBEA}	$AP_{SIM\text{-}RND}$	AP_{NTBEA}
3D Tic Tac Toe	92.1(±2.36)	**92.3**(±2.26)	91.9(±2.34)	**91.4**(±2.41)	90.4(±2.55)	**91.9**(±2.35)	86.7(±2.89)
Breakthrough	63.2(±4.23)	50.6(±4.39)	**61.8**(±4.26)	23.2(±3.70)	**68.0**(±4.09)	19.4(±3.47)	**45.8**(±4.37)
Knightthrough	50.8(±4.39)	35.8(±4.21)	**52.2**(±4.38)	19.6(±3.48)	**74.8**(±3.81)	16.0(±3.22)	**45.0**(±4.37)
Chinook	82.8(±3.22)	86.6(±2.92)	**88.0**(±2.74)	54.1(±4.22)	**81.3**(±3.28)	55.1(±4.24)	**63.4**(±4.10)
Checkers	90.6(±2.32)	86.5(±2.72)	**91.2**(±2.28)	63.2(±3.97)	**87.6**(±2.71)	**65.8**(±3.92)	52.6(±4.12)
Connect Five	70.4(±3.18)	66.8(±3.33)	**68.2**(±3.29)	**61.2**(±3.73)	45.5(±3.78)	**70.4**(±3.54)	51.9(±3.95)
Quad	98.8(±0.96)	**99.6**(±0.55)	99.2(±0.78)	98.8(±0.96)	**99.4**(±0.68)	**99.4**(±0.68)	93.0(±2.24)
Sheep and Wolf	56.8(±4.35)	56.8(±4.35)	**60.4**(±4.29)	**51.6**(±4.38)	**51.6**(±4.38)	**55.6**(±4.36)	50.0(±4.39)
Connect Four	68.2(±3.90)	65.1(±4.04)	**69.7**(±3.92)	**68.5**(±3.98)	63.2(±4.06)	**65.4**(±4.04)	48.0(±4.24)
Pentago	73.0(±3.80)	75.0(±3.62)	**78.1**(±3.52)	69.7(±3.95)	**71.3**(±3.80)	**64.7**(±4.11)	62.6(±4.10)
Avg. Win%	74.7(±1.16)	71.5(±1.21)	**76.1**(±1.14)	60.1(±1.32)	**73.3**(±1.19)	**60.4**(±1.32)	59.9(±1.32)

both tuning and randomizing parameter values does not seem to provide any benefit in most of the games. Once again, Quad is an interesting game, because it seems to significantly benefit from parameter randomization for any tested number of randomized parameters. What is interesting to observe is that among the on-line tuning strategies evaluated in Chapter 6 there was one, LSI, which showed a similarly good performance on Quad. A characteristic of LSI is that during its first phase of *generation*, the allocation strategy is evaluating random parameter combinations, in a similar way to the SIM-RND randomization strategy. This can be seen as a confirmation that the search for the game of Quad particularly benefits from randomizing its control parameters.

The results obtained by matching the AP, $AP_{SIM\text{-}RND}$ and AP_{NTBEA} agents against the agent that uses sub-optimal fixed values for the parameters are presented in Table 7.9. These agents are compared for two, four and six randomized/tuned parameters. Note that for the sub-optimal agent only the parameters that are being randomized/tuned by the opponent are set to sub-optimal values, other parameters are set to their default values. Once again, results show that for two and six parameters $AP_{SIM\text{-}RND}$ and AP_{NTBEA} are close in performance, while for four parameters AP_{NTBEA} performs better than $AP_{SIM\text{-}RND}$. In general, all agents have a much better performance than the agent with sub-optimal values. This supports the claim that parameter values have a strong influence on the search and that it is worth investigating which are the best values for each game. Moreover, in a situation where the optimal parameter values for a specific game are not known in advance, tuning or randomizing them seems to be a valid approach, rather than setting an arbitrary combination that might be sub-optimal.

Results of the final series of experiments are shown in Table 7.10. This table reports the results obtained by matching AP, $AP_{SIM\text{-}RND}$ and AP_{NTBEA} against CadiaPlayer. All the agents tested so far implement the same MCTS strategy with the same enhancements. The purpose of this series of experiments is to verify how the agents perform against an opponent with a different implementation of MCTS. For most of the games, independently of the number of considered parameters, all agents show a better performance than CadiaPlayer, with AP_{NTBEA} that tunes two parameters being the best. It is confirmed that on-line parameter tuning is successful in Breakthrough and Knightthrough for two tuned parameters and in particular for four, while parameter randomization performs the worst on these two games for four and six parameters.

Next, for two parameters, in all previous series of experiments the performance of $AP_{SIM\text{-}RND}$ was close to the one of AP_{NTBEA}. However, against CadiaPlayer the difference in performance is higher. This means that the relative performance of these two agents does not only depend on the number of parameters that they are considering, but also by the type of opponent. From previous experiments, tuning only two parameters might have seemed not very interesting, because it was only slightly improving the performance over randomization. However, against CadiaPlayer on-line tuning is shown to be better than just randomizing the parameters. Therefore, for the agent to be more robust against different types of opponents, two parameters are still worth tuning.

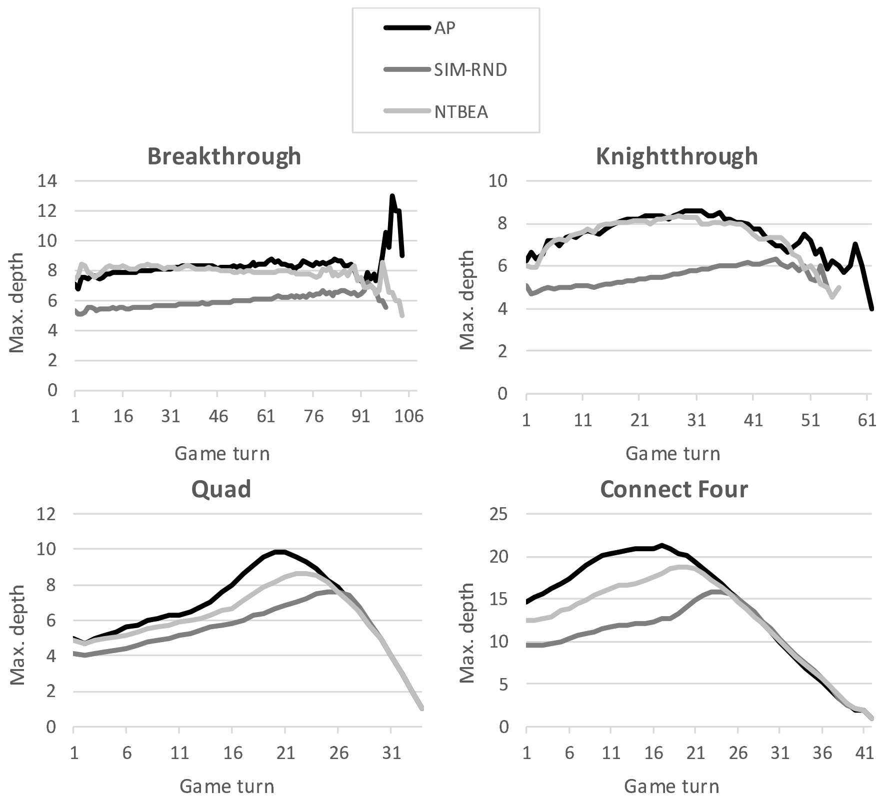


Figure 7.3: Maximum MCTS tree depth over game turns of AP, $AP_{SIM\text{-}RND}$ and AP_{NTBEA}.

The biggest gap in the performance of $AP_{SIM\text{-}RND}$ with respect to AP_{NTBEA} is visible for four parameters. Moreover, by going from two to four parameters the performance of $AP_{SIM\text{-}RND}$ drops much more than the one of AP_{NTBEA} (more than 9 points for $AP_{SIM\text{-}RND}$ and less than 3 for AP_{NTBEA}). This suggest that, for different opponents, on-line tuning might be more robust than parameter randomization.

7.3.6 Search Tree Analysis

To get more insight about the behavior of MCTS when parameters are tuned or randomized, for some of the games considered in previous experiments, statistics about the search trees built by the agent instances are collected for each turn. Moreover, to give an example of the structure of the trees built by the different agent instances, sample trees are plotted for each game.[2] The considered games are Breakthrough, Knightthrough, Quad and Connect Four, and have been selected among the games for which either on-line parameter tuning or parameter randomization show to perform best in many of the performed experiments. The considered agents are AP,

[2]Tree plots are inspired by the tree visualizations used by Wimmenauer (2019).

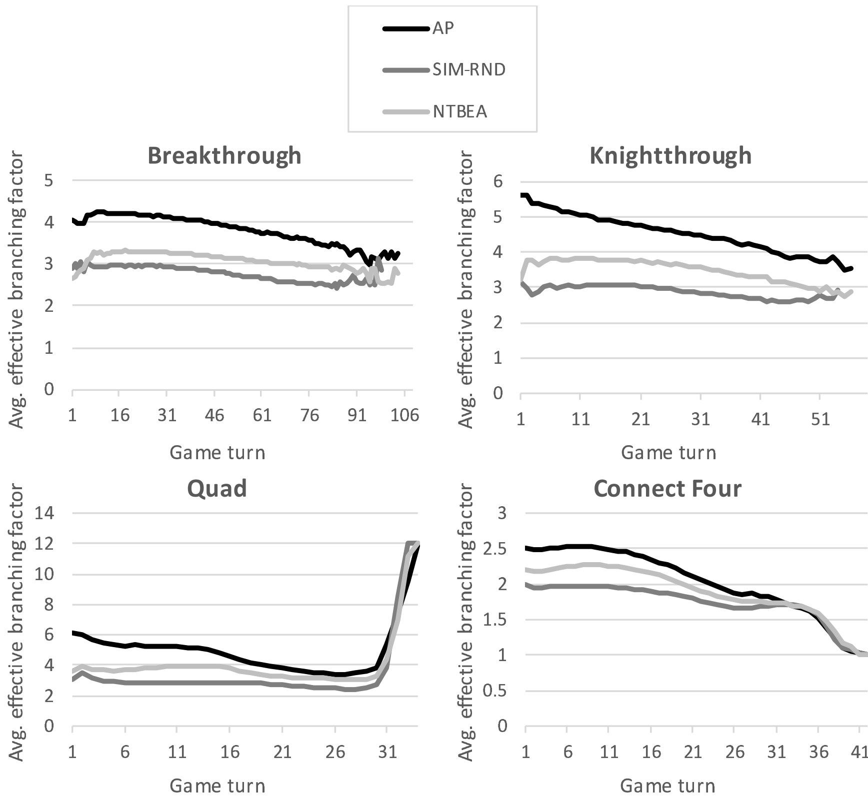


Figure 7.4: Average effective branching factor over game turns of AP, $AP_{\text{SIM-RND}}$ and AP_{NTBEA}.

$AP_{\text{SIM-RND}}$ and AP_{NTBEA}, with the last two agents randomizing/tuning four parameters, C, ϵ_{MAST}, K and ref.

For the considered games, Figure 7.3 reports the maximum depth reached by the tree built by MCTS (considering the current root as having depth 0) during each turn of the game. Moreover, Figure 7.4 reports the average effective branching factor of the tree built by MCTS for each game turn. Note that, while the branching factor is computed considering all the actions that are legal in a node, the effective branching factor is computed considering only the actions that have actually been visited by the search algorithm. Each point in the presented plots has been computed averaging the considered value (i.e. maximum depth or average effective branching factor) over 500 runs for each game. Note that not all runs reach the last game turns, thus data points towards the end of the series are based on fewer samples.

As can be seen, for all games AP is the agent that in each turn builds a tree with the highest maximum depth and highest average effective branching factor. $AP_{\text{SIM-RND}}$ is the one that, in most turns, builds a tree for which these statistics are the lowest, while for the trees built by AP_{NTBEA} these statistics are in-between the ones of AP and $AP_{\text{SIM-RND}}$. Note that maximum depth and average effective

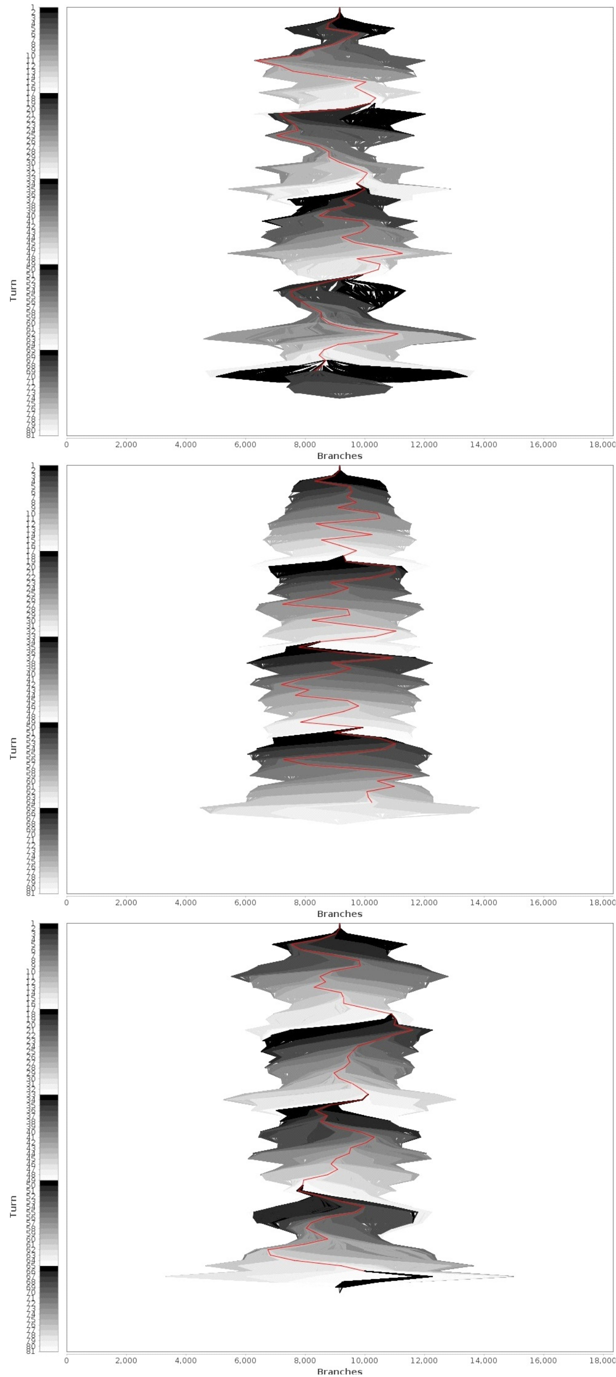


Figure 7.5: Trees built for Breakthrough by the AP, AP$_{\text{SIM-RND}}$ and AP$_{\text{NTBEA}}$ agents, respectively.

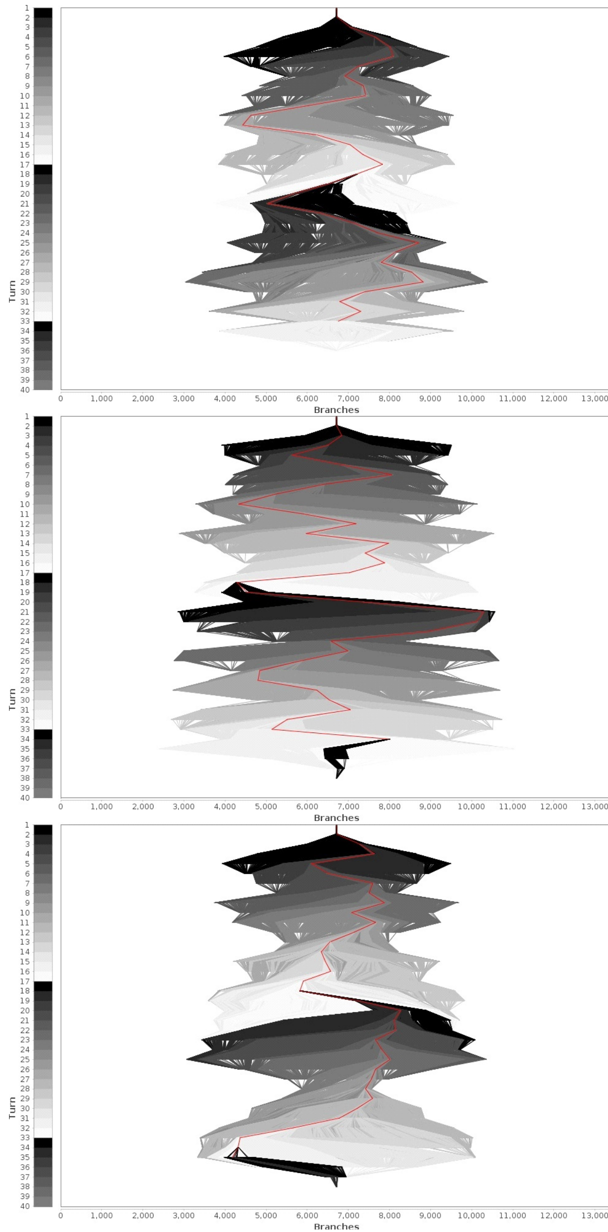


Figure 7.6: Trees built for Knightthrough by the AP, $AP_{\text{SIM-RND}}$ and AP_{NTBEA} agents, respectively.

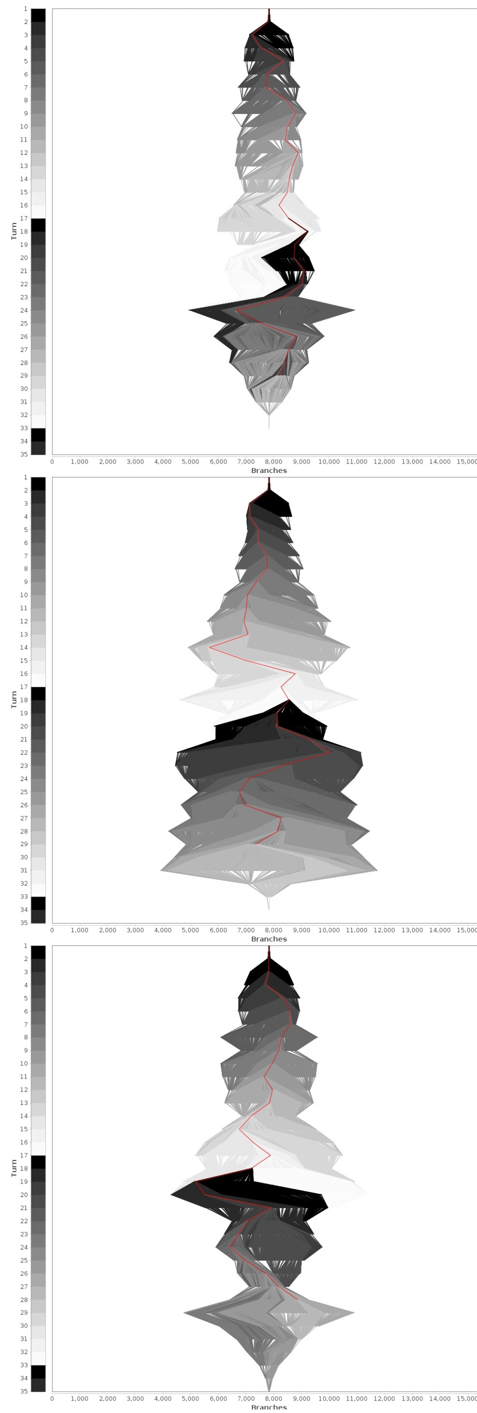


Figure 7.7: Trees built for Quad by the AP, $\text{AP}_{\text{SIM-RND}}$ and AP_{NTBEA} agents, respectively.

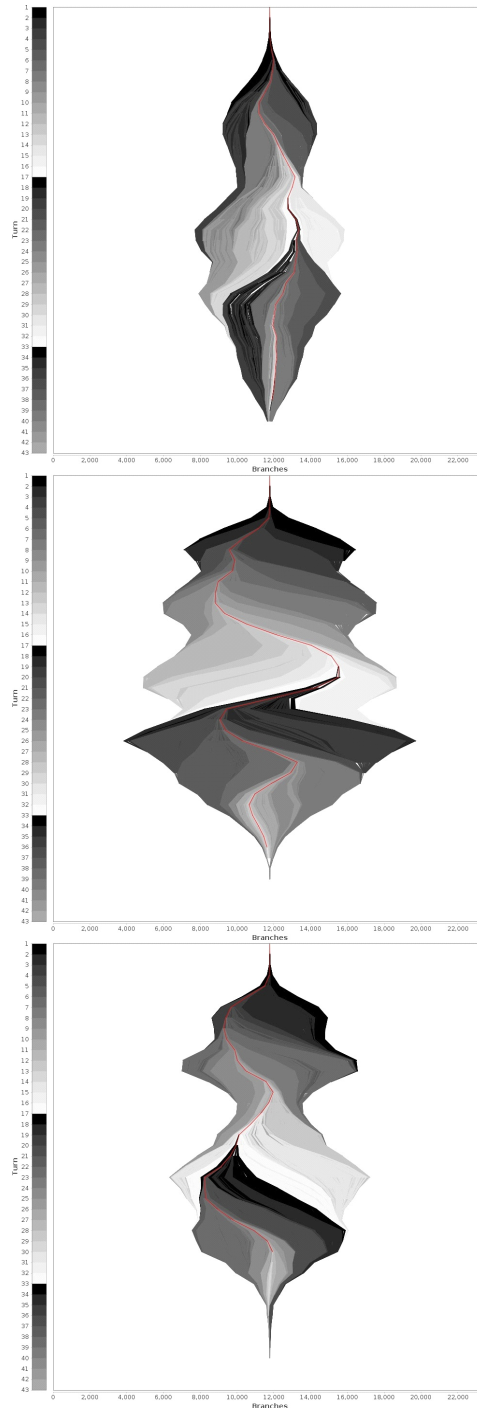


Figure 7.8: Trees built for Connect Four by the AP, $AP_{\text{SIM-RND}}$ and AP_{NTBEA} agents, respectively.

branching factor only give information about how the tree is built by each of the strategies, but not on how nodes are actually visited during the search (i.e. how many times and in which order).

The statistics for the depth seem to confirm that the more randomization is introduced by changing the parameters the less the agent tends to exploit promising paths. With less or no randomization the agent focuses more on a few paths in the tree in order to visit them deeper. Results for the effective branching factor show that the more randomization is introduced the more the agent tends to explore, on average, fewer actions in each node. This might seem counterintuitive at first, because the randomization introduced by changing parameter values on-line would be expected to diversify the search and explore more branches of the tree. These results can be explained by considering that the effective branching factor of a search tree depends on its actual branching factor, and the tree visited by AP likely has a higher average branching factor than the one of $\text{AP}_{\text{SIM-RND}}$ and AP_{NTBEA}. The AP agent is more likely to focus the search on realistic lines of play, which usually consist of states with higher mobility, and therefore higher number of legal actions. $\text{AP}_{\text{SIM-RND}}$ and AP_{NTBEA}, instead, try also to explore less realistic moves that might lead to parts of the tree where the branching factor is lower. For example, in games like Knightthrough and Breakthrough AP tends to explore less the actions that lead to the opponent capturing the player's pieces. On the contrary, with a more diversified search $\text{AP}_{\text{SIM-RND}}$ and AP_{NTBEA} explore such actions more often. This means that the average branching factor of the search tree visited by $\text{AP}_{\text{SIM-RND}}$ and AP_{NTBEA} is overall lower than the one of the tree visited by AP.

More insights about the characteristics of the trees built by the three considered agents are visible in Figures 7.5, 7.6, 7.7 and 7.8. These figures show an example of the structure of the tree built by each considered agent for the games of Breakthrough, Knightthrough, Quad and Connect Four, respectively. In each plot, the x-axis reports the branches of the tree, while the y-axis reports the game turns, each corresponding to a different shade of gray. In the figure, edges added during a certain game turn are plotted with the shade of gray corresponding to the turn in which they were added. In each figure, the trees have been built, in order, by the agents AP, $\text{AP}_{\text{SIM-RND}}$ and AP_{NTBEA}. Looking at the trees it is clear that each agent builds the search tree in a different way. AP seems the one that builds the more focused and deeper tree in each game turn. On the contrary, $\text{AP}_{\text{SIM-RND}}$ seems to be the agent that builds more shallow trees, visiting many different paths in each game turn, confirming the intuition that randomizing parameter values diversifies the search. The behavior of AP_{NTBEA}, instead, seems to be in-between. It seems that in a few game turns the search is more focused and deep, while in others it is more shallow. This suggest that AP_{NTBEA}, by tuning parameters, might also be able to detect when a more diversified search is better than a more focused one.

7.3.7 Parameter Randomization in Real-time Settings

The series of experiments presented in this section evaluate parameter randomization in the real-time domain of the GVG-AI project, applying it to the MCTS-based MaastCTS2 agent. More precisely, parameter randomization per simulation is

Table 7.11: Win percentage of MaastCTS2 with fixed parameter values (MP), with sub-optimal parameter values ($MP_{SUB\text{-}OPT}$), with randomization per simulation ($MP_{SIM\text{-}RND}$) and tuned on-line with the NTBEA strategy (MP_{NTBEA}), with game tick set to $40ms$.

Game	MP	$MP_{SUB\text{-}OPT}$	2 parameters		3 parameters		5 parameters	
			$MP_{SIM\text{-}RND}$	MP_{NTBEA}	$MP_{SIM\text{-}RND}$	MP_{NTBEA}	$MP_{SIM\text{-}RND}$	MP_{NTBEA}
Aliens	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)
Bait	31.8(±4.09)	25.8(±3.84)	32.4(±4.11)	31.2(±4.07)	29.6(±4.01)	32.0(±4.09)	30.4(±4.04)	31.6(±4.08)
Butterflies	98.6(±1.03)	99.0(±0.87)	99.6(±0.55)	98.8(±0.96)	98.8(±0.96)	99.8(±0.39)	99.2(±0.78)	100.0(±0.00)
Camel Race	44.4(±4.36)	33.8(±4.15)	42.0(±4.33)	40.6(±4.31)	41.0(±4.32)	41.6(±4.32)	41.0(±4.32)	42.4(±4.34)
Chase	28.0(±3.94)	28.2(±3.95)	27.6(±3.92)	24.4(±3.77)	25.0(±3.80)	28.2(±3.95)	30.4(±4.04)	26.8(±3.89)
Chopper	99.8(±0.39)	99.2(±0.78)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	99.8(±0.39)	99.6(±0.55)
Crossfire	31.8(±4.09)	27.6(±3.92)	28.6(±3.96)	29.8(±4.01)	27.0(±3.90)	29.4(±4.00)	27.4(±3.91)	28.4(±3.96)
Dig Dug	1.6(±1.10)	0.8(±0.78)	1.2(±0.96)	0.8(±0.78)	1.2(±0.96)	1.8(±1.17)	1.2(±0.96)	1.2(±0.96)
Escape	93.4(±2.18)	93.2(±2.21)	92.2(±2.35)	91.0(±2.51)	93.0(±2.24)	91.0(±2.51)	94.6(±1.98)	92.2(±2.35)
Hungry Birds	100.0(±0.00)	99.8(±0.39)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)
Infection	100.0(±0.00)	99.8(±0.39)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	99.8(±0.39)	100.0(±0.00)
Intersection	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)
Lemmings	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)
Missile Command	96.8(±1.54)	89.6(±2.68)	98.0(±1.23)	97.2(±1.45)	96.0(±1.72)	96.6(±1.59)	96.6(±1.59)	94.6(±1.98)
Modality	25.6(±3.83)	43.4(±4.35)	26.6(±3.88)	25.0(±3.80)	26.6(±3.88)	41.4(±4.32)	24.6(±3.78)	41.0(±4.32)
Plaque Attack	94.8(±1.95)	90.0(±2.63)	95.4(±1.84)	96.6(±1.59)	95.4(±1.84)	94.8(±1.95)	95.0(±1.91)	95.8(±1.76)
Roguelike	4.6(±1.84)	1.8(±1.17)	5.2(±1.95)	4.0(±1.72)	4.0(±1.72)	4.0(±1.72)	4.8(±1.88)	3.2(±1.54)
Sea Quest	58.4(±4.32)	59.6(±4.31)	60.6(±4.29)	53.4(±4.38)	57.2(±4.34)	50.2(±4.39)	53.6(±4.38)	54.6(±4.37)
Survive Zombies	42.4(±4.34)	40.2(±4.30)	43.6(±4.35)	41.4(±4.32)	42.4(±4.34)	42.2(±4.33)	42.8(±4.34)	40.8(±4.31)
Wait For Breakfast	99.0(±0.87)	83.4(±3.26)	98.4(±1.10)	98.4(±1.10)	98.8(±0.96)	99.0(±0.87)	98.4(±1.10)	98.0(±1.23)
Avg. Win%	62.6(±0.95)	60.8(±0.96)	62.6(±0.95)	61.6(±0.95)	61.8(±0.95)	62.6(±0.95)	62.0(±0.95)	62.5(±0.95)

Table 7.12: Win percentage of MaastCTS2 with fixed parameter values (MP), with sub-optimal parameter values ($MP_{SUB\text{-}OPT}$), with randomization per simulation ($MP_{SIM\text{-}RND}$) and tuned on-line with the NTBEA strategy (MP_{NTBEA}), with game tick set to 100*ms*.

Game	MP	$MP_{SUB\text{-}OPT}$	2 parameters		3 parameters		5 parameters	
			$MP_{SIM\text{-}RND}$	MP_{NTBEA}	$MP_{SIM\text{-}RND}$	MP_{NTBEA}	$MP_{SIM\text{-}RND}$	MP_{NTBEA}
Bait	51.8(±4.38)	49.4(±4.39)	37.2(±4.24)	40.8(±4.31)	36.6(±4.23)	34.8(±4.18)	40.4(±4.31)	36.6(±4.23)
Camel Race	95.8(±1.76)	93.8(±2.12)	95.0(±1.91)	92.4(±2.33)	92.4(±2.33)	89.4(±2.70)	92.2(±2.35)	90.8(±2.54)
Chase	56.2(±4.35)	50.0(±4.39)	52.6(±4.38)	52.0(±4.38)	53.2(±4.38)	50.6(±4.39)	50.4(±4.39)	51.6(±4.38)
Crossfire	84.8(±3.15)	80.4(±3.48)	82.2(±3.36)	81.2(±3.43)	83.6(±3.25)	79.8(±3.52)	83.2(±3.28)	81.8(±3.39)
Dig Dug	0.0(±0.00)	0.2(±0.39)	0.2(±0.39)	0.2(±0.39)	0.0(±0.00)	0.2(±0.39)	0.2(±0.39)	0.0(±0.00)
Lemmings	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)
Modality	26.2(±3.86)	25.2(±3.81)	22.8(±3.68)	23.8(±3.74)	25.2(±3.81)	40.6(±4.31)	25.6(±3.83)	40.4(±4.31)
Roguelike	32.6(±4.11)	34.0(±4.16)	31.6(±4.08)	31.8(±4.09)	30.6(±4.04)	36.8(±4.23)	30.6(±4.04)	32.0(±4.09)
Sea Quest	58.6(±4.32)	59.8(±4.30)	57.8(±4.33)	59.8(±4.30)	58.2(±4.33)	61.4(±4.27)	56.0(±4.36)	56.2(±4.35)
Survive Zombies	49.0(±4.39)	49.0(±4.39)	51.4(±4.39)	46.4(±4.38)	44.4(±4.36)	46.0(±4.37)	46.0(±4.37)	45.4(±4.37)
Avg. Win%	45.5(±1.38)	44.2(±1.38)	43.1(±1.37)	42.8(±1.37)	42.4(±1.37)	44.0(±1.38)	42.5(±1.37)	43.5(±1.37)

compared with fixed parameter values, sub-optimal fixed parameter values and on-line parameter tuning with the NTBEA allocation strategy.

Table 6.12 shows the results obtained by testing MaastCTS2 with fixed parameter values (MP), with sub-optimal fixed parameter values ($\text{MP}_{\text{SUB-OPT}}$), with randomization per simulation ($\text{MP}_{\text{SIM-RND}}$) and tuned on-line with the NTBEA strategy (MP_{NTBEA}), with the game tick duration set to $40ms$ (i.e. the default time settings of the GVG-AI competition). The on-line tuning and randomizing agents have been tested for two (C and W), three (C, W and ϵ_{NST}) and five (C, W, ϵ_{NST}, N and L) parameters. Looking at the overall win percentage, all agent instances seem quite close in performance, although the one of $\text{MP}_{\text{SUB-OPT}}$ is slightly lower. As observed for MP_{NTBEA} in Chapter 6, also for $\text{MP}_{\text{SIM-RND}}$ it seems that increasing the number of considered parameters does not influence the performance much. For most of the games, no significant difference in performance is observed among all the agents, even for $\text{MP}_{\text{SUB-OPT}}$, which would be expected to perform badly in many games because of its sub-optimal parameters. One of the reasons for no particular difference in the performance could be, as already mentioned in Subsection 6.4.8, the low number of simulations performed by the agents in each tick. For such games, the performed simulations might not be sufficient for the parameter values, whether fixed, randomized or tuned, to influence the quality of the search. Another reason could be that for these games parameters have only a limited effect on the search and changing their values does not vary the search meaningfully.

More interesting is to compare the performance of $\text{MP}_{\text{SIM-RND}}$ and MP_{NTBEA} on the games for which the performance of $\text{MP}_{\text{SUB-OPT}}$ is significantly different from the one of MP. For example, in Camel Race, Missile Command, Plaque Attack, Roguelike and Wait For Breakfast, $\text{MP}_{\text{SUB-OPT}}$ performs significantly worse than MP. However, for these games both $\text{MP}_{\text{SIM-RND}}$ and MP_{NTBEA} manage to reach the same performance of MP, suggesting that changing parameter values on-line is better than using sub-optimal fixed values. For these games, however, there is no significant difference in performance between $\text{MP}_{\text{SIM-RND}}$ and MP_{NTBEA}, indicating that probably the manually constructed sets of parameter values from which randomization per simulation can select contains mostly reasonable values. If this is the case, there seems to be no advantage in using an informed tuning strategy to find optimal values and exclude bad ones, randomization is already enough to control most of the search with optimal values. Another game worth examining is Modality. Also for this game $\text{MP}_{\text{SUB-OPT}}$ has a significantly different performance with respect to MP. It seems that the parameters of MP, expected to be overall optimal, are actually sub-optimal for Modality, and the values used by $\text{MP}_{\text{SUB-OPT}}$ perform much better. This game highlights the difference between parameter randomization and on-line parameter tuning, showing that there are games where the latter, using a more informed strategy to select parameter values, is able to outperform the former. When three or five parameters are considered, MP_{NTBEA} is able to reach the same performance of $\text{MP}_{\text{SUB-OPT}}$, while the performance of $\text{MP}_{\text{SIM-RND}}$ stays close to the one of the MP agent with the sub-optimal values for the game.

To verify whether more simulations influence the performance, another series of experiments has been performed increasing the tick duration to $100ms$. For the same reasons mentioned in Subsection 6.4.8, when using a tick of $100ms$ only part

of the original set of games are considered. Table 7.12 presents the results of this series of experiments. As can be seen, for many of the games a longer search time substantially increases the win rate of all the agents. Once again, all the agents seem to be overall quite close to each other in performance, although the overall win percentage of $\text{MP}_{\text{SIM-RND}}$ seems to decrease with the increase in the number of randomized parameters. For most of the games there seems to be no significant difference in the performance of all the agents, including the one with sub-optimal parameter values, $\text{MP}_{\text{SUB-OPT}}$. An exception is Bait, for which both $\text{MP}_{\text{SIM-RND}}$ and MP_{NTBEA} perform significantly worse than MP and $\text{MP}_{\text{SUB-OPT}}$, and none of the two seems to outperform the other. For two parameters MP_{NTBEA} seems slightly better than $\text{MP}_{\text{SIM-RND}}$, while for five the situation is reversed. In Camel Race MP_{NTBEA} seems to perform worse than $\text{MP}_{\text{SIM-RND}}$. While $\text{MP}_{\text{SIM-RND}}$ manages to keep its performance close to the one of MP when increasing the number of considered parameters, the one of MP_{NTBEA} significantly decreases with respect to MP. For Modality results are similar to the ones obtained with a tick of $40ms$. Both $\text{MP}_{\text{SIM-RND}}$ and MP_{NTBEA} are at least performing as well as MP, but when the considered parameters are three or five, the win percentage of MP_{NTBEA} increases significantly, confirming that for this game on-line parameter tuning is superior to parameter randomization.

7.4 Chapter Conclusions and Future Research

This chapter evaluated four different strategies that randomize search-control parameters for MCTS in GGP: per game, per turn, per simulation and per state. Moreover, randomization per simulation has been compared with on-line parameter tuning both on the Stanford GGP project and the GVG-AI project, giving more insights on the performance of both approaches.

Results for the Stanford GGP project showed that, when compared to each other, the randomization strategy that performs best is the one that randomizes parameter values before each simulation. Moreover, for single parameters, it has been shown that for some games it is better to randomize the value per simulation rather than keeping it fixed for the whole game. This suggests that MCTS might benefit from diversifying the search not only by using strategies like UCT and MAST that try to balance exploration and exploitation of moves, but also by changing the strategy itself while searching. Randomizing parameter values also allows to explore the search space of the strategies, diversifying the search even more. The fact that the search is further diversified by constantly changing the parameter values while searching for the best ones, might also be one of the reasons behind the success of on-line parameter tuning in some games.

Results on the Stanford GGP project have also shown that the effect of parameter randomization depends on multiple factors. Whether it is beneficial to randomize parameter values per simulation rather than keeping them fixed or rather than tuning them on-line does not only depend on the game. It also depends on which and how many parameters are being randomized and on the type of opponent the agent is facing. Future work could look into devising a mechanism to automatically detect

for each new game if it is worth using parameter randomization, and if so, which parameters are worth randomizing. This mechanism could further be extended to detect on-line if, for a given game, it is better to randomize or tune the parameters.

Results for the GVG-AI games show that parameter randomization performs more or less the same as on-line parameter tuning. This suggest that on-line parameter tuning might be more suitable for domains where a higher number of simulations can be reached, or for domains that are more sensitive to changes in the search-control parameter values. It may be concluded that in a real-time context like GVGP, randomization of parameter values gives a more robust setting for a small number of parameters, especially when a small set of feasible values is selected in advance. For future work it would be interesting to see if increasing time constraints to achieve a few thousands simulations per tick in GVG-AI would make a difference on the performance of parameter randomization an on-line parameter tuning.

Given the results presented in this chapter, it may be concluded that, although not always the best solution for all games, randomization within a given set of values is still beneficial in GGP for games where the fixed parameter settings optimized off-line on a predefined set of games are actually performing poorly. Moreover, parameter randomization might be a valid alternative to on-line parameter tuning when the number of parameters to tune is high and time settings are limited, because the problem of tuning them on-line becomes too hard due to the increased combinatorial complexity.

Chapter 8

Conclusions and Future Research

This thesis investigated how search can be utilized to support Artificial General Intelligence (AGI) in games. General game playing (GGP) was identified as a suitable domain to test search techniques for AGI. In addition, Monte-Carlo Tree Search (MCTS) was presented as a successful search technique for domains like GGP, where no specific domain knowledge is available. This has led to the formulation of the problem statement in Section 1.5, which focuses on how the performance of MCTS in GGP can be improved. Four research questions have also been formulated and they should be addressed before answering the problem statement.

This chapter provides the conclusions of this thesis. Section 8.1 answers the four research questions, while Section 8.2 answers the problem statement. Finally, recommendations for future research are given in Section 8.3.

8.1 Answers to the Research Questions

The four research questions formulated in Chapter 1 concern different aspects of MCTS. More precisely, they deal with (1) speeding up the interpretation of game rules written in a declarative language, (2) evaluating the use of global or local information to enhance the selection strategy of MCTS, (3) on-line tuning of search-control parameters for MCTS, and (4) investigating the effect of search-control parameter randomization in MCTS. The research questions are answered in the following subsections.

8.1.1 Speeding Up Game Rule Interpretation

A possible approach to formally represent game rules in GGP consists in using a declarative language. When rules are expressed in a declarative language, agents usually have to implement a mechanism that interprets them and computes all the elements that are necessary to reason on the game (i.e., game states, legal moves,

etc.). However, this interpretation process is in general slow and might reduce the number of simulations that an agent can perform. This might hinder the performance of MCTS, which instead benefits from the more accurate statistics that can be collected with a higher number of simulations. This has led to the formulation of the first research question.

> **Research question 1**: *How can the process of interpreting on-line the game rules written in a declarative language be sped up?*

To answer this research question, this thesis dealt with the problem of speeding up the rule interpretation process of an agent for the Stanford GGP project. A rule interpreter based on PropNets (Schkufza *et al.*, 2008; Cox *et al.*, 2009; Genesereth and Thielscher, 2014) was investigated, together with four optimizations for its structure. Moreover, its encoding on an FPGA was evaluated.

Results showed that a software implementation of the PropNet performs better than the GGP Base Prover, a custom made interpreter for GDL rules. The software implementation of the PropNet increases the number of game simulations by an average of two orders of magnitude with respect to the GGP Base Prover. Moreover, the speed of the PropNet is further increased by applying four optimizations, which, in order, remove PropNet components with constant truth values, remove PropNet propositions that do not have any special meaning for the game, detect and then remove components that will only assume a constant truth value during the game reasoning process, and remove components that have no output and thus no particular meaning for the game because of that. The use of a cache that memorizes results previously computed by the PropNet is shown to increase the overall speed further. However, its use is recommended only for games with a small search space, like Chinese Checkers with 1 player and Tic Tac Toe, for which it increases the speed already in the first search steps. Furthermore, the use of a PropNet based reasoner enables the MCTS agent to reach a win rate close to 100% against an agent that uses the Prover. Finally, the speed of a PropNet-based reasoner can be further increased by at least one order of magnitude by encoding the PropNet on an FPGA board. It may be concluded that using a PropNet with an optimized structure to represent game rules written in GDL is beneficial for MCTS-based agents, because they are able to perform more simulations in the given time frame and they can profit from hardware acceleration.

8.1.2 Local and Global Information in the Selection Strategy

Previous research has shown that enhancing the MCTS selection strategy by increasing the amount of information used to guide the search can consistently improve the overall performance (Chaslot *et al.*, 2008b; Finnsson and Björnsson, 2010; Nijssen and Winands, 2011; Gelly and Silver, 2011; Cazenave, 2015). The RAVE strategy (Gelly and Silver, 2007) and its generalization, GRAVE (Cazenave, 2015) have been shown to be successful enhancements for the selection phase of MCTS. Both of them bias the selection towards actions that seem to perform generally well in the game. However, GRAVE uses more global information than RAVE to bias action selection in nodes that only have a low number of visits. This strategy has been

shown to perform better than RAVE on some variants of Go and a few other games (Cazenave, 2015), therefore it might be successful in GGP as well. This has led to the formulation of the second research question.

> **Research question 2**: *What is the effect of using locally or globally collected information to enhance the selection strategy for Monte-Carlo Tree Search?*

To answer this research question, this thesis proposed another variant of RAVE, HRAVE, which biases action selection always using global information about the actions. It then compared the performance of RAVE, GRAVE and HRAVE on a set of games from the Stanford GGP project, both combined with a random play-out strategy and with the more informed MAST play-out strategy.

Results show that, when RAVE variants are combined with a random play-out strategy, the performance of GRAVE is, in the worst case, comparable with the one of RAVE, both when using 1s or 10s play-clock. The performance of HRAVE, instead, is more game dependent, sometimes being better than RAVE or GRAVE and sometimes being worse. Moreover, when RAVE variants are combined with the MAST play-out strategy, GRAVE still seems to be overall better than RAVE. However, its advantage is less than when both strategies are combined with random play-outs, and there are a few games where the combination GRAVE-MAST actually performs worse than RAVE-MAST. Additionally, the combination HRAVE-MAST seems to perform slightly less than both RAVE-MAST and GRAVE-MAST.

Over all the experiments, the difference in performance between RAVE, GRAVE and HRAVE is not large. However, the overall win rate of GRAVE is never inferior to the one of RAVE and HRAVE, and it seems the most robust among all the RAVE variants. Therefore, it may be concluded that a strategy that starts biasing action selection with global information and uses more local information the more nodes have been visited is the most suitable to enhance MCTS for GGP. In addition, and advantage of GRAVE is that it can be switched to a pure RAVE or a pure HRAVE strategy by simply modifying one of its parameters. With respect to the other two variants, this makes it more promising to be tuned on-line with the approach presented in Chapter 6.

8.1.3 On-line Search-Control Parameter Tuning

MCTS and its enhancements are usually controlled by multiple parameters that require extensive and time consuming off-line optimization. Moreover, optimal parameter values usually vary across games. In GGP, where the games to be played are not known in advance, off-line optimization cannot tune parameters specifically for each of them. Agents would have to adjust parameter values while playing the game, therefore in an on-line fashion. This has led to the formulation of the third research question.

> **Research question 3**: *How can search-control parameters for Monte-Carlo Tree Search be tuned effectively on-line?*

To answer this research question this thesis proposed an on-line tuning method for search-control parameters that enables MCTS to be self-adaptive during game play (SA-MCTS). Seven different strategies were introduced to decide how to allocate the available samples to test the parameter combinations: MAB, HE, NMC, LSI, EA, NTBEA and CMA-ES. The performance of on-line parameter tuning has been tested both on the Stanford GGP and the GVG-AI projects.

Results show that among the tested allocation strategies to tune parameters on-line, the ones considering a discrete parameter domain and based on evolutionary algorithms perform best. NTBEA seems to have the best performance overall, but EA is also quite close.

Results for the Stanford GGP project show that on-line parameter tuning is beneficial both for simple and more informed agents, when two parameters are tuned. The performance decreases when tuning more parameters. However, when tuning four parameters, the performance is still close to the one obtained using fixed default parameter values.

Results for the GVG-AI project show that it is harder to tune parameters on-line with much shorter time settings, even when the number of tuned parameters is small. However, it may still be better to tune parameters on-line when fixed parameter settings might be sub-optimal, such as was seen in the game Modality.

It may be concluded that the proposed approach is useful when off-line parameter tuning is infeasible, or in contexts like GGP, both for abstract and real-time games, where parameters cannot be tuned in advance for each game. Moreover, it is useful when off-line tuned values might be sub-optimal for some games, or off-line tuning incurs in the risk of overfitting the values to the set of games selected for the purpose of tuning. It may also be concluded that on-line parameter tuning is robust against different types of opponents.

8.1.4 Search-Control Parameter Randomization

Previous research has shown that adding randomization to certain components of the search might increase its diversification and improve its performance (Beal and Smith, 1994; Bošanský *et al.*, 2016; Chen, 2012). Moreover, the success of on-line search-control parameter tuning on some of the tested games might be partially due to the randomization introduced by exploring different parameter combinations. This might be introducing diversification in the search process, making it explore different parts of the tree that would not be explored keeping the parameters fixed. In a domain like GGP, that deals with many games with different characteristics, adding more randomization might be a good strategy for some games. This has led to the formulation of the fourth research question.

> **Research question 4**: *What is the effect of randomizing search-control parameters for Monte-Carlo Tree Search?*

To answer this research question, this thesis evaluated four different strategies that randomize search-control parameters for MCTS in GGP: randomization per game, per turn, per simulation and per state. Moreover, search-control parameter randomization has been compared with fixed parameter settings and with on-line

parameter tuning both in the framework of the Stanford GGP project and in the framework of the GVG-AI project.

For the Stanford GGP project, results show that the randomization strategy that performs best is the one that randomizes parameter values before each simulation, selecting such values within a predefined reasonable interval. Moreover, results show that for some games randomizing per simulation the value of a single parameter is better than keeping a good value fixed for the whole game. Furthermore, results show that the effect of parameter randomization depends on multiple factors, such as the game being played, which and how many parameters are being randomized and the type of opponent.

It may be concluded that, although not always the best solution for all games, randomization within a given set of values is still beneficial in GGP for games where the fixed parameter settings optimized off-line on a predefined set of games are actually performing poorly. Moreover, parameter randomization might be a valid alternative to on-line parameter tuning when the number of parameters to tune is high and time settings are limited, because the problem of tuning them on-line becomes too hard due to the combinatorial complexity.

8.2 Answer to the Problem Statement

After addressing the four research questions, an answer to the problem statement can be provided.

> **Problem statement**: *How can the performance of Monte-Carlo Tree Search for general game playing be improved?*

The answer to the problem statement is based on the answers to the research questions given above. First, the process of interpreting the game rules written in a declarative language can be sped up by using a PropNet representation of these rules, optimizing the structure of such a PropNet, and embedding the PropNet structure on an FPGA. By speeding up the process of interpreting the game rules, the number of simulations that can be performed by MCTS can be increased. Second, the selection strategy of MCTS can be enhanced using both locally and globally collected information about the available actions. In this case, the best approach is using a mix of global and local information, the first for states that have been visited less and the second for states that have been visited more, like the GRAVE selection strategy does. Furthermore, search-control parameters can be tuned on-line and adapted to each new game being played, using the NTBEA strategy to allocate samples for evaluating parameter combinations. Finally, randomizing search-control parameters within a predefined set of values before each simulation can be used as an alternative to on-line parameter tuning when the number of parameters to tune is high and the time settings are limited.

All the approaches presented in this thesis have been shown to enhance the performance of MCTS for a wide variety of games, without relying on game-specific pre-coded information. Although evaluated only on a subset of all the planning tasks that Artificial General Intelligence (AGI) is aiming to tackle, the games considered

in this thesis present a wide variety of characteristics. They include abstract games, video games, deterministic and non-deterministic games, games with a discrete or continuous game flow, with sequential or simultaneous moves, with constant-sum or variable-sum payoffs, and with different numbers of players. Therefore, the presented MCTS enhancements are promising to also support search and planning for AGI.

8.3 Recommendations for Future Research

This section gives recommendations for future research that result from the research presented in this thesis. First, Subsection 8.3.1 summarizes the recommendations that directly follow from the content presented in the chapters. Subsequently, a higher level discussion of more generic directions for future research is presented in Subsection 8.3.2.

8.3.1 Specific Recommendations

Below is a summary of the recommendations for future research that follow from Chapters 4, 5, 6 and 7.

Speeding Up Game Rule Interpretation. Chapter 4 introduced a fast interpreter for GDL rules based on optimized PropNets, which can also be embedded on FPGAs. Three main directions for future research are suggested. (i) The use of a cache that memorizes results of the queries performed on the software implementation of the PropNet has been shown beneficial only for some of the tested games and only for some stages of the search. The use of the cache for the software implementation of the PropNet could be further improved by devising a strategy to detect for each game if and when the use of a cache is helpful. (ii) Another interesting aspect that future research could consider is the impact that the use of different strategies to propagate truth values among the components of the PropNet would have on the reasoning speed. For the software implementation of the PropNet truth values are computed for one component at a time. This offers two main propagation options that could be tested. The first is forward propagation, which, whenever a component changes truth value, immediately propagates the change to its outputs. The second is backward propagation, which, whenever the truth value of a component needs to be computed, first computes the truth values of its inputs recursively. (iii) Finally, the use of the FPGA-PropNet reasoner can be further investigated. First of all, its performance when integrated in an MCTS agent can be improved by compensating the increased communication overhead. This could be done, for example, by embedding MCTS on the FPGA, or by using hardware with shorter communication latency. Moreover, if the integration of the FPGA-PropNet reasoner within MCTS can be improved to achieve a higher simulation speed, it would be interesting to test other MCTS play-out strategies, to see how the speed increase influences their performance.

Local and Global Information in the Selection Strategy. Chapter 5 investigated three selection strategies, RAVE, GRAVE and HRAVE, to verify how

the search is influenced when the selection strategy is enhanced with information collected at different levels. Two main directions for future research are suggested. (i) The formula proposed more recently by Gelly and Silver (2011) to compute the β parameter could be tested for all the RAVE variants. According to their findings, with this formula the performance of the three RAVE variants could improve further. (ii) In this thesis these strategies were only tested in combination with MAST. Other play-out strategies might influence them in a different way. Testing the combination with the NST play-out strategy could be an idea for future research.

On-line Search-Control Parameter Tuning. Chapter 6 proposed to automatically adapt MCTS on-line by tuning its search control parameters, and tested seven allocation strategies, MAB, HE, NMC, LSI, EA, NTBEA and CMA-ES. Three directions for future research are suggested. (i) It might be interesting to investigate other strategies for parameter tuning. For example, evolutionary strategies for continuous domains, such as *Differential Evolution* (Storn and Price, 1997), to see if they can improve with respect to the performance of CMA-ES, which is the only tested allocation strategy that considers a continuous domain for the parameters. Moreover, given the good performance of NTBEA, other parameter optimization methods that are based on a model of the parameters landscape could be investigated. An example is Sequential Model-based Algorithm Configuration (SMAC) (Hutter *et al.*, 2011), which builds explicit regression models to predict the performance of parameters. This method was shown to be comparable to NTBEA when tuning the parameters of an agent for the Planet Wars game (Lucas *et al.*, 2019). (ii) The self-adaptive MCTS agents proposed in this thesis are not able to choose which and how many parameters to tune. These choices can be seen as extra parameters of the agent and future work could design agents that consider these choices as part of the on-line automatic adaptation. Moreover, performing this decision on-line could help automatically reduce the size of the combinatorial search space by excluding less relevant parameters. (iii) It would also be interesting to see if the devised on-line parameter tuning method can be successfully applied to other domains as well.

Search-Control Parameter Randomization. Chapter 7 evaluated the effect of randomizing search control parameters for MCTS. Two main directions for future research are suggested. (i) As for on-line parameter tuning, it might be profitable to devise a mechanism to automatically detect for each new game if it is worth using parameter randomization, and if so, which parameters are worth randomizing. This mechanism could further be extended to detect on-line if, for a given game, it is better to randomize or tune the parameters. (ii) In GVG-AI, results show that often the performance of parameter randomization and on-line parameter tuning is comparable. For future work it would be interesting to see if increasing time constraints to achieve a few thousands simulations per tick in GVG-AI would make a difference on the performance of parameter randomization and on-line parameter tuning.

8.3.2 General Recommendations

In order to evaluate the performance of MCTS and the proposed enhancements this thesis considered GGP environments that included games with various heterogeneous characteristics. To further enhance MCTS to be an AGI technique, even more challenging domains could be considered. Results in Chapters 6 and 7 have already shown how real-time games are harder to tackle for a GGP agent and more research is required in this direction. Imperfect-information games are another category that was not considered in this thesis, except for games where the imperfect information is introduced by simultaneous moves. The agent for the Stanford GGP project presented in this thesis could be further extended to manage imperfect information games written in GDL-II (Thielscher, 2011). This would give the possibility to further improve MCTS for such games as well, extending on previous work in this direction (Schofield and Thielscher, 2015; Koriche *et al.*, 2016). Other interesting challenges are games with a continuous-action space and games where the agent, together with having no access to the game rules, has no access to a game model. This thesis only focused on the single-player planning track of the GVG-AI competition with agents that only play games with grid physics. Since 2017, the GVG-AI framework supports the creation of agents that consider games with real-world physics, thus with a continuous-action space, and agents that have no access to the game model. The GVG-AI agent evaluated in this thesis could be extended to manage such games as well, giving the opportunity to test MCTS on a wider set of games.

When extending the categories of considered games, results presented in this thesis suggest a promising direction to follow to further improve the performance of MCTS. Previous chapters have shown how a certain search strategy, parameter tuning or randomization technique might work well on a game but not on another. Moreover, techniques that might work for games with certain properties (e.g. discrete abstract games) might not have the same effect on games with different properties (e.g. real-time video games). The more heterogeneous the set of games or tasks that an agent has to perform, the more it seems reasonable to focus on a dynamic approach that adapts to the game or to the category of games being played. An effort to follow this direction is already discussed in the literature. Various approaches have been proposed that consider different game-playing techniques and select among them the ones that seem more suitable for each game or use them in combination to perform decisions (Mendes *et al.*, 2016; Bontrager *et al.*, 2016; Ashlock, Perez-Liebana, and Saunders, 2017; Anderson *et al.*, 2019).

The parameter tuning/randomization strategies presented in this thesis are also a step in the direction of self-adaptive agents. However, the possibility of automatic adaptation is not only limited to the search-control parameters. For example, from Chapter 4 emerged the idea of adapting the interpreter for the game rules. The agent could be provided with a mechanism that, for each game, decides whether to use a cache for the results computed by the interpreter. Moreover, a mechanism could decide at what point in the game to start using the cache (e.g. once the number of states that are re-visited often gets above a certain threshold). Furthermore, in case of a PropNet-based interpreter, if different value propagation strategies are implemented the agent could detect on-line which one is faster for the game at hand,

depending on the estimated average speed. Another example comes from the suggestion of testing a different formula for the computation of the β parameter used by the RAVE variants presented in Chapter 5. As for the RAVE variants, also for other strategies multiple formulas might be proposed to compute certain parameters, and such formulas might perform differently depending on the game. This suggest that formulas might also be changed on-line depending on their performance. For example, a formula might be considered as a variable that has to be selected from a set of feasible formulas. Many ideas for improving automatic adaptation of MCTS come from Chapters 6 and 7. As already mentioned, agents could select on-line whether to tune or randomize the parameters and which and how many of them. This could also be extended by changing on-line which tuning or randomization strategy is used, instead of simply considering a single tuning and a single randomizing strategy. Moreover, self-adaptation could be pushed further by tuning on-line also the parameters that control the tuning strategies. Although, caution is necessary in this case to avoid the risk that designing such strategies would introduce even more parameters that require to be tuned.

Many of the ideas discussed so far for implementing a self-adaptive agent assume that the aspect of the agent that is being adapted (e.g. the interpreter, a strategy, some search parameters, a formula) can only vary within a finite set of predefined configurations. An alternative that might be worth investigating would be to let the agent explore new feasible configurations on-line. For parameter tuning, for example, agents could adapt the set of feasible values over time. They could remove values that turn out to perform poorly and add more values close to the ones that seem to perform well, with the possibility of finding even better values. Another example takes inspiration from the work of Bravi *et al.* (2017), which proposes to evolve variants of the UCB family of formulas for each specific game. This evolution is performed off-line, but it could be adapted to take place on-line and it could be used to evolve other types of formulas as well.

While adapting the agent automatically on-line would help to better deal with the heterogeneity of the games, another challenge becomes soon apparent. In many domains, only a short amount of time is available to the agents for making decisions. The more aspects of the agents are being adapted on-line and the more configurations are considered or even generated, the more computational resources, among which time, are required to evaluate all possibilities. It has already been seen in Chapter 6 that, when time is limited, it is hard to even tune a few parameters with a small finite set of possible values. One final suggestion for future research that might help mitigate this issue is to look into designing a strategy that does not adapt the agent from scratch, but starts from configurations that might work well for the game at hand. For example, similarly to what has been done by Mendes *et al.* (2016) and Bontrager *et al.* (2016), the agent could extract some features (e.g. number of players, whether it is played on a board, whether there are NPCs and how many, whether there are capture moves, etc.) from the games it plays and learn which configurations work best for certain features. Therefore, whenever a new game is played, the agent would first extract the features and then use configurations that worked well with such features in the past. These configurations would then be further adapted on-line. Note that this approach does not contradict the principles

of GGP, because, even if the agent uses prior knowledge, it is not hard coded nor specific for a single domain. Finally, to make the approach even more generic, more game features could be generated automatically by the agent.

References

Abdo, Ashraf, Edelkamp, Stefan, and Lawo, Michael (2016). Nested Rollout Policy Adaptation for Optimizing Vehicle Selection in Complex VRPs. *41st IEEE Conference on Local Computer Networks Workshops, LCN Workshops 2016*, pp. 213–221. [271]

Abramson, Bruce (1990). Expected-Outcome: A General Model of Static Evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 12, No. 2, pp. 182–193. [11, 20]

Al-Kanj, Lina, Powell, Warren B., and Bouzaiene-Ayari, Belgacem (2016). The Information-Collecting Vehicle Routing Problem: Stochastic Optimization for Emergency Storm Response. *arXiv preprint arXiv:1605.05711.* [271]

Anderson, Damien, Guerrero-Romero, Cristina, Perez-Liebana, Diego, Rodgers, Philip, and Levine, John (2019). Ensemble Decision Systems for General Video Game Playing. *2019 IEEE Conference on Games (COG)*, IEEE. [210]

Arneson, Broderick, Hayward, Ryan B., and Henderson, Philip (2010). Monte Carlo Tree Search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 2, No. 4, pp. 251–258. [11]

Arora, Akash, Fitch, Robert, and Sukkarieh, Salah (2017). An Approach to Autonomous Science by Modeling Geological Knowledge in a Bayesian Framework. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3803–3810, IEEE. [271]

Ashlock, Daniel (2006). *Evolutionary Computation for Modeling and Optimization.* Springer Science & Business Media. [125, 137]

Ashlock, Daniel, Perez-Liebana, Diego, and Saunders, Amanda (2017). General Video Game Playing Escapes the No Free Lunch Theorem. *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 17–24, IEEE. [210]

Auer, Peter, Cesa-Bianchi, Nicoló, Freund, Yoav, and Schapire, Robert E. (1995). Gambling in a Rigged Casino: The Adversarial Multi-Armed Bandit Problem. *36th Annual Symposium on Foundations of Computer Science*, pp. 322–331. [24]

Auer, Peter, Cesa-Bianchi, Nicolò, and Fischer, Paul (2002). Finite-Time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, Vol. 47, Nos. 2–3, pp. 235–256. [11, 20, 22, 24, 124, 128, 129]

Baier, Hendrik (2015). *Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains*. Ph.D. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. [127]

Baier, Hendrik, Sattaur, Adam, Powley, Edward J., Devlin, Sam, Rollason, Jeff, and Cowling, Peter I. (2018). Emulating Human Play in a Leading Mobile Card Game. *IEEE Transactions on Games*. In press. [269]

Beal, Don F. and Smith, Martin C. (1994). Random Evaluations in Chess. *ICCA Journal*, Vol. 17, No. 1, pp. 3–9. [14, 171, 172, 206]

Beattie, Charles, Leibo, Joel Z., Teplyashin, Denis, Ward, Tom, Wainwright, Marcus, Küttler, Heinrich, Lefrancq, Andrew, Green, Simon, Valdés, Víctor, Sadik, Amir, Schrittwieser, Julian, Anderson, Keith, York, Sarah, Cant, Max, Cain, Adam, Bolton, Adrian, Gaffney, Stephen, King, Helen, Hassabis, Demis, Legg, Shane, and Petersen, Stig (2016). DeepMind Lab. *arXiv preprint arXiv:1612.03801*. [10]

Bellemare, Marc G., Naddaf, Yavar, Veness, Joel, and Bowling, Michael (2013). The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, Vol. 47, pp. 253–279. [9]

Bertsimas, Dimitris, Griffith, J. Daniel, Gupta, Vishal, Kochenderfer, Mykel J., and Mišić, Velibor V. (2017). A Comparison of Monte Carlo Tree Search and Rolling Horizon Optimization for Large-Scale Dynamic Resource Allocation Problems. *European Journal of Operational Research*, Vol. 263, No. 2, pp. 664–678. [271]

Best, Graeme, Cliff, Oliver M., Patten, Timothy, Mettu, Ramgopal R., and Fitch, Robert (2019). Dec-MCTS: Decentralized Planning for Multi-Robot Active Perception. *The International Journal of Robotics Research*, Vol. 38, Nos. 2–3, pp. 316–337. [270]

Bhonker, Nadav, Rozenberg, Shai, and Hubara, Itay (2016). Playing SNES in the Retro Learning Environment. *arXiv preprint arXiv:1611.02205*. [10]

Billings, Darse, Davidson, Aaron, Schaeffer, Jonathan, and Szafron, Duane (2002). The Challenge of Poker. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 201–240. [6]

Bischl, Bernd, Kerschke, Pascal, Kotthoff, Lars, Lindauer, Marius, Malitsky, Yuri, Fréchette, Alexandre, Hoos, Holger, Hutter, Frank, Leyton-Brown, Kevin, Tierney, Kevin, and Vanschoren, Joaquin (2016). Aslib: A Benchmark Library for Algorithm Selection. *Artificial Intelligence*, Vol. 237, pp. 41–58. [125]

Björnsson, Yngvi and Finnsson, Hilmar (2009). CadiaPlayer: A Simulation-Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 1, No. 1, pp. 4–15. [35, 58, 59, 106, 172]

Björnsson, Yngvi and Marsland, T. Anthony (2003). Learning Extension Parameters in Game-Tree Search. *Information Sciences*, Vol. 154, Nos. 3–4, pp. 95–118. [125]

Bontrager, Philip, Khalifa, Ahmed, Mendes, Andre, and Togelius, Julian (2016). Matching Games and Algorithms for General Video Game Playing. *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)* (eds. N.R. Sturtevant and B. Magerko), pp. 122–128, AAAI Press. [149, 210, 211]

Bošanskỳ, Branislav, Lisỳ, Viliam, Lanctot, Marc, Čermák, Jiří, and Winands, Mark H.M. (2016). Algorithms for Computing Strategies in Two-Player Simultaneous Move Games. *Artificial Intelligence*, Vol. 237, pp. 1–40. [14, 33, 35, 171, 172, 206]

Bowling, Michael, Burch, Neil, Johanson, Michael, and Tammelin, Oskari (2015). Heads-up Limit Hold'em Poker is Solved. *Science*, Vol. 347, No. 6218, pp. 145–149. [6]

Bravi, Ivan, Khalifa, Ahmed, Holmgård, Christoffer, and Togelius, Julian (2017). Evolving Game-Specific UCB Alternatives for General Video Game Playing. *Applications of Evolutionary Computation* (eds. G. Squillero and K. Sim), Vol. 10199 of *LNCS*, pp. 393–406, Springer. [211]

Breuker, Dennis M. (1998). *Memory versus Search in Games*. Ph.D. thesis, Department of Computer Science, Maastricht University, Maastricht, The Netherlands. [42]

Brockman, Greg, Cheung, Vicki, Pettersson, Ludwig, Schneider, Jonas, Schulman, John, Tang, Jie, and Zaremba, Wojciech (2016). OpenAI Gym. *arXiv preprint arXiv:1606.01540*. [10]

Browne, Cameron B. (2010). On the Dangers of Random Playouts. *ICGA Journal*, Vol. 34, No. 1, pp. 25–26. [11]

Browne, Cameron (2012). Elegance in Game Design. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 3, pp. 229–240. [270]

Browne, Cameron (2013). UCT for PCG. *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pp. 137–144, IEEE. [270]

Browne, Cameron (2018). Modern Techniques for Ancient Games. *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 490–497, IEEE. [9]

Browne, Cameron B., Powley, Edward, Whitehouse, Daniel, Lucas, Simon M., Cowling, Peter I., Rohlfshagen, Philipp, Tavener, Stephen, Perez, Diego, Samothrakis, Spyridon, and Colton, Simon (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 1, pp. 1–43. [1, 20, 36, 123]

Brown, Stephen D., Francis, Robert J., Rose, Jonathan, and Vranesic, Zvonko G. (2012). *Field-Programmable Gate Arrays*, Vol. 180. Springer Science & Business Media. [95]

Brügmann, Bernd (1993). Monte Carlo Go. Technical report, Max Planck Institute of Physics, München, Germany. [104]

Burke, Edmund K., Gendreau, Michel, Hyde, Matthew, Kendall, Graham, Ochoa, Gabriela, Özcan, Ender, and Qu, Rong (2013). Hyper-Heuristics: A Survey of the State of the Art. *Journal of the Operational Research Society*, Vol. 64, No. 12, pp. 1695–1724. [125]

Campbell, Murray (1985). The Graph-History Interaction: On Ignoring Position History. *1985 ACM Annual Conference on the Range of Computing: Mid-80's Perspective*, pp. 278–280, ACM. [42]

Cazenave, Tristan (2015). Generalized Rapid Action Value Estimation. *Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)* (eds. Q. Yang and M. Wooldridge), pp. 754–760, AAAI Press. [13, 103, 106, 117, 119, 124, 204, 205]

Cazenave, Tristan and Abdallah, Saffidine (2010). Monte-Carlo Hex. *XIIIth Board Games Studies Colloquium*, Paris, France. [37]

Cazenave, Tristan and Hamida, Sana Ben (2015). Forecasting Financial Volatility Using Nested Monte Carlo Expression Discovery. *2015 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 726–733, IEEE. [271]

Cazenave, Tristan, Balbo, Flavien, and Pinson, Suzanne (2009). Using a Monte-Carlo Approach for Bus Regulation. *12th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pp. 340–345, IEEE. [271]

Chaslot, Guillaume M.J.-B., Winands, Mark H.M., and Herik, H. Jaap van den (2008a). Parallel Monte-Carlo Tree Search. *Computers and Games* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *LNCS*, pp. 60–71, Springer. [96]

Chaslot, Guillaume M.J.-B., Winands, Mark H.M., Herik, H. Jaap van den, Uiterwijk, Jos W.H.M., and Bouzy, Bruno (2008b). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357. [23, 103, 126, 204]

Chaslot, Guillaume M.J.-B., Winands, Mark H.M., Szita, István, and Herik, H. Jaap van den (2008c). Cross-Entropy for Monte-Carlo Tree Search. *ICGA Journal*, Vol. 31, No. 3, pp. 145–156. [125, 126]

Chen, Keh-Hsun (2012). Dynamic Randomization and Domain Knowledge in Monte-Carlo Tree Search for Go Knowledge-Based Systems. *Knowledge-Based Systems*, Vol. 34, pp. 21–25. [14, 171, 172, 175, 178, 206]

Childs, Benjamin E., Brodeur, James H., and Kocsis, Levente (2008). Transpositions and Move Groups in Monte Carlo Tree Search. *2008 IEEE Symposium On Computational Intelligence and Games (CIG)*, pp. 389–395, IEEE. [43, 46, 47, 60]

Churchill, David and Buro, Michael (2015). Hierarchical Portfolio Search: Prismata's Robust AI Architecture for Games with Large Search Spaces. *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)* (eds. A. Jhala and N.R. Sturtevant), pp. 16–22, AAAI Press. [270]

Clune, James (2007). Heuristic Evaluation Functions for General Game Playing. *Twenty-Second AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1134–1139, AAAI Press. [58]

Cole, Nicholas, Louis, Sushil J., and Miles, Chris (2004). Using a Genetic Algorithm to Tune First-Person Shooter Bots. *2004 IEEE Congress on Evolutionary Computation (CEC)*, pp. 139–145, IEEE. [125, 126]

Couëtoux, Adrien and Doghmen, Hassen (2011). Adding Double Progressive Widening to Upper Confidence Trees to Cope with Uncertainty in Planning Problems. *9th European Workshop on Reinforcement Learning (EWRL-9)*. [271]

Coulom, Rémi (2007a). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Computers and Games* (eds. H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers), Vol. 4630 of *LNCS*, pp. 72–83, Springer. [6, 11, 20, 23, 147]

Coulom, Rémi (2007b). Computing "Elo Ratings" of Move Patterns in the Game of Go. *ICGA Journal*, Vol. 30, No. 4, pp. 198–208. [6]

Coulom, Rémi (2012). CLOP: Confident Local Optimization for Noisy Black-Box Parameter Tuning. *Advances in Computer Games* (eds. H.J. van den Herik and A. Plaat), Vol. 7168 of *LNCS*, pp. 146–157, Springer. [125]

Cox, Evan, Schkufza, Eric, Madsen, Ryan, and Genesereth, Michael R. (2009). Factoring General Games Using Propositional Automata. *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA)* (eds. Y. Björnsson, P. Stone, and M. Thielscher), pp. 13–20. [13, 71, 72, 204]

Den Teuling, Niek G.P. and Winands, Mark H.M. (2012). Monte-Carlo Tree Search for the Simultaneous Move Game Tron. *Computer Games Workshop at ECAI 2012*, pp. 126–141, Montpellier, France. [34]

Dieb, Thaer M., Ju, Shenghong, Shiomi, Junichiro, and Tsuda, Koji (2019). Monte Carlo Tree Search for Materials Design and Discovery. *MRS Communications*, Vol. 9, No. 2, pp. 532–536. [271]

Draper, Steve and Rose, Andrew (2014). Sancho GGP Player. `http://sanchoggp.blogspot.nl/2014/07/sancho-is-ggp-champion-2014.html`. [58, 72]

Ebner, Marc, Levine, John, Lucas, Simon M., Schaul, Tom, Thompson, Tommy, and Togelius, Julian (2013). Towards a Video Game Description Language. *Artificial and Computational Intelligence in Games* (eds. S.M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius), Vol. 6 of *Dagstuhl Follow-Ups*, pp. 85–100. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. [9, 61]

Edelkamp, Stefan, Gath, Max, Greulich, Christoph, Humann, Malte, Herzog, Otthein, and Lawo, Michael (2016). Monte-Carlo Tree Search for Logistics. *Commercial Transport* (eds. U. Clausen, H. Friedrich, C. Thaller, and C. Geiger), LNL, pp. 427–440, Springer. [12, 271]

Finnsson, Hilmar (2012a). Generalized Monte-Carlo Tree Search Extensions for General Game Playing. *Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1550–1556, AAAI Press. [36]

Finnsson, Hilmar (2012b). *Simulation-Based General Game Playing.* Ph.D. thesis, School of Computer Science, Reykjavik University, Reykjavik, Iceland. [49, 56, 109, 147, 176, 255]

Finnsson, Hilmar and Björnsson, Yngvi (2008). Simulation-Based Approach to General Game Playing. *Twenty-Third AAAI Conference on Artificial Intelligence (AAAI)*, pp. 259–264, AAAI Press. [38, 104]

Finnsson, Hilmar and Björnsson, Yngvi (2010). Learning Simulation Control in General Game-Playing Agents. *Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI)*, pp. 954–959, AAAI Press. [1, 11, 13, 24, 37, 103, 105, 123, 204]

Gaina, Raluca D., Perez-Liebana, Diego, and Lucas, Simon M. (2016). General Video Game for 2 Players: Framework and Competition. *8th Computer Science and Electronic Engineering Conference (CEEC)*, pp. 186–191, IEEE. [61]

Gaina, Raluca D., Liu, Jialin, Lucas, Simon M., and Perez-Liebana, Diego (2017). Analysis of Vanilla Rolling Horizon Evolution Parameters in General Video Game Playing. *Applications of Evolutionary Computation* (eds. G. Squillero and K. Sim), Vol. 10199 of *LNCS*, pp. 418–434, Springer. [126, 149]

Gaina, Raluca D., Couëtoux, Adrien, Soemers, Dennis J.N.J., Winands, Mark H.M., Vodopivec, Tom, Kirchgeßner, Florian, Liu, Jialin, Lucas, Simon M., and Perez-Liebana, Diego (2018). The 2016 Two-Player GVGAI Competition. *IEEE Transactions on Games*, Vol. 10, No. 2, pp. 209–220. [27]

Geffner, Tomás and Geffner, Hector (2015). Width-Based Planning for General Video-Game Playing. *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)* (eds. A. Jhala and N.R. Sturtevant), pp. 23–29, AAAI Press. [68]

Gelly, Sylvain and Silver, David (2007). Combining Online and Offline Knowledge in UCT. *24th International Conference on Machine Learning (ICML)* (ed. Z. Ghahramani), pp. 273–280, ACM. [13, 37, 103, 105, 106, 204]

Gelly, Sylvain and Silver, David (2008). Achieving Master Level Play in 9 × 9 Computer Go. *Twenty-Third AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1537–1540, AAAI Press. [6]

Gelly, Sylvain and Silver, David (2011). Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go. *Artificial Intelligence*, Vol. 175, No. 11, pp. 1856–1875. [6, 13, 37, 103, 105, 106, 121, 204, 209]

Gelly, Sylvain and Wang, Yizao (2007). MoGo Wins 19x19 Go Tournament. *ICGA Journal*, Vol. 30, No. 2, pp. 111–112. [6]

Gelly, Sylvain, Wang, Yizao, Munos, Rémi, and Teytaud, Olivier (2006). Modification of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA, France. [6, 24]

Gelly, Sylvain, Kocsis, Levente, Schoenauer, Marc, Sebag, Michele, Silver, David, Szepesvári, Csaba, and Teytaud, Olivier (2012). The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. *Communications of the ACM*, Vol. 55, No. 3, pp. 106–113. [5]

Genesereth, Michael and Thielscher, Michael (2014). *General Game Playing*, Vol. 8 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool Publishers. [5, 13, 51, 57, 71, 72, 204]

Genesereth, Michael, Love, Nathaniel, and Pell, Barney (2005). General Game Playing: Overview of the AAAI Competition. *AI Magazine*, Vol. 26, No. 2, pp. 62–72. [9, 35, 51, 58]

Ginsberg, Matthew L. (2001). GIB: Imperfect Information in a Computationally Challenging Game. *Journal of Artificial Intelligence Research*, Vol. 14, pp. 303–358. [11]

Goertzel, Ben (2014). Artificial General Intelligence: Concept, State of the Art, and Future Prospects. *Journal of Artificial General Intelligence*, Vol. 5, No. 1, pp. 1–48. [1]

Goertzel, Ben and Pennachin, Cassio (2007). *Artificial General Intelligence*. Springer. [1, 8]

Goldhoorn, Alex, Garrell, Anaís, Alquézar, René, and Sanfeliu, Alberto (2014). Continuous Real Time POMCP to Find-and-Follow People by a Humanoid Service Robot. *14th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pp. 741–747, IEEE. [12, 270]

Golpayegani, Fatemeh, Dusparic, Ivana, and Clarke, Siobhán (2015). Collaborative, Parallel Monte Carlo Tree Search for Autonomous Electricity Demand Management. *2015 Sustainable Internet and ICT for Sustainability (SustainIT)*, pp. 1–8. [271]

Greenblatt, Richard D., Eastlake III, Donald E., and Crocker, Stephen D. (1967). The Greenblatt Chess Program. *Proceedings of the Fall Joint Computer Conference*, pp. 801–810. [41]

Guo, Qingyu, An, Bo, and Kolobov, Andrey (2015). Approximation Approaches for Solving Security Games with Surveillance Cost: A Preliminary Study. *2015 International Workshop on Issues with Deployment of Emerging Agent-based Systems (IDEAS).* [271]

Hansen, Nikolaus (2016). The CMA Evolution Strategy: A Tutorial. *arXiv preprint arXiv:1604.00772.* [142, 144, 146]

Hansen, Nikolaus, Müller, Sibylle D., and Koumoutsakos, Petros (2003). Reducing the Time Complexity of the Derandomized Evolution Strategy with Covariance Matrix Adaptation (CMA-ES). *Evolutionary Computation*, Vol. 11, No. 1, pp. 1–18. [124, 128, 142]

Harsanyi, John C. (1967). Games with Incomplete Information Played by "Bayesian" Players, I–III Part I. The Basic Model. *Management Science*, Vol. 14, No. 3, pp. 159–182. [4]

Hart, Sergiu and Mas-Colell, Andreu (2000). A Simple Adaptive Procedure Leading to Correlated Equilibrium. *Econometrica*, Vol. 68, No. 5, pp. 1127–1150. [24]

Hauer, Bradley, Hayward, Ryan, and Kondrak, Grzegorz (2014). Solving Substitution Ciphers with Combined Language Models. *25th International Conference on Computational Linguistics (COLING)* (eds. J. Hajic and J. Tsujii), pp. 2314–2325. [271]

Hausknecht, Matthew, Lehman, Joel, Miikkulainen, Risto, and Stone, Peter (2014). A Neuroevolution Approach to General Atari Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 6, No. 4, pp. 355–366. [142]

Heinz, Ernst A. (2001). Self-Play, Deep Search and Diminishing Returns. *ICGA Journal*, Vol. 24, No. 2, pp. 75–79. [115]

Hennes, Daniel and Izzo, Dario (2015). Interplanetary Trajectory Planning with Monte Carlo Tree Search. *Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)* (eds. Q. Yang and M. Wooldridge), pp. 769–775, AAAI Press. [12, 271]

Hocine, Nadia, Gouaïch, Abdelkader, and Cerri, Stefano A. (2014). Dynamic Difficulty Adaptation in Serious Games for Motor Rehabilitation. *Games for Training, Education, Health and Sports* (eds. S. Göbel and J. Wiemeyer), Vol. 8395 of *LNCS*, pp. 115–128, Springer. [270]

Hoock, Jean-Baptiste, Lee, Chang-Shing, Rimmel, Arpad, Teytaud, Fabien, Teytaud, Olivier, and Wang, Mei-Hui (2010). Intelligent Agents for the Game of Go. *IEEE Computational Intelligence Magazine*, Vol. 5, No. 4, pp. 28–42. [37]

Hsu, Feng-Hsiung (2002). *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, Princeton, NJ, USA. [5]

Hutter, Frank, Hoos, Holger H., and Leyton-Brown, Kevin (2011). Sequential Model-Based Optimization for General Algorithm Configuration. *International Conference on Learning and Intelligent Optimization*, pp. 507–523, Springer. [168, 209]

Jiang, Daniel R., Al-Kanj, Lina, and Powell, Warren B. (2017). Monte Carlo Tree Search with Sampled Information Relaxation Dual Bounds. *arXiv preprint arXiv:1704.05963*. [271]

Joppen, Tobias, Moneke, Miriam Ulrike, Schröder, Nils, Wirth, Christian, and Fürnkranz, Johannes (2018). Informed Hybrid Game Tree Search for General Video Game Playing. *IEEE Transactions on Games*, Vol. 10, No. 1, pp. 78–90. [66, 68]

Karnin, Zohar, Koren, Tomer, and Somekh, Oren (2013). Almost Optimal Exploration in Multi-Armed Bandits. *30th International Conference on Machine Learning (ICML)* (eds. S. Dasgupta and D. McAllester), pp. 1238–1246. [136]

Kartal, Bilal (2015). Monte Carlo Tree Search with Useful Cycles for Motion Planning. *2015 IEEE International Conference on Robotics and Automation (ICRA) Ph.D. Forum*. [270]

Kartal, Bilal, Koenig, John, and Guy, Stephen J. (2014). User-Driven Narrative Variation in Large Story Domains Using Monte Carlo Tree Search. *2014 International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pp. 69–76. [270]

Kartal, Bilal, Sohre, Nick, and Guy, Stephen J. (2016). Data Driven Sokoban Puzzle Generation with Monte Carlo Tree Search. *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)* (eds. N.R. Sturtevant and B. Magerko), pp. 58–64, AAAI Press. [270]

Kempka, Michał, Wydmuch, Marek, Runc, Grzegorz, Toczek, Jakub, and Jaśkowski, Wojciech (2016). Vizdoom: A Doom-Based AI Research Platform for Visual Reinforcement Learning. *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 341–348, IEEE. [7]

Kishimoto, Akihiro and Müller, Martin (2004). A General Solution to the Graph History Interaction Problem. *Nineteenth National Conference on Artificial Intelligence (AAAI)*, pp. 644–649, AAAI Press. [42]

Knuth, Donald E. and Moore, Ronald W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [10]

Kocsis, Levente and Szepesvári, Csaba (2006). Bandit Based Monte-Carlo Planning. *Machine Learning: ECML 2006* (eds. J. Fürnkranz, T. Scheffer, and M.

Spiliopoulou), Vol. 4212 of *LNCS*, pp. 282–293. Springer. [6, 11, 13, 20, 23, 30, 103]

Kocsis, Levente, Szepesvári, Csaba, and Willemson, Jan (2006a). Improved Monte-Carlo Search. Technical Report 1, University of Tartu, Estonia. [46]

Kocsis, Levente, Szepesvári, Csaba, and Winands, Mark H.M. (2006b). RSPSA: Enhanced Parameter Optimization in Games. *Advances in Computer Games* (eds. H.J. van den Herik, S.-C. Hsu, T.-S. Hsu, and H.H.L.M. Donkers), Vol. 4250 of *LNCS*, pp. 39–56, Springer. [125]

Koriche, Frédéric, Lagrue, Sylvain, Piette, Éric, and Tabary, Sébastien (2016). Stochastic Constraint Programming for General Game Playing with Imperfect Information. *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA)* (eds. S. Schiffel, M. Thielscher, and J. Togelius), pp. 39–46. [210]

Kowalski, Jakub (2016). Towards a Real-Time Game Description Language. *8th International Conference on Agents and Artificial Intelligence (ICAART)* (eds. H.J. van den Herik and J. Filipe), Vol. 2, pp. 494–499. [52]

Kowalski, Jakub, Mika, Maksymilian, Sutowicz, Jakub, and Szykuła, Marek (2019). Regular Boardgames. *Thirty-Third AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1699–1706, AAAI Press. [9]

Kunanusont, Kamolwan, Gaina, Raluca D., Liu, Jialin, Perez-Liebana, Diego, and Lucas, Simon M. (2017). The N-Tuple Bandit Evolutionary Algorithm for Automatic Game Improvement. *2017 IEEE Congress on Evolutionary Computation (CEC)*, pp. 2201–2208, IEEE. [124, 126, 128, 138]

Laird, John and VanLent, Michael (2001). Human-Level AI's Killer Application: Interactive Computer Games. *AI Magazine*, Vol. 22, No. 2, pp. 15–26. [6]

Lai, Tze Leung and Robbins, Herbert (1985). Asymptotically Efficient Adaptive Allocation Rules. *Advances in Applied Mathematics*, Vol. 6, No. 1, pp. 4–22. [22]

Lanctot, Marc, Wittlinger, Christopher, Winands, Mark H.M., and Den Teuling, Niek G.P. (2013). Monte Carlo Tree Search for Simultaneous Move Games: A Case Study in the Game of Tron. *25th Benelux Conference on Artificial Intelligence (BNAIC)* (eds. K. Hindriks, M. de Weerdt, B. van Riemsdijk, and M. Warnier), pp. 104–111. [34]

Laschet, Cliff (2014). Home Care Service Selection Using Predictive Models and Monte-Carlo Tree Search. M.Sc. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. [270]

Lenz, David, Kessler, Tobias, and Knoll, Alois (2016). Tactical Cooperative Planning for Autonomous Highway Driving Using Monte-Carlo Tree Search. *2016 IEEE Intelligent Vehicles Symposium (IV)*, pp. 447–453, IEEE. [271]

Levine, John, Congdon, Clare Bates, Ebner, Marc, Kendall, Graham, Lucas, Simon M., Miikkulainen, Risto, Schaul, Tom, and Thompson, Tommy (2013). General Video Game Playing. *Artificial and Computational Intelligence in Games* (eds. S.M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius), Vol. 6 of *Dagstuhl Follow-Ups*, pp. 77–83. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. [9, 61]

Liu, Jialin, Togelius, Julian, Perez-Liebana, Diego, and Lucas, Simon M. (2017). Evolving Game Skill-Depth Using General Video Game AI Agents. *2017 IEEE Congress on Evolutionary Computation (CEC)*, pp. 2299–2307, IEEE. [126]

Lorentz, Richard J. (2008). Amazons Discover Monte-Carlo. *Computers and Games* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *LNCS*, pp. 13–24, Springer. [11]

Lorentz, Richard J. (2011). Improving Monte-Carlo Tree Search in Havannah. *Computers and Games* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 105–115, Springer. [37]

Love, Nathaniel, Hinrichs, Timothy, and Genesereth, Michael (2006). General Game Playing: Game Description Language Specification. Technical Report LG-2006-01, Stanford Logic Group, Stanford University, Stanford, CA. [9, 12, 51, 52]

Lucas, Simon M., Liu, Jialin, and Perez-Liebana, Diego (2018). The N-Tuple Bandit Evolutionary Algorithm for Game Agent Optimisation. *2018 IEEE Congress on Evolutionary Computation (CEC)*, pp. 221–229, IEEE. [124, 128, 138]

Lucas, Simon M., Liu, Jialin, Bravi, Ivan, Gaina, Raluca D., Woodward, John, Volz, Vanessa, and Perez-Liebana, Diego (2019). Efficient Evolutionary Methods for Game Agent Optimisation: Model-Based is Best. *arXiv preprint arXiv:1901.00723*. [168, 209]

Mallett, Jeff and Lefler, Mark (1998). Zillions of Games. Available online at: www.zillions-of-games.com. [8]

Marecki, Janusz, Tesauro, Gerry, and Segal, Richard (2012). Playing Repeated Stackelberg Games with Unknown Opponents. *11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 821–828. [271]

Marks, Christopher, Darken, Christian, Buss, Arnie, Lin, Kyle, and Alt, Jonathan (2013). Mission Command Analysis Using Monte Carlo Tree Search. Technical Report TRAC-M-TR-13-050, TRADOC Analysis Center, Monterey, CA. [271]

Méhat, Jean and Cazenave, Tristan (2010). Ary, a General Game Playing Program. *XIIIth Board Games Studies Colloquium*, Paris, France. [58]

Mendes, Andre, Togelius, Julian, and Nealen, Andy (2016). Hyper-Heuristic General Video Game Playing. *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 94–101, IEEE. [125, 210, 211]

Metropolis, Nicholas and Ulam, Stanislaw (1949). The Monte Carlo Method. *Journal of the American Statistical Association*, Vol. 44, No. 247, pp. 335–341. [20]

Mirheli, Amir and Hajibabai, Leila (2019). Utilization Management and Pricing of Parking Facilities Under Uncertain Demand and User Decisions. *IEEE Transactions on Intelligent Transportation Systems*. In press. [271]

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A., Veness, Joel, Bellemare, Marc G., Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K., Ostrovski, Georg, et al. (2015). Human-Level Control Through Deep Reinforcement Learning. *Nature*, Vol. 518, No. 7540, pp. 529–533. [9]

Müller, Martin (2002). Computer Go. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 145–179. [5]

Murray, Harold James Ruthven (1952). *A History of Board-Games Other Than Chess*. Oxford University Press, Oxford. [2]

Nelson, Mark J. (2016). Investigating Vanilla MCTS Scaling on the GVG-AI Game Corpus. *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 403–409, IEEE. [149]

Neumann, John von and Morgenstern, Oskar (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ, USA. [5, 10]

Nijssen, J. (Pim) A.M. (2013). *Monte-Carlo Tree Search for Multi-Player Games*. Ph.D. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. [108]

Nijssen, J. (Pim) A.M. and Winands, Mark H.M. (2011). Enhancements for Multi-Player Monte-Carlo Tree Search. *Computers and Games* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 238–249, Springer. [13, 37, 103, 108, 124, 204]

Ontañón, Santiago (2013). The Combinatorial Multi-Armed Bandit Problem and Its Application to Real-Time Strategy Games. *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)* (eds. G. Sukthankar and I. Horswill), pp. 58–64, AAAI Press. [124, 128, 129, 132]

Ontañón, Santiago (2017). Combinatorial Multi-Armed Bandits for Real-Time Strategy Games. *Journal of Artificial Intelligence Research*, Vol. 58, pp. 665–702. [124, 128, 132]

Ontañón, Santiago, Synnaeve, Gabriel, Uriarte, Alberto, Richoux, Florian, Churchill, David, and Preuss, Mike (2013). A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 5, No. 4, pp. 293–311. [7, 145]

OpenAI (2018). Gym Retro. Available online at: https://github.com/openai/retro. [10]

Palay, Andrew J. (1983). *Searching with Probabilities.* Ph.D. thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA. [42]

Parlett, David (2018). *History of Board Games.* Echo Point Books and Media. [2]

Patten, Timothy, Martens, Wolfram, and Fitch, Robert (2018). Monte Carlo Planning for Active Object Classification. *Autonomous Robots*, Vol. 42, No. 2, pp. 391–421. [270]

Pell, Barney (1993). *Strategy Generation and Evaluation for Meta-Game Playing.* Ph.D. thesis, Trinity College, University of Cambridge, Cambridge, England. [8]

Pepels, Tom, Winands, Mark H.M., and Lanctot, Marc (2014). Real-Time Monte Carlo Tree Search in Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in games*, Vol. 6, No. 3, pp. 245–257. [11, 47, 48]

Perez-Liebana, Diego (2018). The GVG-AI Competition Framework. `https://github.com/GAIGResearch/GVGAI`. [64, 149, 177]

Perez-Liebana, Diego, Samothrakis, Spyridon, and Lucas, Simon M. (2014). Knowledge-Based Fast Evolutionary MCTS for General Video Game Playing. *2014 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 68–75, IEEE. [68]

Perez-Liebana, Diego, Dieskau, Jens, Hunermund, Martin, Mostaghim, Sanaz, and Lucas, Simon M. (2015). Open Loop Search for General Video Game Playing. *2015 Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pp. 337–344, ACM. [27]

Perez-Liebana, Diego, Samothrakis, Spyridon, Togelius, Julian, Schaul, Tom, Lucas, Simon M., Couëtoux, Adrien, Lee, Jerry, Lim, Chong-U, and Thompson, Tommy (2016). The 2014 General Video Game Playing Competition. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 8, No. 3, pp. 229–243. [1, 10, 11, 27, 64, 65, 66, 149, 177]

Perez-Liebana, Diego, Stephenson, Matthew, Gaina, Raluca D., Renz, Jochen, and Lucas, Simon M. (2017). Introducing Real World Physics and Macro-Actions to General Video Game AI. *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 248–255, IEEE. [61]

Perez-Liebana, Diego, Liu, Jialin, Khalifa, Ahmed, Gaina, Raluca D., Togelius, Julian, and Lucas, Simon M. (2018). General Video Game AI: a Multi-Track Framework for Evaluating Agents, Games and Content Generation Algorithms. *arXiv preprint arXiv:1802.10363.* [27, 61, 68]

Perick, Pierre, St-Pierre, David L., Maes, Francis, and Ernst, Damien (2012). Comparison of Different Selection Strategies in Monte-Carlo Tree Search for the Game of Tron. *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 242–249, IEEE. [35]

Piccione, Peter A. (1980). In Search of the Meaning of Senet. *Archaeology*, Vol. 33, No. 4, pp. 55–58. [2]

Piette, Éric (2016). *General Game Playing*. Ph.D. thesis, Université d'Artois, France. [58]

Piette, Éric, Soemers, Dennis J.N.J., Stephenson, Matthew, Sironi, Chiara F., Winands, Mark H.M., and Browne, Cameron (2019). Ludii-The Ludemic General Game System. *arXiv preprint arXiv:1905.05013*. [9]

Pinheiro, Miguel A., Kybic, Jan, and Fua, Pascal (2017). Geometric Graph Matching Using Monte Carlo Tree Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 39, No. 11, pp. 2171–2185. [270]

Pitrat, Jacques (1968). Realization of a General Game-Playing Program. *IFIP Congress*, Vol. 2, pp. 1570–1574. [8]

Powley, Edward J., Whitehouse, Daniel, and Cowling, Peter I. (2013). Bandits All the Way Down: UCB1 as a Simulation Policy in Monte Carlo Tree Search. *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, pp. 81–88, IEEE. [39, 40]

Powley, Edward J., Cowling, Peter I., and Whitehouse, Daniel (2014). Information Capture and Reuse Strategies in Monte Carlo Tree Search, with Applications to Games of Hidden Information. *Artificial Intelligence*, Vol. 217, pp. 92–116. [40]

Qian, Yundi, Haskell, William B., Jiang, Albert Xin, and Tambe, Milind (2014). Online Planning for Optimal Protector Strategies in Resource Conservation Games. *2014 International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pp. 733–740. [271]

Rimmel, Arpad, Teytaud, Fabien, and Teytaud, Olivier (2011). Biasing Monte-Carlo Simulations Through RAVE Values. *Computers and Games* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 59–68, Springer. [37]

Robilliard, Denis, Fonlupt, Cyril, and Teytaud, Fabien (2014). Monte-Carlo Tree Search for the Game of "7 Wonders". *Computer Games* (eds. T. Cazenave, M.H.M. Winands, and Y. Björnsson), Vol. 504 of *CCIS*, pp. 64–77, Springer. [12, 69, 71]

Roelofs, Gijs-Jan (2015). Action Space Representation in Combinatorial Multi-Armed Bandits. M.Sc. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. [124, 128, 130]

Roelofs, Gijs-Jan (2017). Pitfalls and Solutions When Using Monte Carlo Tree Search for Strategy and Tactical Games. *Game AI Pro 3: Collected Wisdom of Game AI Professionals* (ed. S. Rabin), pp. 343–354, CRC Press. [124, 128, 130]

Rohlfshagen, Philipp and Lucas, Simon M. (2011). Ms Pac-Man versus Ghost Team CEC 2011 Competition. *2011 IEEE Congress on Evolutionary Computation (CEC)*, pp. 70–77, IEEE. [7]

Rohlfshagen, Philipp, Liu, Jialin, Perez-Liebana, Diego, and Lucas, Simon M. (2018). Pac-Man Conquers Academia: Two Decades of Research Using a Classic Arcade Game. *IEEE Transactions on Games*, Vol. 10, No. 3, pp. 233–256. [7]

Samothrakis, Spyridon, Robles, David, and Lucas, Simon M. (2010). A UCT Agent for Tron: Initial Investigations. *2010 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 365–371, IEEE. [34]

Samothrakis, Spyridon, Roberts, Samuel A., Perez, Diego, and Lucas, Simon M. (2014). Rolling Horizon Methods for Games with Continuous States and Actions. *2014 IEEE Conference on Computational Intelligence and Games*, pp. 224–231, IEEE. [142]

Samuel, Arthur L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, Vol. 3, No. 3, pp. 210–229. [5]

Sanselone, Maxime, Sanchez, Stéphane, Sanza, Cédric, Panzoli, David, and Duthen, Yves (2014). Constrained Control of Non-Playing Characters Using Monte Carlo Tree Search. *2014 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 208–215, IEEE. [270]

Schaeffer, Jonathan, Lake, Robert, Lu, Paul, and Bryant, Martin (1996). CHINOOK the World Man-Machine Checkers Champion. *AI Magazine*, Vol. 17, No. 1, pp. 21–29. [5]

Schaeffer, Jonathan, Burch, Neil, Björnsson, Yngvi, Kishimoto, Akihiro, Müller, Martin, Lake, Robert, Lu, Paul, and Sutphen, Steve (2007). Checkers is Solved. *Science*, Vol. 317, No. 5844, pp. 1518–1522. [5]

Schaul, Tom (2013). A Video Game Description Language for Model-Based or Interactive Learning. *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, pp. 193–200, IEEE. [9, 61]

Schaul, Tom, Togelius, Julian, and Schmidhuber, Jürgen (2011). Measuring Intelligence Through Games. *arXiv preprint arXiv:1109.1314.* [1]

Schiffel, Stephan (2011). *Knowledge-Based General Game Playing*. Ph.D. thesis, Department of Computer Science, Technische Universität Dresden, Dresden, Germany. [56]

Schiffel, Stephan (2017). Grounding GDL Game Descriptions. *Computer Games* (eds. T. Cazenave, M. Winands, S. Edelkamp, S. Schiffel, M. Thielscher, and J. Togelius), Vol. 705 of *CCIS*, pp. 152–164. Springer. [73]

Schiffel, Stephan and Björnsson, Yngvi (2014). Efficiency of GDL Reasoners. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 6, No. 4, pp. 343–354. [12, 72, 75, 86, 93]

Schiffel, Stephan and Thielscher, Michael (2007). Fluxplayer: A Successful General Game Player. *Twenty-Second AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1191–1196, AAAI Press. [58]

Schkufza, Eric, Love, Nathaniel, and Genesereth, Michael R. (2008). Propositional Automata and Cell Automata: Representational Frameworks for Discrete Dynamic Systems. *AI 2008: Advances in Artificial Intelligence* (eds. W. Wobcke and M. Zhang), Vol. 5360 of *LNAI*, pp. 56–66, Springer. [13, 71, 72, 204]

Schofield, Michael and Thielscher, Michael (2015). Lifting Model Sampling for General Game Playing to Incomplete-Information Models. *Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI)*, pp. 3585–3591, AAAI Press. [210]

Schreiber, Sam (2016). Games - Base Repository. `http://games.ggp.org/base/`. [86, 88, 109, 147, 176]

Schreiber, Sam (2018). Games - Stanford Gamemaster. `http://games.ggp.org/stanford/`. [98]

Schreiber, Sam and Landau, Alex (2016). The General Game Playing Base Package. `https://github.com/ggp-org/ggp-base`. [59, 72, 75, 76]

Segler, Marwin H.S., Preuss, Mike, and Waller, Mark P. (2018). Learning to Plan Chemical Syntheses. *Nature*, Vol. 555, No. 7698, pp. 604–610. [12, 271]

Shafiei, Mohammad, Sturtevant, Nathan R., and Schaeffer, Jonathan (2009). Comparing UCT versus CFR in Simultaneous Games. *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA)* (eds. Y. Björnsson, P. Stone, and M. Thielscher), pp. 75–82. [35]

Shannon, C.E. (1950). Programming a Computer for Playing Chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, Vol. 41, No. 314, pp. 256–275. [5]

Sheppard, Brian (2002). World-Championship-Caliber Scrabble. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 241–275. [11]

Shleyfman, Alexander, Komenda, Antonín, and Domshlak, Carmel (2014). On Combinatorial Actions and CMABs with Linear Side Information. *21st European Conference on Artificial Intelligence (ECAI)*, pp. 825–830, IOS Press. [124, 128, 134]

Silver, David, Huang, Aja, Maddison, Chris J., Guez, Arthur, Sifre, Laurent, Driessche, George van den, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, Dieleman, Sander, Grewe, Dominik, Nham, John, Kalchbrenner, Nal, Sutskever, Ilya, Lillicrap, Timothy, Leach, Madeleine,

Kavukcuoglu, Koray, Graepel, Thore, and Hassabis, Demis (2016). Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, Vol. 529, No. 7587, pp. 484–503. [6]

Silver, David, Schrittwieser, Julian, Simonyan, Karen, Antonoglou, Ioannis, Huang, Aja, Guez, Arthur, Hubert, Thomas, Baker, Lucas, Lai, Matthew, Bolton, Adrian, Chen, Yutian, Lillicrap, Timothy, Hui, Fan, Sifre, Laurent, Driessche, George van den, Graepel, Thore, and Hassabis, Demis (2017). Mastering the Game of Go without Human Knowledge. *Nature*, Vol. 550, No. 7676, pp. 354–359. [6]

Silver, David, Hubert, Thomas, Schrittwieser, Julian, Antonoglou, Ioannis, Lai, Matthew, Guez, Arthur, Lanctot, Marc, Sifre, Laurent, Kumaran, Dharshan, Graepel, Thore, Lillicrap, Timothy, Simonyan, Karen, and Hassabis, Demis (2018). A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go through Self-Play. *Science*, Vol. 362, No. 6419, pp. 1140–1144. [6]

Sironi, Chiara F. and Winands, Mark H.M. (2016). Comparison of Rapid Action Value Estimation Variants for General Game Playing. *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 309–316, IEEE. [17, 103]

Sironi, Chiara F. and Winands, Mark H.M. (2017). Optimizing Propositional Networks. *Computer Games* (eds. T. Cazenave, M. Winands, S. Edelkamp, S. Schiffel, M. Thielscher, and J. Togelius), Vol. 705 of *CCIS*, pp. 133–151. Springer. [51, 71]

Sironi, Chiara F. and Winands, Mark H.M. (2018a). On-Line Parameter Tuning for Monte-Carlo Tree Search in General Game Playing. *Computer Games* (eds. T. Cazenave, M.H.M. Winands, and A. Saffidine), Vol. 818, pp. 75–95, Springer. [123]

Sironi, Chiara F. and Winands, Mark H.M. (2018b). Analysis of Self-Adaptive Monte Carlo Tree Search in General Video Game Playing. *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 397–400, IEEE. [123, 171]

Sironi, Chiara F. and Winands, Mark H.M. (2018c). On-Line Parameter Tuning for Monte-Carlo Tree Search in General Game Playing. *30th Benelux Conference on Artificial Intelligence (BNAIC)*, pp. 235–236. [123]

Sironi, Chiara F. and Winands, Mark H.M. (2019). Comparing Randomization Strategies for Search-Control Parameters in MCTS. *2019 IEEE Conference on Games (COG)*, IEEE. [171]

Sironi, Chiara F., Liu, Jialin, Perez-Liebana, Diego, Gaina, Raluca D., Bravi, Ivan, Lucas, Simon M., and Winands, Mark H.M. (2018). Self-Adaptive MCTS for General Video Game Playing. *Applications of Evolutionary Computation* (eds. K. Sim and P. Kaufmann), Vol. 10784 of *LNCS*, pp. 358–375, Springer. [123]

Sironi, Chiara F., Liu, Jialin, and Winands, Mark H.M. (2019). Self-Adaptive Monte-Carlo Tree Search in General Game Playing. *IEEE Transactions on Games*. In press. [17, 123]

Siwek, Cezary, Kowalski, Jakub, Sironi, Chiara F., and Winands, Mark H.M. (2018). Implementing Propositional Networks on FPGA. *AI 2018: Advances in Artificial Intelligence* (eds. T. Mitrovic, B. Xue, and X. Li), Vol. 11320 of *LNCS*, pp. 133–145, Springer. [71]

Soemers, Dennis J.N.J., Sironi, Chiara F., Schuster, Torsten, and Winands, Mark H.M. (2016). Enhancements for Real-Time Monte-Carlo Tree Search in General Video Game Playing. *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 436–443, IEEE. [13, 17, 37, 40, 47, 51, 66, 67, 123]

Song, Yu and Gong, Shengping (2019). Solar-Sail Trajectory Design for Multiple Near-Earth Asteroid Exploration Based on Deep Neural Networks. *Aerospace Science and Technology*, Vol. 91, pp. 28–40. [271]

Steinhauer, Janik (2010). Monte-Carlo Twixt. M.Sc. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. [47]

Storn, Rainer and Price, Kenneth (1997). Differential Evolution–a Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization*, Vol. 11, No. 4, pp. 341–359. [168, 209]

Sturtevant, Nathan R. (2008). An Analysis of UCT in Multi-Player Games. *ICGA Journal*, Vol. 31, No. 4, pp. 195–208. [87, 110]

Sutton, Richard S. and Barto, Andrew G. (1998). *Reinforcement Learning: An Introduction*. MIT Press. [24]

Świechowski, Maciej and Mańdziuk, Jacek (2014). Self-Adaptation of Playing Strategies in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 6, No. 4, pp. 367–381. [125, 127]

Tak, Mandy J.W., Winands, Mark H.M., and Björnsson, Yngvi (2012). N-Grams and the Last-Good-Reply Policy Applied in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 2, pp. 73–83. [13, 24, 39, 40, 110, 123, 255]

Tak, Mandy J.W., Lanctot, Marc, and Winands, Mark H.M. (2014a). Monte Carlo Tree Search Variants for Simultaneous Move Games. *2014 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 232–239, IEEE. [32, 34, 35, 59, 124]

Tak, Mandy J.W., Winands, Mark H.M., and Björnsson, Yngvi (2014b). Decaying Simulation Strategies. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 6, No. 4, pp. 395–406. [39, 40, 41, 109, 110]

Tesauro, Gerald (1992). Practical Issues in Temporal Difference Learning. *Machine Learning*, Vol. 8, Nos. 3–4, pp. 257–277. [5]

Tesauro, Gerald (1995). Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, Vol. 38, No. 3, pp. 58–68. [5]

Teter, Michael, Buss, Arnold, Darken, Christian, and Baez, Ricardo (2014). Implementation of Monte Carlo Tree Search (MCTS) Algorithm in COMBATXXI using JDAFS. Technical Report TRAC-M-TR-14-031, TRADOC Analysis Center, Monterey, CA. [271]

Teytaud, Fabien and Teytaud, Olivier (2010). Creating an Upper-Confidence-Tree Program for Havannah. *Advances in Computer Games* (eds. H.J. van den Herik and P. Spronck), Vol. 6048 of *LNCS*, pp. 65–74. Springer. [11, 37, 106]

Thielscher, Michael (2011). GDL-II. *KI-Künstliche Intelligenz*, Vol. 25, No. 1, pp. 63–66. [52, 210]

Thielscher, Michael (2017). GDL-III: A Description Language for Epistemic General Game Playing. *Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI)* (ed. C. Sierra), pp. 1276–1282, AAAI Press. [52]

Thompson, Tommy (2018). Revolutionary Warfare | The AI of Total War (Part 3). `https://www.gamasutra.com/blogs/TommyThompson/20180212/314399/Revolutionary_Warfare__The_AI_of_Total_War_Part_3.php`. [270]

Togelius, Julian, Yannakakis, Georgios N., Stanley, Kenneth O., and Browne, Cameron (2011). Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 3, No. 3, pp. 172–186. [6]

Tom, David and Müller, Martin (2011). Computational Experiments with the RAVE Heuristic. *Computers and Games* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 69–80, Springer. [37]

Trunda, Otakar and Barták, Roman (2013). Using Monte Carlo Tree Search to Solve Planning Problems in Transportation Domains. *Advances in Soft Computing and Its Applications* (eds. F. Castro, A. Gelbukh, and M. González), Vol. 8266 of *LNCS*, pp. 435–449, Springer. [12, 271]

Turing, Alan M. (1953). Digital Computers Applied to Games. *Faster Than Thought* (ed. B.V. Bowden), pp. 286–297, Pitman Publishing, London, England. [5]

Van Eyck, Jelle, Ramon, Jan, Guiza, Fabian, Meyfroidt, Geert, Bruynooghe, Maurice, and Berghe, Greet Van den (2013). Guided Monte Carlo Tree Search for Planning in Learned Environments. *Asian Conference on Machine Learning (ACML)* (eds. C.S. Ong and T.B. Ho), Vol. 29 of *JMLR Workshop and Conference Proceedings*, pp. 33–47. [270]

Vinyals, Oriol, Babuschkin, Igor, Chung, Junyoung, Mathieu, Michael, Jaderberg, Max, Czarnecki, Wojciech M., Dudzik, Andrew, Huang, Aja, Georgiev, Petko, Powell, Richard, Ewalds, Timo, Horgan, Dan, Kroiss, Manuel, Danihelka, Ivo, Agapiou, John, Oh, Junhyuk, Dalibard, Valentin, Choi, David, Sifre, Laurent, Sulsky, Yury, Vezhnevets, Sasha, Molloy, James, Cai, Trevor, Budden, David, Paine, Tom, Gulcehre, Caglar, Wang, Ziyu, Pfaff, Tobias, Pohlen, Toby, Wu, Yuhuai, Yogatama, Dani, Cohen, Julia, McKinney, Katrina, Smith, Oliver, Schaul, Tom, Lillicrap, Timothy, Apps, Chris, Kavukcuoglu, Koray, Hassabis, Demis, and Silver, David (2019). AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`. [7]

Wang, Yizao and Gelly, Sylvain (2007). Modifications of UCT and Sequence-like Simulations for Monte-Carlo Go. *2007 IEEE Symposium on Computational Intelligence and Games*, pp. 175–182, IEEE. [36]

Whitehouse, Daniel, Cowling, Peter I., Powley, Edward J., and Rollason, Jeff (2013). Integrating Monte Carlo Tree Search with Knowledge-Based Methods to Create Engaging Play in a Commercial Mobile Game. *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)* (eds. G. Sukthankar and I. Horswill), pp. 100–106, AAAI Press. [269]

Williams, Piers R., Perez-Liebana, Diego, and Lucas, Simon M. (2016). Ms. Pac-Man versus Ghost Team CIG 2016 Competition. *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 420–426, IEEE. [7]

Wimmenauer, Florian (2019). Monte-Carlo Search for Leveraging Performance of Unknown Job Shop Scheduling Heuristics. M.Sc. thesis, Department of Data Science and Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. [191, 271]

Winands, Mark H.M., Björnsson, Yngvi, and Saito, Jahn-Takeshi (2010). Monte Carlo Tree Search in Lines of Action. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 2, No. 4, pp. 239–250. [11, 47]

Yajima, Takayuki, Hashimoto, Tsuyoshi, Matsui, Toshiki, Hashimoto, Junichi, and Spoerer, Kristian (2011). Node-Expansion Operators for the UCT Algorithm. *Computers and Games* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 116–123, Springer. [24]

Yannakakis, Georgios N. and Togelius, Julian (2015). A Panorama of Artificial and Computational Intelligence in Games. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 7, No. 4, pp. 317–335. [1]

Yannakakis, Georgios N. and Togelius, Julian (2018). *Artificial Intelligence and Games*. Springer. [1, 6, 9]

Zech, Philipp, Xiong, Hanchen, and Piater, Justus (2015). Rotation Optimization on the Unit Quaternion Manifold and its Application for Robotic Grasping. *IMA Conference on Mathematics of Robotics. IMA.* [270]

Zhu, George, Lizotte, Dan, and Hoey, Jesse (2014). Scalable Approximate Policies for Markov Decision Process Models of Hospital Elective Admissions. *Artificial Intelligence in Medicine*, Vol. 6, No. 1, pp. 21–34. [12, 270]

Zobrist, Albert L. (1970). A New Hashing Method with Applications for Game Playing. Technical Report 88, Computer Science Department, The University of Wisconsin. Reprinted (1990) in *ICCA Journal*, Vol. 13, No. 2, pp. 69–73. [43]

Appendix A

Example of Game Descriptions

This appendix provides examples of game descriptions for the test domains used in this thesis. First, Section A.1 gives an example of GDL game description for Tic Tac Toe. Subsequently, Section A.2 gives an example of VGDL game description for Zelda.

A.1 GDL Description for Tic Tac Toe

Below is the complete GDL description of the game Tic Tac Toe.[1] Semicolons in GDL are used to identify the entire line as a comment.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Tictactoe
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Roles
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(role xplayer)
(role oplayer)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Base & Input
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(index 1) (index 2) (index 3)
(<= (base (cell ?x ?y b)) (index ?x) (index ?y))
(<= (base (cell ?x ?y x)) (index ?x) (index ?y))
(<= (base (cell ?x ?y o)) (index ?x) (index ?y))
```

[1] Description taken from the Stanford GGP game repository: http://games.ggp.org.

```
(<= (base (control ?r)) (role ?r))

(<= (input ?r (mark ?x ?y)) (index ?x) (index ?y) (role ?r))
(<= (input ?r noop) (role ?r))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Initial State
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (cell 1 1 b))
(init (cell 1 2 b))
(init (cell 1 3 b))
(init (cell 2 1 b))
(init (cell 2 2 b))
(init (cell 2 3 b))
(init (cell 3 1 b))
(init (cell 3 2 b))
(init (cell 3 3 b))
(init (control xplayer))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Dynamic Components
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (cell ?x ?y x))
(does xplayer (mark ?x ?y))
(true (cell ?x ?y b)))

(<= (next (cell ?x ?y o))
(does oplayer (mark ?x ?y))
(true (cell ?x ?y b)))

(<= (next (cell ?x ?y ?w))
(true (cell ?x ?y ?w))
(distinct ?w b))

(<= (next (cell ?x ?y b))
(does ?r (mark ?j ?k))
(true (cell ?x ?y b))
(or (distinct ?x ?j) (distinct ?y ?k)))

(<= (next (control xplayer))
(true (control oplayer)))

(<= (next (control oplayer))
(true (control xplayer)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (row ?x ?w)
(true (cell ?x 1 ?w))
(true (cell ?x 2 ?w))
(true (cell ?x 3 ?w)))

(<= (column ?y ?w)
(true (cell 1 ?y ?w))
(true (cell 2 ?y ?w))
(true (cell 3 ?y ?w)))

(<= (diagonal ?w)
(true (cell 1 1 ?w))
(true (cell 2 2 ?w))
(true (cell 3 3 ?w)))

(<= (diagonal ?w)
(true (cell 1 3 ?w))
(true (cell 2 2 ?w))
(true (cell 3 1 ?w)))

(<= (line ?w) (row ?x ?w))
(<= (line ?w) (column ?y ?w))
(<= (line ?w) (diagonal ?w))

(<= open (true (cell ?x ?y b)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal ?r (mark ?x ?y))
(true (cell ?x ?y b))
(true (control ?r)))

(<= (legal xplayer noop) (true (control oplayer)))

(<= (legal oplayer noop) (true (control xplayer)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (goal xplayer 100) (line x))

(<= (goal xplayer 50)
(not (line x))
(not (line o))
```

```
(not open))

(<= (goal xplayer 0) (line o))

(<= (goal oplayer 100) (line o))

(<= (goal oplayer 50)
(not (line x))
(not (line o))
(not open))

(<= (goal oplayer 0) (line x))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= terminal (line x))

(<= terminal (line o))

(<= terminal (not open))
```

A.2 VGDL Game Description for Zelda

Below is the complete VGDL game description of the game of Zelda.[2] In this game, the player must navigate a maze to find a key. This key opens a door through which the player can exit the maze. Enemies are present in the maze and the player is equipped with a sword to kill them. If the player exits the maze she wins, while if she is hit by an enemy she loses.

```
BasicGame
  SpriteSet
    floor > Immovable randomtiling=0.9 img=oryx/floor3 hidden=True
    goal  > Door color=GREEN img=oryx/doorclosed1
    key   > Immovable color=ORANGE img=oryx/key2
    sword > OrientedFlicker limit=5 singleton=True img=oryx/slash1
    movable >
      avatar  > ShootAvatar stype=sword frameRate=8
        nokey   > img=oryx/swordman1
        withkey > color=ORANGE img=oryx/swordmankey1
      enemy >
        monsterQuick > RandomNPC cooldown=2 cons=6 img=oryx/bat1
        monsterNormal > RandomNPC cooldown=4 cons=8 img=oryx/spider2
        monsterSlow > RandomNPC cooldown=8 cons=12 img=oryx/scorpion1
      wall > Immovable autotiling=true img=oryx/wall3
```

[2]Description taken from the GVG-AI framework: https://github.com/GAIGResearch/GVGAI.

```
LevelMapping
  g > floor goal
  + > floor key
  A > floor nokey
  1 > floor monsterQuick
  2 > floor monsterNormal
  3 > floor monsterSlow
  w > wall
  . > floor

InteractionSet
  movable wall > stepBack
  nokey goal > stepBack
  goal withkey > killSprite scoreChange=1
  enemy sword > killSprite scoreChange=2
  enemy enemy > stepBack
  avatar enemy > killSprite scoreChange=-1
  nokey key > transformTo stype=withkey scoreChange=1 killSecond=True

TerminationSet
  SpriteCounter stype=goal   win=True
  SpriteCounter stype=avatar win=False
```

Appendix B

Game Rules

This appendix provides the characteristics and the rules of all the games used in the experiments presented in this thesis. Rules of the games tested for the Stanford GGP project are given in Section B.1, while rules of the games tested for the GVG-AI project are given in Section B.2.

B.1 Games for the Stanford GGP Project

Below are the rules of the games of the Stanford GGP project that are used in the experiments presented in this thesis. All these games are turn-based, with perfect information and deterministic. Table B.1 reports for each game the number of players, whether they have simultaneous moves and whether they have constant-sum payoffs. Note that payoffs (goals) is GDL are required to be consistently defined only for terminal states, therefore in Table B.1 a game is classified as constant-sum if terminal payoffs have a constant sum. However, for some of the constant-sum games, GDL defines intermediate payoffs that do not have constant sum. Such games are identified with the symbol *.

3D Tic Tac Toe. This game is a variant of *Tic Tac Toe* played on a 4×4×4 cube, where the goal of each player is to align four of her marks in a row (in any direction).

Amazons. This game is played on a 10×10 Chess board. Each player controls four amazons. The players alternate moves, each of which consists of two parts. First, the player moves one of her own amazons in a straight line on the board (vertically, horizontally or diagonally), as a queen moves in chess. Second, the amazon shoots an arrow from its landing square to another square. This arrow can be shot in any orthogonal or diagonal direction, once again like the movement of a queen in Chess. The square where the arrow lands can no longer be used. Both an amazon and an arrow cannot cross or enter a square occupied by either another amazon or a previously shot arrow. The last player to be able to make a move wins.

Table B.1: Characteristics of the games of the Stanford GGP project used for the experiments.

Game	Players	Simultaneous	Constant-Sum
3D Tic Tac Toe	2	×	✓
Amazons	2	×	✓*
Battle	2	✓	×
Breakthrough	2	×	✓*
Checkers	2	×	✓
Chin.Checkers1P	1	×	n/a
Chin.Checkers2P	2	×	×
Chin.Checkers3P	3	×	×
Chin.Checkers4P	4	×	×
Chin.Checkers6P	6	×	×
Chinook	2	✓	×
Connect Four	2	×	✓*
Connect Five	2	×	✓
Horseshoe	2	×	✓*
Joint Connect Four	2	✓	×
Knightthrough	2	×	✓*
Othello	2	×	✓*
Pentago	2	×	✓*
Quad	2	×	✓
Reversi	2	×	✓
Sheep and Wolf	2	×	✓
Skirmish	2	×	×
TTCC4 2P	2	×	✓
TTCC4 3P	3	×	×
Tic Tac Toe	2	×	✓*
Zhadu	2	×	✓

Battle. This game is played on an 8×8 board. Each player has 20 pieces distributed along two adjacent sides of the board, 12 are regular checkers and 8 are kings. Checkers can move one square in any of the orthogonal directions, while kings can move one square in any direction. At each step each player can chose if to move a piece to an empty square, capture an opponent's piece or defend one of her pieces. If a player tries to capture a piece that is being defended, her piece will be captured instead. The player that first captures 10 opponent's pieces wins.

Breakthrough. This game is played on an 8×8 board, where each player has 16 pawns placed in two lines on her side. Each pawn can move one square forward or one square diagonally forward, and can only capture diagonally forward. Players move alternately and the first that reaches the opponent's side with one of her pawns wins.

Checkers. This game is played on an 8×8 board with alternating colored squares. Each player starts with 12 pieces on the black squares of one side of the board. Players alternately move one of their pieces. A piece in a turn can be move diagonally forward by one square or can be used to capture adjacent opponent's pieces by jumping over them diagonally forward. Multiple captures are allowed in a single turn. When a piece reaches the opposite side of the board it is turned into a king, which can also move and capture backwards. The player that is left with no pieces or cannot move loses the game.

Chinese Checkers with 1, 2, 3, 4 or 6 players. This game is played on a hexagram board. Each player has her pieces placed in one corner of the board. The traditional size of the board has space for 10 pieces in each corner, while the GDL descriptions used in this thesis define a smaller board that enables each player to only have three pieces in her corner. Players move alternately and on their turn they can either move one of their pieces to an adjacent empty space on the board or make it perform a chain of jumps over adjacent pieces (either own pieces or of the opponent). The first player that moves all of her pieces in the opposite corner of the board wins.

Chinook. This game is played on an 8×8 board with alternating colored squares and wrapped around vertically to form a cylinder. Chinook consists of two independent games, one played on the even and one on the odd squares of the board. The players start with their pieces on opposite sides of the board. Both players move simultaneously in each turn, alternating in such a way that whenever a player is moving on even squares the other is moving on odd squares. The pieces, like in Checkers, can either move diagonally forward by one square or capture an adjacent opponent's piece by jumping over it in a diagonal direction. Similarly to Breakthrough, the first player that reaches the opposite side of the board with one of her pieces wins.

Connect Four. This game is played on a vertical grid, which is initially empty. Alternating, the players drop one of their pieces from the top of the grid in one of the columns. The first player that aligns four of her pieces horizontally, vertically or diagonally wins the game.

Connect Five. This game is played on an 8×8 board. Two players alternate turns placing one of their pieces on an empty square on the board. The first player that places five of her pieces in a horizontal, vertical or diagonal line wins.

Horseshoe. This game is played on a graph with five nodes. Four nodes are connected to form the shape of a horseshoe, while the remaining node is connected to all other nodes. Each player has two pieces. Initially, the pieces of one player are placed on the top nodes of the horseshoe and the pieces of the other player are placed on the bottom nodes. Players move in turns. During their turn they can move one of their pieces to an empty adjacent node. The goal of a player is to move her pieces to block the opponent, leaving her without legal moves. The game ends when one of the players cannot move anymore.

Joint Connect Four. This game consists in playing two games simultaneously, Connect Four and Connect Four Suicide (the same as Connect Four, but a player loses when four of her pieces are aligned). In each turn one player plays on the Connect Four grid and one on the Connect Four Suicide grid, alternating in subsequent turns. The first player to win any of the two games wins.

Knightthrough. This game is similar to Breakthrough, but is played with chess knights that can only play knight-type moves to advance on the board.

Othello/Reversi. This game is played on an 8×8 board. There are 64 identical pieces called 'disks', which have a dark side and a light side. The game starts with four disks placed in the middle of the board, two of each color connected diagonally. The players place in turns one of the disks on the board, with the color assigned to them facing up. A player can place a disk with her color next to another disk with the opponent's color only if there is another disk of her color somewhere in a straight line (vertically, horizontally or diagonally) from her disk and the opponent's disk, with no empty spaces in between. After placing the disk, all opponent's disks in between the two player's disks are turned over changing the color. If a player has no place where to put her piece, passes the turn. A player wins if all the disks on the board become of her color or if most pieces on the board at the end of the game have her color.

Pentago. This game is played on a 6×6 board, divided into four 3×3 quadrants. Taking turns, the players place a mark of their color on an empty square of the board and then rotate one of the quadrants by 90 degrees, either clockwise or counterclockwise. The first player that aligns vertically, horizontally or diagonally five of her marks, either before or after performing a rotation in her turn, wins the game. If all the squares of the board are marked and no line of five marks is formed, the game ends in a draw.

Quad. The version of Quad used in this thesis is played on a 7×7 board with square tiles. Each player has 12 pieces of her color, the quads, and 5 white pieces. In her turn, a player can place on the empty tiles of the board any number of her white pieces and one of her quads. After a player has placed a quad, her turn ends. The first player that forms a square with four of her quads as vertexes is the winner. A square might be formed also with edges that are not parallel to the edges of the board. White pieces on the board cannot be used to form a square, but are useful to block the opponent. There is also the possibility that all quads are played, but no square is formed. In this situation, if the player who was on the move is only missing a quad to win, she is declared the winner. Otherwise, the one who has the most white pieces left wins. If both players have the same number of white pieces left the game is a draw.

Sheep and Wolf. This game is played on an 8×8 board. One player controls four sheep, which are placed on the left side of the board and can only move diagonally forward, and the other player controls the wolf, which is placed on the right side of the board and can move diagonally both backwards and

forward. The game ends when either of the two players is left with no legal moves or when the wolf is behind all the sheep. The wolf-player wins if at the end of the game she can still move, otherwise the sheep-player wins.

Skirmish. This game is a variant of Chess played on an 8×8 board in which the goal of each player is to capture as many pieces of the opponent as possible. The game ends when one player has no remaining pieces on the board. The final score of a player is a linear function of the number of opponent pieces captured, with every piece worth approximately the same amount.

Tic-Tac-Chess-Checkers-Four (TTCC4) with 2 or 3 players. This game takes elements from the games of Tic Tac Toe, Chess, Checkers and Connect Four. It is played on a 5×5 board (plus the starting cells for the pieces). Each player has three pieces that she can move on the board, a Chess pawn, a Checkers king and a Chess knight. The movement of these pieces follows the rules of the game they are taken from and according to these rules they can also capture opponent's pieces. During her turn, a player can choose whether to move one of her pieces or drop a disc in one of the three central columns of the board. The winner is the player that first manages to align three of her pieces in the central 3×3 squares of the board.

Tic Tac Toe. This game played on a 3×3 board. The players alternate marking one of the empty cells on the board with their symbol. The first player that aligns three of its symbols vertically, horizontally or diagonally wins the game. If all cells are marked but no line is formed, the game ends in a draw.

Zhadu. This game is played on a rhombus-shaped board divided into eight equilateral triangles. Each player has five pieces with values 1, 2, 3, (1,2,3) and 4, respectively. Each piece can move as many steps as specified by its value(s). Each triangle on the board has four spaces that the pieces can occupy, one in each corner and one in the center. For moving purposes, the center of a triangle is adjacent to that triangle's corners and a triangle's corners are adjacent to the corners of neighboring triangles. In the first phase of the game, players alternately place one of their pieces in one of the allowed locations on the board. In the second phase of the game, they alternately move one of their pieces. Pieces cannot jump over other pieces, but can capture a piece when moving to its location. The first captured piece determines the win condition for the player. A player wins when the values of her first and last captured pieces sums up to 4.

B.2 Games for the GVG-AI Project

Below are the rules of the games of the GVG-AI project that are used in the experiments presented in this thesis. All these games are real-time, single-player and with perfect information. Table B.2 reports for each game whether it is deterministic or not.

Table B.2: Characteristics of the games of the GVG-AI project used for the experiments.

Game	Deterministic
Aliens	×
Bait	✓
Butterflies	×
Camel Race	×
Chase	×
Chopper	×
Crossfire	×
Dig Dug	×
Escape	✓
Hungry Birds	✓
Infection	×
Intersection	×
Lemmings	×
Missile Command	×
Modality	✓
Plaque Attack	×
Roguelike	×
Sea Quest	×
Survive Zombies	×
Wait For Breakfast	×

Aliens. This game is similar to Space Invaders. The player in the bottom of the screen shoots upwards at aliens that approach Earth. Aliens can shoot bombs back at the avatar. The player loses if any alien touches her or hits her with a bomb, and wins if all aliens are eliminated.

Bait. The objective of this game is to reach the door, collecting a key first. The player can push boxes around to open paths. There are holes in the ground that kill the player, but they can be filled with boxes (and both hole and box disappear). The player can also collect mushrooms that give points.

Butterflies. The player must capture butterflies that move randomly around the level. If a butterfly touches a cocoon, more butterflies are spawned. The player wins if she collects all butterflies, but loses if all cocoons are opened.

Camel Race. The player must get to the finish line before any other camel does. Depending on the level, there are some camels that proceed toward the goal at a fixed speed and some that move around randomly.

Chase. The player must chase and kill scared goats that flee from her. If a goat finds another goat's corpse, it becomes angry and chases the player. The player wins if all scared goats are dead, but loses if is hit by an angry goat.

Chopper. The objective is to avoid the tanks on the ground to destroy all satellites floating in space. The tanks shoot missiles to the satellites, although they are stopped if they hit a cloud in between. The player flies a helicopter at the clouds levels, which can shoot at the tanks to destroy them.

Crossfire. The objective is to reach the exit of the level by avoiding the shots (with random direction) of the multiple turrets in the level.

Dig Dug. The player must collect all gems and gold coins in the cave, digging its way through it. There are also enemies in the level that kill the player on collision with him. The player can kill the enemies using boulders, which can be shot at them by performing the USE action for two consecutive time steps.

Escape. The objective is to leave the level through the exit door, pushing away boxes that are in the way. This boxes can be destroyed by pushing them into holes, but those holes also kill the player if the avatar falls into them.

Hungry Birds. The avatar is a bird that becomes hungrier at any game tick. It needs to exit a maze before dying of hunger, and while looking for the exit it can find food in the maze to reduce its hunger.

Infection. The objective of the game is to infect all healthy people with a virus. The player can get infected by colliding with sources of the virus scattered around the level, or with other people that are infected. Medics cure infected people and the avatar, and can be killed by the avatar with a sword.

Intersection. The goal of this game is to collect as many delivery items are possible. These items are spawned, one by one, at different places in the level. In order to reach them, the avatar must cross a road that is traveled by fast cars. Every time the avatar is hit by a car, it loses a life. The avatar starts with 5 lives, and the game is lost when this number gets to 0.

Lemmings. Lemmings are spawned from a door and try to get to the exit of the level. The player must destroy walls in the level so the lemmings can reach the exit. There are traps as well that kill the lemmings and the player if they fall into them. Score is given for every lemming that reaches the exit, but subtracted from every piece of wall destroyed, hence the game rewards players that do less digging.

Missile Command. The avatar must shoot at several missiles that fall from the sky before they reach the cities they are directed towards. The player wins if she is able to save at least one city, and loses if all cities are hit.

Modality. The goal is to push a crate into a hole. The avatar can walk over two types of surfaces, and there is a unique point where the player can move from one to the other surface. The box can cross surfaces freely.

Plaque Attack. Hamburgers and hotdogs are attacking the teeth. The player must shoot them in order to save at least one tooth. Damaged teeth can be repaired by the player upon contact. When all food items are destroyed, the player wins. If all teeth are destroyed, the player loses.

Roguelike. In this game, the objective is to find the exit and escape the maze through it. There are monsters in the maze that can be killed with a sword that can be picked up. Doors can be opened with collectible keys and gems and gold are available to be looted. There is also a market where keys can be exchanged to recover health.

Sea Quest. The player controls a submarine that must avoid being killed by animals and rescue divers taking them to the surface. Also, the submarine must return to the surface regularly to collect more oxygen, or the player would lose. Submarine capacity is for four divers, and it can shoot torpedoes to the animals.

Survive Zombies. The avatar must stay alive while being attacked by spawned zombies. It may collect honey, dropped by bees, in order to avoid being killed by zombies. The player wins if the timer runs out, and loses if hit by a zombie while having no honey (otherwise, the zombie dies).

Wait For Breakfast. The objective of the game is to eat breakfast. For that, the player must wait until the waiter comes and leaves the breakfast on a table. If the avatar gets to the table before the food has arrived, it must wait at the chair, or lose the game if it leaves the table.

Appendix C

Supplementary Results for Chapter 4

This appendix reports supplementary results for Chapter 4 with the purpose of providing a more detailed overview of the performance of the Prover and the optimized PropNet (Opt1023) when a cache is used to memorize results of previous queries to the reasoners.

To better understand how the cache influences the speed over time for the tested games, the speed of the reasoners has been further analyzed over the game turns. More precisely, for each turn in a game the average speed over all the game runs has been plotted. Figures C.1 and C.2 show such plots for each of the tested 13 games for the Prover, both without and with cache. Figures C.3 and C.4 show the same plots for the PropNet, without and with cache. Note that for some games the difference between the cached and non-cached version of a reasoner is quite high. Therefore, to improve readability, some of the plots use a base-10 logarithmic scale for the speed. Whether the logarithmic scale is used is indicated in the y-axis label of each plot. Looking at the plots for the Prover, the cache seems to always be beneficial. In Chinese Checkers with 1 player and Tic Tac Toe the cache substantially increases the speed already from the first turns. For all other games, in the first turns the Prover with the cache seems to have a similar speed to the Prover without the cache, but it increases the speed in later turns. For some games, like most of the Chinese Checkers variants and Connect Four, the cache starts having an impact already in the middle game, therefore being more promising than in games such as Battle, Othello or Skirmish, where its effect is only visible in the endgame.

Looking at the plots for the PropNet, once again the cache seems beneficial already from the first turns for Chinese Checkers with 1 player and Tic Tac Toe. However, in many games the cache is actually decreasing the speed of the PropNet reasoner during the initial turns. This loss is then balanced towards the endgame, when the chance of finding cached query results increases. For the PropNet it takes more time for the cache to be filled with a sufficient number of entries to outperform the speed with which the PropNet computes queries. The same effect was not observed for the Prover because the time for computing the answer of a query with

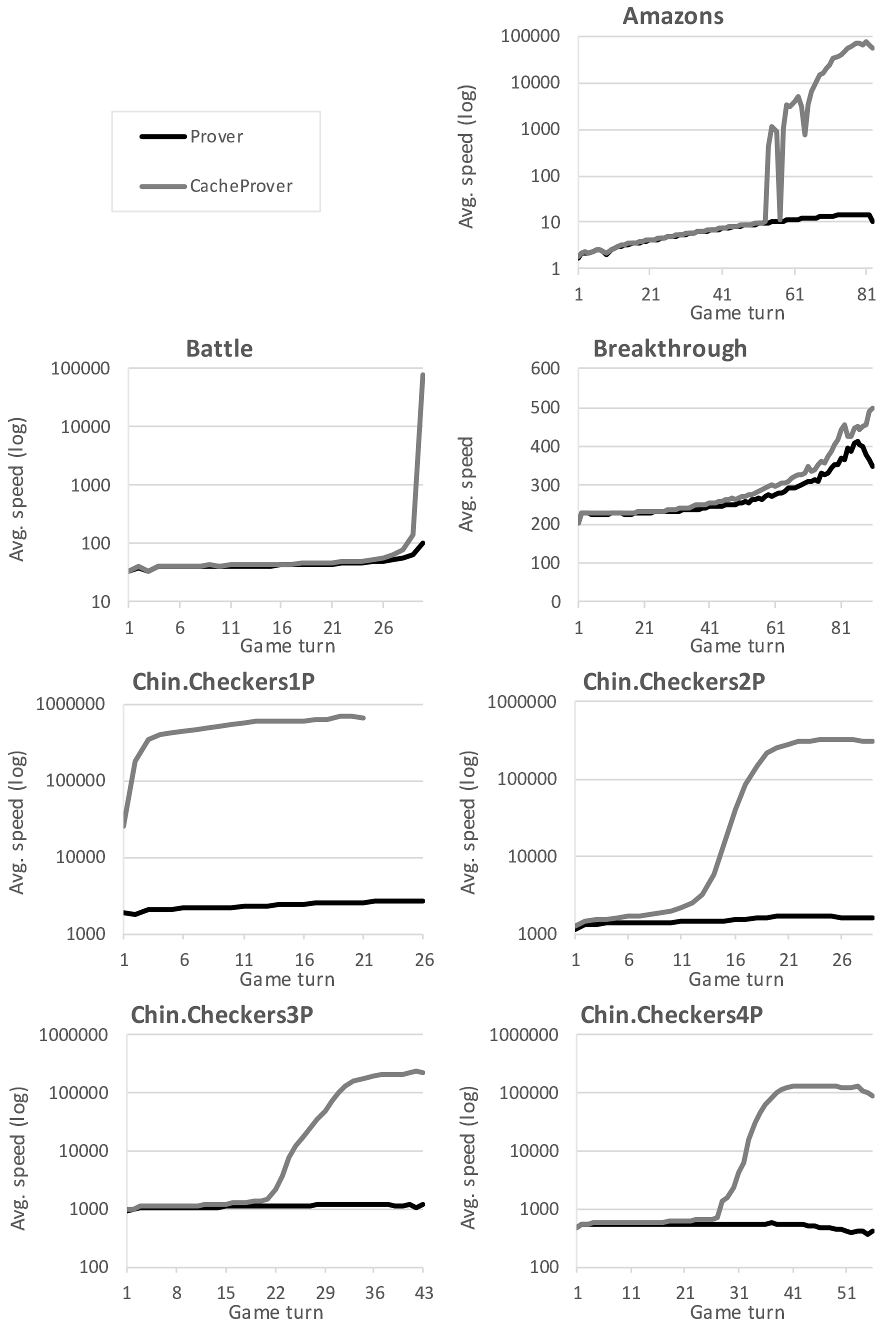


Figure C.1: Speed of the Prover without and with cache over different game turns for the games of Amazons, Battle, Breakthrough, and Chinese Checkers with 1, 2, 3 and 4 players.

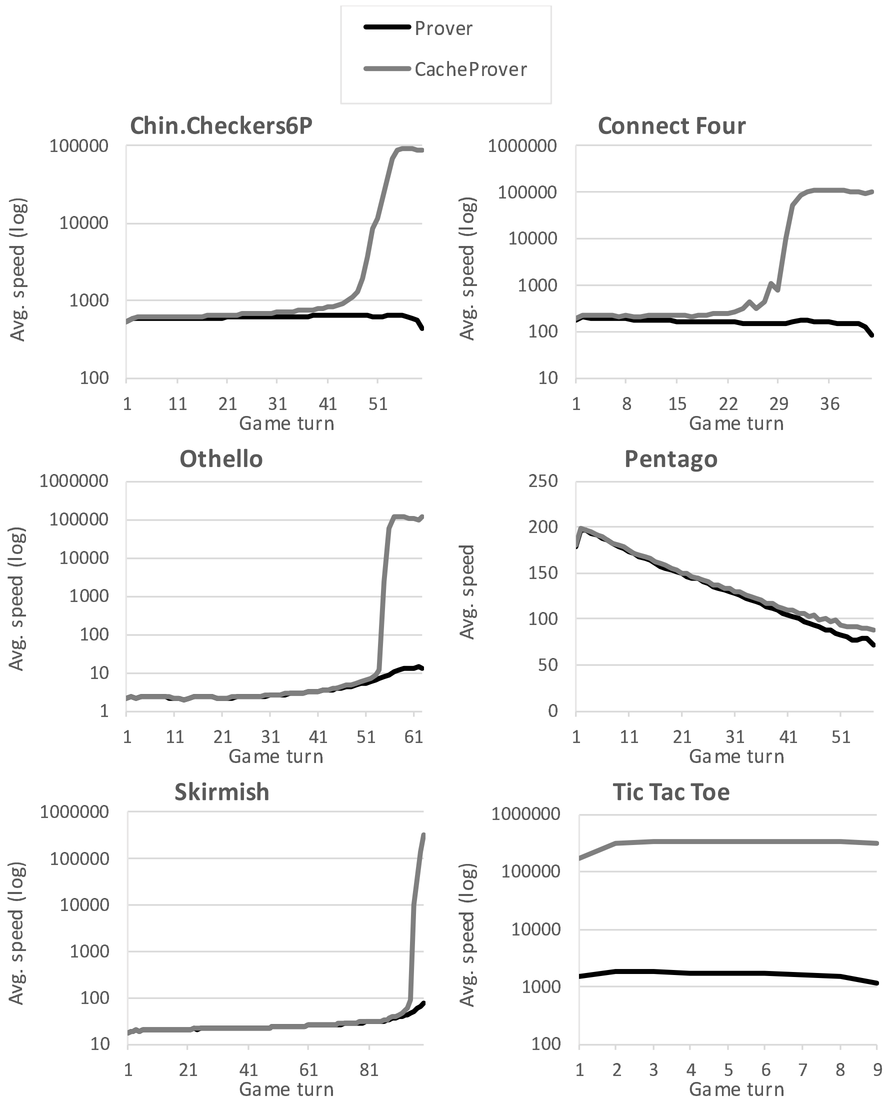


Figure C.2: Speed of the Prover without and with cache over different game turns for the games of Chinese Checkers with 6 players, Connect Four, Othello, Pentago, Skirmish, and Tic Tac Toe.

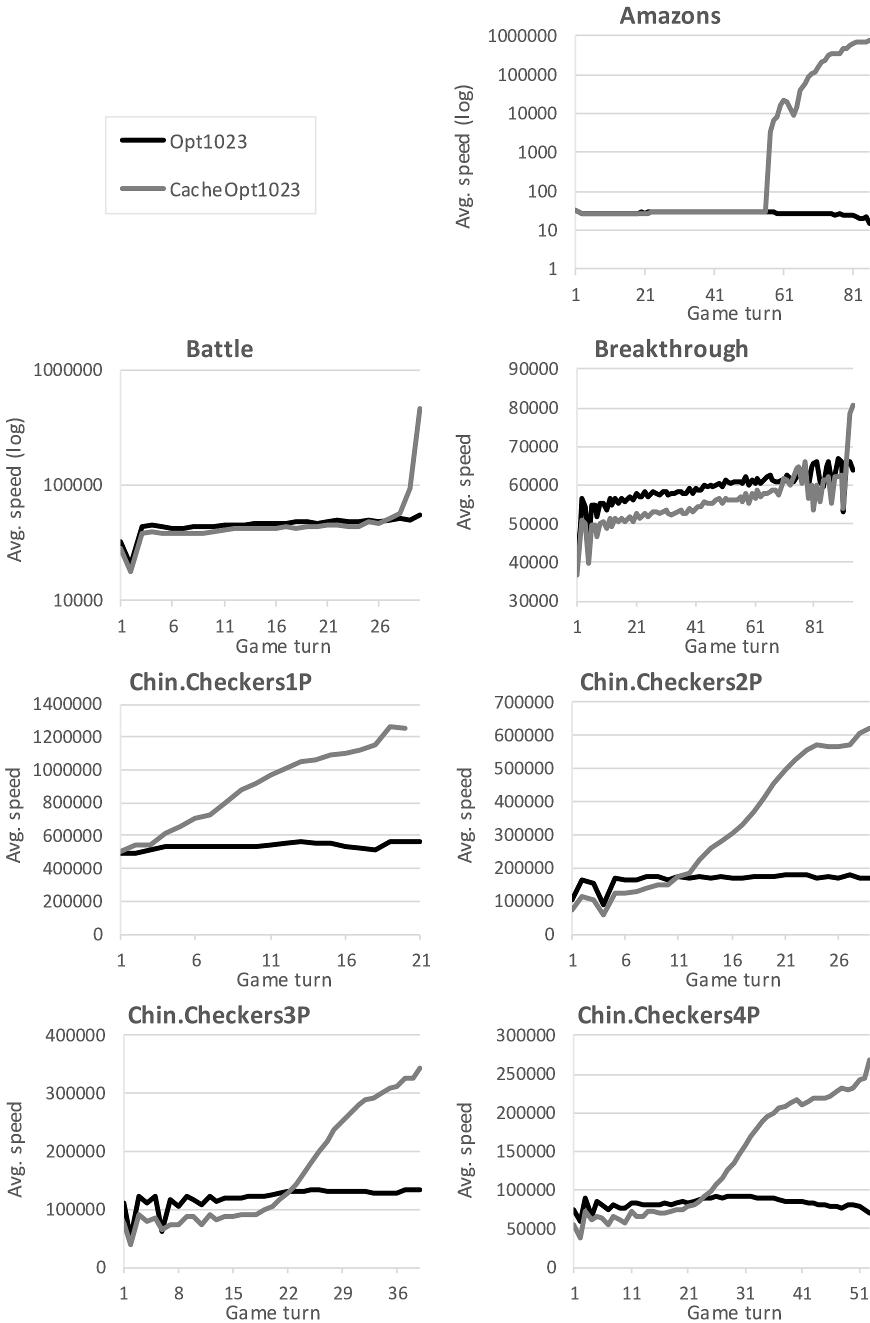


Figure C.3: Speed of the optimized PropNet without and with cache over different game turns for the games of Amazons, Battle, Breakthrough, and Chinese Checkers with 1, 2, 3 and 4 players.

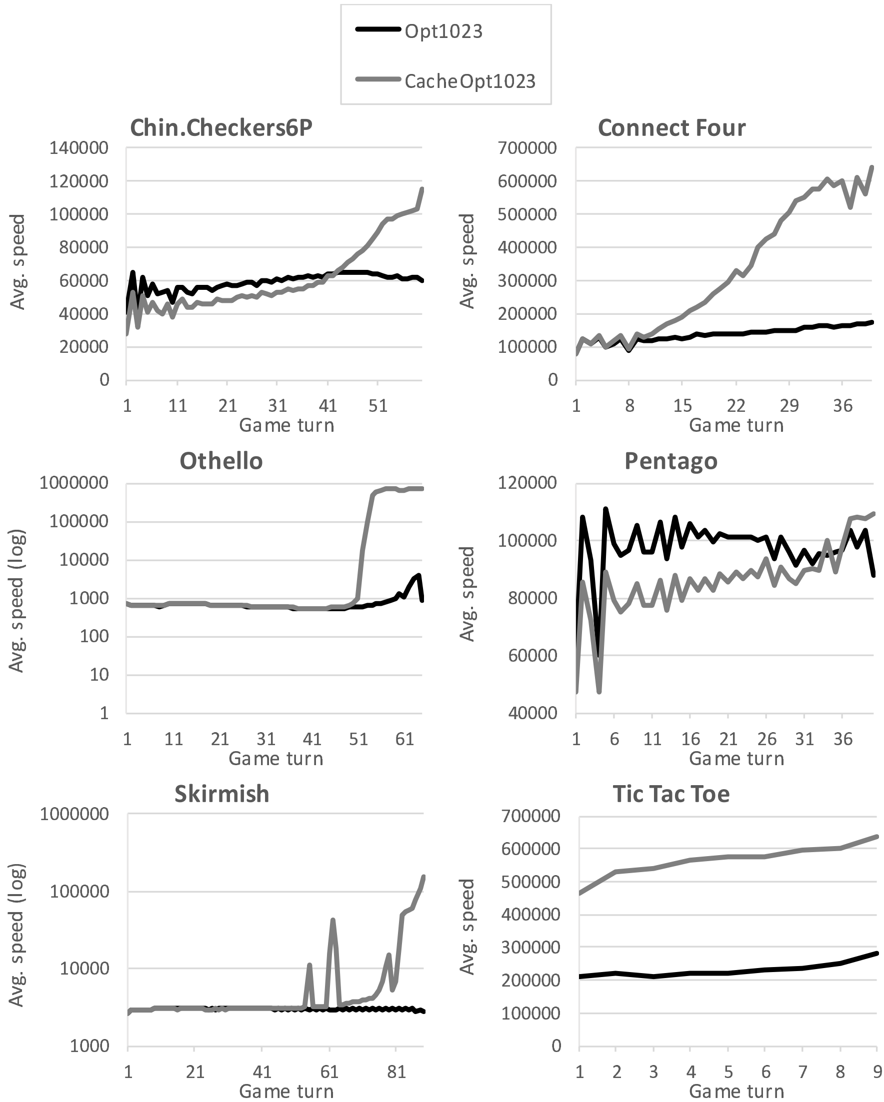


Figure C.4: Speed of the optimized PropNet without and with cache over different game turns for the games of Chinese Checkers with 6 players, Connect Four, Othello, Pentago, Skirmish, and Tic Tac Toe.

the Prover is in general much higher than the one of the PropNet reasoner. Thus, for the Prover finding in the cache even a small number of query results saves enough computational time to compensate the extra time spent looking in the cache for results that are not present yet.

Appendix D

Supplementary Results for Chapter 5

This appendix reports supplementary results for Chapter 5 with the purpose of providing a comparison of the performance of the MAST play-out strategy against the random play-out strategy. It has been shown in the literature that, when used in MCTS together with the UCT selection strategy and tested on a set of games taken from the Stanford GGP project, MAST significantly outperforms a random play-out strategy in most of them (Finnsson, 2012b; Tak *et al.*, 2012). In this appendix,

Table D.1: Win percentage of $\mathrm{P}_{\text{UCT-MAST}}$ against P_{UCT} with 1s play-clock and start-clock.

Game	$\mathbf{P}_{\textbf{UCT-MAST}}$
3D Tic Tac Toe	80.1(±3.48)
Breakthrough	82.6(±3.33)
Knightthrough	89.0(±2.75)
Skirmish	44.6(±3.63)
Battle	22.4(±3.49)
Chinook	64.8(±3.13)
Chin.Checkers3P	68.1(±4.07)
Checkers	73.4(±3.60)
Connect Five	87.2(±2.66)
Othello	61.4(±4.21)
Quad	80.1(±3.44)
Sheep and Wolf	44.4(±4.36)
TTCC4 2P	86.5(±2.98)
Zhadu	52.4(±4.38)
TTCC4 3P	58.0(±4.22)
Avg. Win%	66.3(±1.03)

MAST and the random play-out strategy are compared on the same 15 games used in Chapter 5 and with the same experimental settings. More precisely, the agents that are used as baselines in the chapter, P_{UCT} and $P_{\text{UCT-MAST}}$, are matched against each other with 1s start- and play-clock. For this experiment, results are based on at least 500 runs for each game. These experiments were performed on a Linux server consisting of 64 AMD Opteron 6274 2.2-GHz cores. Table D.1 reports the win percentage for the $P_{\text{UCT-MAST}}$ agent. These results confirm that MAST generally outperforms the random play-out strategy. In most of the tested games the win percentage of $P_{\text{UCT-MAST}}$ is significantly higher than the one of P_{UCT}. Only in Battle, Skirmish and Sheep and Wolf P_{UCT} is significantly better.

Appendix E

Supplementary Results for Chapter 6

This appendix reports supplementary results for Chapter 5 that were obtained when tuning different settings for the following allocation strategies: MAB, HE, NMC, LSI, EA and NTBEA. All the experiments are performed for the Stanford GGP project, on the same games used in 6. The AP agent as described in Subsection 6.4.1 is used as baseline, and the strategies are tested for four tuned parameters, C, ϵ_{MAST}, K and ref. All the settings of the strategies are the same as specified in Subsection 6.4.1, unless stated otherwise. Start- and play-clock are both set to $1s$.

MAB. The MAB allocation strategy presents a high overhead caused by having to compute the UCB1 value of all the parameter combinations before performing each MCTS evaluation. Using a batch of simulations to evaluate a parameter combination instead of using only a single simulation might alleviate this problem. In this way, MAB would have to compute the value of all combinations only every few simulations. Table E.1 shows the result of testing the MAB strategy with a batch size of 1 (i.e. each parameter combination is evaluated by a single MCTS simulation) and a batch size of 10. The positive effect of using a batch of simulations is visible for most of the games, although the performance is still worse than the one of AP with fixed parameter settings. A batch of 10 simulations is used in the experiments in chapter 6.

HE. When using HE, the combinatorial parameter space is represented as a tree, where each level corresponds to a different tunable parameter. To build this tree, an order must be imposed on the parameters. Randomizing this order before each time a new game is played might be a robust choice in domains like GGP, where the optimal order might vary depending on the game that is being played. The purpose of the experiments for which Table E.2 reports the results is to verify whether imposing a predefined order on the parameters would be a better choice than randomizing it. The predefined order is the following: ϵ_{MAST}, K, ref, C. Parameters have been ordered from the one that seems to have in general the highest influence on the search to the one

that seems to have the least influence on the search. As can be seen, in none of the games the agent using the predefined order is significantly better than the one that randomizes it, therefore, the random order is used in the experiments performed in Chapter 6.

NMC. To select parameter combinations from the local and global MABs, NMC uses the UCB1 strategy, with exploration constants C_l and C_g, respectively. An experiment to test a few combinations of values for these parameters has been performed. For C_l, only high values are tested, because the exploration of parameter values during the exploration phase of NMC should be beneficial. For C_g, instead, it could be argued that smaller values could be better. In NMC, a combination is in the global MAB only if it has been generated using the local MABs first. Over time, many combinations in the global MAB should contain at least a few single values that have shown a generally good performance. Therefore, many of such combinations could likely be good as well. Increasing exploitation of such combinations using a low value for C_g might be a good strategy. Results of these experiments are reported in Table E.3. For C_l, there is not a significant difference between the values 0.7 and 1.0, independently from the value of C_g. Therefore, for the experiments in Chapter 6 more exploration is preferred, setting $C_l = 0.1$. More interesting are the results for C_g, which contradict the intuition that less exploration in the global MAB should be beneficial. This suggests that for the tuning problem more exploration should be preferred, because the best parameter combinations might change over time. Thus, in the experiments performed in Chapter 6 C_g is also set to 1.0.

LSI. The LSI allocation strategy requires to know in advance the number of samples (i.e. MCTS simulations) that are available to tune the parameters, N_{tot}. However, this value can only be estimated approximately. In this thesis, this is done during the start-clock by estimating how many simulations per second can be performed, and then multiplying them by the duration of the play-clock and by the number of expected turns for the game. The number of simulations per second performed in the start-clock are generally less than the average over all the game, therefore they are multiplied by a factor κ. Three different values for this parameter have been tested to see which one increases the win rate of LSI the most. Results are presented in Table E.4. There does not seem to be a significant difference among the three values. However, $\kappa = 3$ is never significantly worse than the other two values in any of the games. Moreover, in Quad and Pentago it is significantly better than $\kappa = 2$, therefore it has been selected as default value for the experiments in Chapter 6.

EA. An important parameter that influences the performance of the EA allocation strategy is the size of the elite μ, i.e. the number of combinations in the total population that are used to generate a new population. Keeping too many combinations might slow down the evolutionary process, while keeping too few might prevent some good combinations from surviving over multiple generations. Table E.5 shows the results obtained by testing three different values

for the elite size μ. Note that this experiment has been performed using an earlier implementation of EA than the one used in Chapter 6. The settings are the same, but the earlier implementation is resetting all the statistics collected for the elite individuals each time a new population is generated. Results show that for many games a high elite size is better. This is particularly visible for Knightthrough and Breakthrough. Therefore, $\mu = 25$ is used in the experiments of Chapter 6.

NTBEA. An interesting parameter to test for the NTBEA allocation strategy is the number of neighbors, X, of the current individual that are generated in each iteration. With more neighbors there are more chances to find an individual with a better value than the current one, but at the same time more overhead is introduced to compute the UCB1 value of each of them. Also interesting is to test whether considering n-tuples with intermediate lengths improves the performance over considering only 1-tuples and d-tuples, i.e. with the maximum length d. Results of experiments that test values for X are reported in Table E.6, while results for experiments that compare the use of different lengths for the n-tuples are shown in Table E.7. Note that these experiments have been performed using an earlier implementation of NTBEA than the one used in Chapter 6. All settings are the same, except the exploration constant C_{NTBEA} that is set to 0.7. Results show a decrease in performance with the increase in number of generated neighbors X, therefore the smallest value is used in Chapter 6, $X = 5$. For the lengths of the n-tuples, results show that there is not much difference in performance between using all lengths or only 1 and 4. In order to reduce the overhead of estimating the UCB1 value, in Chapter 6 only 1- and d-tuples are considered.

Table E.1: Win percentage of the AP agent that tunes four parameters on-line with the MAB allocation strategy with or without using a batch of simulations to evaluate each parameter combination.

Game	**AP_{MAB}**	
	Batch size = 1	***Batch size = 10***
3D Tic Tac Toe	6.2(±2.00)	20.8(±3.38)
Breakthrough	2.2(±1.29)	8.0(±2.38)
Knightthrough	7.4(±2.30)	20.0(±3.51)
Chinook	11.3(±2.58)	21.5(±3.31)
Chin.Checkers3P	32.1(±4.08)	38.7(±4.26)
Checkers	8.3(±2.25)	8.3(±2.23)
Connect Five	12.1(±2.25)	13.8(±2.29)
Quad	25.3(±3.71)	46.8(±4.23)
Sheep and Wolf	34.0(±4.16)	42.6(±4.34)
TTCC4 2P	17.5(±3.28)	22.4(±3.63)
TTCC4 3P	40.8(±4.24)	43.9(±4.25)
Connect Four	29.3(±3.84)	42.0(±4.14)
Pentago	16.1(±3.18)	26.1(±3.75)
Reversi	31.9(±4.04)	31.1(±4.00)
Avg. Win%	19.6(±0.91)	27.6(±1.01)

Table E.2: Win percentage of the AP agent that tunes four parameters on-line with the MAB allocation strategy with or without using a batch of simulations to evaluate each parameter combination.

Game	**AP_{HE}**	
	Random order	**Predefined order**
3D Tic Tac Toe	34.5(±3.97)	38.8(±4.08)
Breakthrough	53.6(±4.38)	50.2(±4.39)
Knightthrough	69.8(±4.03)	68.6(±4.07)
Chinook	30.8(±3.70)	27.7(±3.47)
Chin.Checkers3P	34.9(±4.17)	37.3(±4.23)
Checkers	33.6(±3.92)	35.6(±3.96)
Connect Five	25.9(±3.01)	24.1(±2.90)
Quad	29.9(±3.76)	33.2(±3.89)
Sheep and Wolf	43.2(±4.35)	46.4(±4.38)
TTCC4 2P	38.6(±4.15)	37.5(±4.18)
TTCC4 3P	40.6(±4.14)	44.2(±4.21)
Connect Four	37.0(±4.09)	41.9(±4.15)
Pentago	40.6(±4.14)	42.0(±4.07)
Reversi	41.5(±4.26)	42.7(±4.26)
Avg. Win%	39.6(±1.10)	40.7(±1.10)

Table E.3: Win percentage of the AP agent that tunes four parameters on-line with the NMC allocation strategy for different combinations of values for C_g and C_l.

	Game	$\mathbf{AP_{NMC}}$			
		$C_g = 0.0$	$C_g = 0.4$	$C_g = 0.7$	$C_g = 1.0$
$C_l = 0.7$	3D Tic Tac Toe	32.8(±3.97)	37.5(±4.02)	45.4(±4.17)	40.5(±4.09)
	Breakthrough	49.6(±4.39)	56.2(±4.35)	59.4(±4.31)	56.8(±4.35)
	Knightthrough	65.2(±4.18)	68.6(±4.07)	70.0(±4.02)	73.4(±3.88)
	Chinook	27.3(±3.54)	30.7(±3.71)	36.0(±3.89)	36.3(±3.76)
	Chin.Checkers3P	31.7(±4.07)	36.5(±4.21)	33.9(±4.14)	37.1(±4.22)
	Checkers	31.1(±3.83)	36.2(±3.96)	35.4(±3.94)	38.2(±4.05)
	Connect Five	30.1(±3.03)	32.3(±3.14)	30.0(±3.03)	29.3(±3.00)
	Quad	31.6(±3.79)	35.6(±4.00)	39.1(±4.04)	36.2(±4.04)
	Sheep and Wolf	42.6(±4.34)	44.4(±4.36)	42.8(±4.34)	44.0(±4.36)
	TTCC4 2P	30.9(±3.97)	38.6(±4.15)	43.2(±4.22)	44.8(±4.24)
	TTCC4 3P	38.3(±4.08)	43.9(±4.22)	41.3(±4.14)	42.2(±4.20)
	Connect Four	34.0(±3.98)	39.1(±4.12)	41.3(±4.14)	39.6(±4.15)
	Pentago	34.1(±3.93)	42.5(±4.06)	42.6(±4.17)	42.8(±4.15)
	Reversi	40.3(±4.25)	40.7(±4.22)	36.7(±4.17)	41.5(±4.25)
	Avg. Win%	37.1(±1.08)	41.6(±1.11)	42.6(±1.11)	43.0(±1.11)
$C_l = 1.0$	3D Tic Tac Toe	34.6(±3.97)	40.1(±4.04)	41.8(±4.08)	39.8(±4.05)
	Breakthrough	51.2(±4.39)	55.8(±4.36)	63.4(±4.23)	60.6(±4.29)
	Knightthrough	64.8(±4.19)	67.6(±4.11)	67.4(±4.11)	74.2(±3.84)
	Chinook	27.4(±3.46)	33.5(±3.69)	36.5(±3.84)	36.7(±3.75)
	Chin.Checkers3P	32.3(±4.09)	33.3(±4.12)	35.3(±4.18)	36.9(±4.22)
	Checkers	34.3(±3.94)	34.9(±3.91)	36.2(±3.90)	37.8(±3.91)
	Connect Five	29.7(±3.03)	31.1(±3.12)	28.8(±3.00)	30.7(±3.11)
	Quad	30.4(±3.84)	36.8(±4.02)	38.2(±4.05)	37.9(±4.04)
	Sheep and Wolf	47.4(±4.38)	45.8(±4.37)	44.2(±4.36)	45.2(±4.37)
	TTCC4 2P	36.6(±4.10)	42.2(±4.19)	45.2(±4.25)	44.6(±4.25)
	TTCC4 3P	40.2(±4.15)	42.2(±4.21)	42.6(±4.17)	40.4(±4.17)
	Connect Four	40.5(±4.13)	38.4(±4.09)	38.6(±4.02)	42.9(±4.15)
	Pentago	35.4(±4.00)	43.4(±4.20)	44.4(±4.19)	38.8(±4.07)
	Reversi	39.1(±4.19)	36.5(±4.14)	38.9(±4.18)	39.8(±4.24)
	Avg. Win%	38.8(±1.09)	41.5(±1.10)	43.0(±1.11)	43.3(±1.11)

Table E.4: Win percentage of the AP agent that tunes four parameters on-line with the LSI allocation strategy, for different values of the factor κ.

Game	$\mathbf{AP_{LSI}}$		
	$\kappa = 2$	$\kappa = 3$	$\kappa = 4$
3D Tic Tac Toe	42.1(±4.08)	42.3(±4.11)	41.3(±4.09)
Breakthrough	39.0(±4.28)	37.2(±4.24)	32.6(±4.11)
Knightthrough	59.8(±4.30)	53.8(±4.37)	50.6(±4.39)
Chinook	24.1(±3.45)	24.8(±3.55)	22.0(±3.41)
Chin.Checkers3P	34.1(±4.14)	36.1(±4.20)	36.5(±4.21)
Checkers	17.0(±3.12)	19.7(±3.28)	18.7(±3.23)
Connect Five	38.4(±3.30)	40.3(±3.27)	40.3(±3.10)
Quad	65.3(±3.96)	75.6(±3.56)	71.8(±3.72)
Sheep and Wolf	50.6(±4.39)	49.0(±4.39)	48.4(±4.38)
TTCC4 2P	25.3(±3.74)	22.6(±3.60)	23.1(±3.63)
TTCC4 3P	46.6(±4.25)	47.3(±4.24)	48.2(±4.28)
Connect Four	56.8(±4.14)	59.1(±4.18)	53.9(±4.19)
Pentago	40.1(±4.18)	48.8(±4.22)	46.3(±4.20)
Reversi	31.8(±4.00)	27.1(±3.83)	25.3(±3.73)
Avg. Win%	40.8(±1.11)	41.7(±1.11)	39.9(±1.10)

Table E.5: Win percentage of the AP agent that tunes four parameters on-line with the EA allocation strategy, for different values of the elite size μ.

Game	$\mathbf{AP_{EA}}$		
	$\mu = 5$	$\mu = 10$	$\mu = 25$
3D Tic Tac Toe	38.1(±4.06)	38.3(±4.01)	41.2(±4.09)
Breakthrough	39.6(±4.29)	48.2(±4.38)	58.6(±4.32)
Knightthrough	49.0(±4.39)	61.4(±4.27)	69.0(±4.06)
Chinook	48.6(±4.04)	46.5(±4.07)	49.8(±4.08)
Chin.Checkers3P	41.4(±4.30)	43.9(±4.33)	41.5(±4.31)
Checkers	27.6(±3.64)	31.9(±3.78)	37.3(±4.01)
Connect Five	28.9(±3.00)	27.9(±2.80)	25.7(±2.92)
Quad	46.8(±4.16)	43.4(±4.12)	38.7(±4.08)
Sheep and Wolf	48.8(±4.39)	47.0(±4.38)	49.4(±4.39)
TTCC4 2P	46.4(±4.30)	45.4(±4.22)	47.9(±4.24)
TTCC4 3P	48.1(±4.25)	44.3(±4.22)	43.3(±4.21)
Connect Four	50.7(±4.23)	49.3(±4.20)	52.1(±4.23)
Pentago	42.5(±4.17)	39.0(±4.11)	45.5(±4.18)
Reversi	37.5(±4.18)	37.7(±4.20)	35.9(±4.11)
Avg. Win%	42.4(±1.11)	43.2(±1.11)	45.4(±1.12)

Table E.6: Win percentage of the AP agent that tunes four parameters on-line with the NTBEA allocation strategy, for different number of generated neighbors X.

Game	**AP$_{\text{NTBEA}}$**			
	$X = 5$	$X = 10$	$X = 50$	$X = 100$
3D Tic Tac Toe	38.0(±3.95)	37.1(±4.07)	23.6(±3.48)	22.7(±3.52)
Breakthrough	48.4(±4.38)	50.2(±4.39)	30.4(±4.04)	21.6(±3.61)
Knightthrough	62.6(±4.25)	67.2(±4.12)	48.0(±4.38)	35.2(±4.19)
Chinook	52.5(±4.01)	51.9(±4.06)	30.0(±3.75)	29.0(±3.71)
Chin.Checkers3P	44.0(±4.34)	37.3(±4.23)	34.3(±4.15)	34.3(±4.15)
Checkers	31.7(±3.76)	36.4(±3.97)	30.2(±3.74)	26.5(±3.67)
Connect Five	24.9(±2.89)	26.1(±2.86)	29.1(±3.12)	25.3(±3.02)
Quad	53.0(±4.13)	44.1(±4.07)	37.6(±4.07)	32.4(±3.94)
Sheep and Wolf	47.4(±4.38)	48.0(±4.38)	47.4(±4.38)	47.8(±4.38)
TTCC4 2P	46.2(±4.25)	44.8(±4.23)	30.7(±3.95)	24.6(±3.71)
TTCC4 3P	44.0(±4.23)	44.2(±4.23)	37.6(±4.12)	33.4(±4.04)
Connect Four	49.2(±4.19)	48.0(±4.22)	34.0(±3.98)	32.1(±3.90)
Pentago	40.1(±4.14)	46.3(±4.15)	32.1(±3.93)	27.0(±3.77)
Reversi	38.4(±4.22)	32.6(±4.05)	33.7(±4.07)	34.0(±4.09)
Avg. Win%	44.3(±1.11)	43.9(±1.11)	34.2(±1.07)	30.4(±1.04)

Table E.7: Win percentage of the AP agent that tunes four parameters on-line with the NTBEA allocation strategy, using different lengths for the n-tuples.

Game	**AP$_{\text{NTBEA}}$**	
	$\mathcal{L} = \{1, 4\}$	$\mathcal{L} = \{1, 2, 3, 4\}$
3D Tic Tac Toe	38.0(±3.95)	36.8(±3.97)
Breakthrough	48.4(±4.38)	46.8(±4.38)
Knightthrough	62.6(±4.25)	62.0(±4.26)
Chinook	52.5(±4.01)	51.7(±4.08)
Chin.Checkers3P	44.0(±4.34)	38.7(±4.26)
Checkers	31.7(±3.76)	29.3(±3.72)
Connect Five	24.9(±2.89)	26.0(±2.82)
Quad	53.0(±4.13)	51.4(±4.16)
Sheep and Wolf	47.4(±4.38)	46.4(±4.38)
TTCC4 2P	46.2(±4.25)	41.1(±4.20)
TTCC4 3P	44.0(±4.23)	45.6(±4.27)
Connect Four	49.2(±4.19)	48.5(±4.20)
Pentago	40.1(±4.14)	41.5(±4.16)
Reversi	38.4(±4.22)	35.1(±4.11)
Avg. Win%	44.3(±1.11)	42.9(±1.11)

Appendix F

Supplementary Results for Chapter 7

This appendix reports the detailed results for the graph presented in Figure 7.2. The graph shows the win percentage over all the tested games of the agent instances that randomize different numbers of parameters per game ($AP_{\text{GAME-RND}}$), per turn ($AP_{\text{TURN-RND}}$), per simulation ($AP_{\text{SIM-RND}}$) and per state ($AP_{\text{STATE-RND}}$), against the agent that uses fixed parameter values (AP). The win percentage of these agents for each game is reported in Table F.1.

Results in the table confirm that among the randomization strategies the one randomizing per simulation is performing best in most of the considered games, although randomization per state is also quite close in performance for many games. Randomization per turn seems to be the worst in most of the games, especially when the number of randomized parameters increases. The performance of randomization per game is also lower when compared to randomization per simulation or per state.

Interestingly, although the overall performance shows that no randomization strategy is significantly better than fixed default parameter values, there are still some specific games for which randomizing seems positive. In Quad, for example, randomization per simulation performs significantly better than fixed parameter values for all numbers of randomized parameters. The same holds for simulation per state, although for four parameters the performance increase is not significant. Moreover, when randomizing two parameters per state or per simulation also the performance in Chinook is significantly better than the performance of the fixed parameter values. Finally, a significant increase in performance is visible also for Connect Four when randomizing four parameters per simulation. This supports the claim that search-control parameter randomization might be beneficial for some games.

Table F.1: Win percentage of the AP agent that randomizes two, four and six parameters with different randomization strategies, against the AP agent with fixed default parameter values.

Game	2 parameters			
	$AP_{GAME-RND}$	$AP_{TURN-RND}$	$AP_{SIM-RND}$	$AP_{STATE-RND}$
3D Tic Tac Toe	36.8(±3.97)	35.9(±4.00)	**46.4**(±4.12)	35.4(±3.94)
Breakthrough	34.8(±4.18)	36.6(±4.23)	**40.4**(±4.31)	35.0(±4.19)
Knightthrough	41.8(±4.33)	43.2(±4.35)	**44.2**(±4.36)	43.4(±4.35)
Chinook	47.3(±4.06)	47.6(±4.02)	**61.8**(±3.99)	57.9(±4.08)
Chin.Checkers3P	44.4(±4.34)	**47.8**(±4.37)	45.6(±4.35)	47.4(±4.36)
Checkers	36.9(±4.01)	35.3(±3.97)	**48.8**(±4.08)	47.0(±4.11)
Connect Five	36.3(±3.14)	35.6(±3.11)	43.9(±3.12)	**45.2**(±3.20)
Quad	46.8(±4.12)	48.4(±4.24)	65.0(±3.93)	**67.4**(±3.80)
Sheep and Wolf	46.6(±4.38)	48.4(±4.38)	**52.0**(±4.38)	49.4(±4.39)
TTCC4 2P	35.4(±4.06)	40.1(±4.16)	**49.5**(±4.27)	46.4(±4.21)
TTCC4 3P	44.6(±4.25)	46.4(±4.24)	**48.7**(±4.26)	48.6(±4.24)
Connect Four	40.1(±4.10)	40.5(±4.07)	**50.9**(±4.17)	45.5(±4.24)
Pentago	42.6(±4.20)	45.5(±4.19)	**53.7**(±4.19)	50.9(±4.20)
Reversi	38.4(±4.18)	35.5(±4.13)	42.8(±4.29)	**45.5**(±4.28)
Avg. Win%	40.9(±1.10)	41.9(±1.11)	**49.6**(±1.12)	47.5(±1.12)

Game	4 parameters			
	$AP_{GAME-RND}$	$AP_{TURN-RND}$	$AP_{SIM-RND}$	$AP_{STATE-RND}$
3D Tic Tac Toe	31.7(±3.85)	18.2(±3.22)	**39.4**(±4.00)	28.6(±3.66)
Breakthrough	**15.6**(±3.18)	10.0(±2.63)	11.0(±2.75)	9.0(±2.51)
Knightthrough	**22.2**(±3.65)	20.8(±3.56)	20.6(±3.55)	15.8(±3.20)
Chinook	22.5(±3.40)	16.4(±2.98)	**23.6**(±3.61)	18.9(±3.28)
Chin.Checkers3P	**35.7**(±4.19)	29.2(±3.97)	34.7(±4.16)	31.3(±4.05)
Checkers	18.5(±3.19)	9.0(±2.35)	17.9(±3.17)	**18.6**(±3.20)
Connect Five	30.7(±3.15)	25.8(±2.96)	36.5(±3.22)	**41.8**(±3.49)
Quad	47.3(±4.20)	46.7(±4.18)	**72.8**(±3.71)	53.6(±4.23)
Sheep and Wolf	48.6(±4.39)	44.2(±4.36)	**49.8**(±4.39)	42.0(±4.33)
TTCC4 2P	**23.5**(±3.62)	14.5(±3.05)	20.6(±3.47)	17.2(±3.28)
TTCC4 3P	41.9(±4.23)	38.4(±4.16)	**46.1**(±4.28)	45.8(±4.28)
Connect Four	46.3(±4.17)	45.5(±4.22)	**55.4**(±4.13)	42.6(±4.16)
Pentago	40.2(±4.17)	32.4(±4.01)	**50.2**(±4.22)	30.0(±3.90)
Reversi	25.7(±3.75)	31.0(±4.01)	31.0(±3.99)	**31.3**(±3.99)
Avg. Win%	32.2(±1.05)	27.3(±1.01)	**36.4**(±1.08)	30.5(±1.04)

Game	6 parameters			
	$AP_{GAME-RND}$	$AP_{TURN-RND}$	$AP_{SIM-RND}$	$AP_{STATE-RND}$
3D Tic Tac Toe	26.6(±3.67)	15.2(±3.03)	**40.6**(±4.01)	27.8(±3.55)
Breakthrough	**13.2**(±2.97)	6.2(±2.12)	9.4(±2.56)	6.8(±2.21)
Knightthrough	19.4(±3.47)	11.4(±2.79)	19.8(±3.50)	**23.0**(±3.69)
Chinook	**27.2**(±3.71)	14.4(±2.93)	19.8(±3.34)	25.9(±3.72)
Chin.Checkers3P	30.6(±4.03)	25.6(±3.81)	**33.7**(±4.13)	29.4(±3.98)
Checkers	14.4(±2.91)	7.8(±2.24)	**20.4**(±3.29)	**20.4**(±3.37)
Connect Five	30.6(±3.02)	19.1(±2.77)	**45.7**(±3.21)	38.5(±3.38)
Quad	40.6(±4.14)	34.8(±4.00)	**72.4**(±3.67)	54.1(±4.25)
Sheep and Wolf	43.6(±4.35)	41.2(±4.32)	**50.0**(±4.39)	42.8(±4.34)
TTCC4 2P	**20.2**(±3.46)	12.0(±2.84)	19.0(±3.40)	15.6(±3.17)
TTCC4 3P	40.4(±4.20)	37.1(±4.13)	40.8(±4.23)	**41.6**(±4.23)
Connect Four	33.5(±3.99)	33.4(±3.98)	**48.0**(±4.23)	40.5(±4.14)
Pentago	31.2(±3.94)	25.9(±3.75)	**42.6**(±4.13)	29.3(±3.85)
Pentago	27.1(±3.85)	21.0(±3.52)	28.7(±3.92)	**34.7**(±4.10)
Avg. Win%	28.5(±1.02)	21.8(±0.94)	**35.1**(±1.07)	30.7(±1.04)

Index

$\alpha\beta$-search 10
"open-loop" MCTS 27
All-Moves-As-First 104
allocation strategy 126, 127
  continuous 142
    CMA-ES 142
  discrete 129
    EA 137
    HE 130
    LSI 134
    MAB 129
    NMC 132
    NTBEA 138
Artificial General Intelligence . 1, 8, 208
CMAB 127
decay 39, 41, 48
first-play urgency 36
Flat Monte-Carlo Search 11, 20
FPGA 95
game 2, 17
  abstract games 2
  board games 2
  card games 2
  complete-information 4
  constant-sum 4
  deterministic 4
  digital games 3
  Eurogames 2
  imperfect-information 4
  incomplete-information 4
  multi-player 3
  non-deterministic 4
  one-player 3
  perfect-information 3
  real-time 3
  sequential move 3
  simultaneous move 3
  single-player 3
  stochastic 4
  tile-based game 2
  turn-based 3
  turn-taking 3
  two-player 3
  variable-sum 4
  video games 3
  zero-sum 4
game model 18, 56, 65
game tree 18
GDL 52
general game playing 1, 8–10
GGP Base agent 59
graph reuse 48
GVG-AI agents 66
GVG-AI project 10, 61
MAST 38
minimax 10
Monte-Carlo evaluations 11, 20
Monte-Carlo Tree Search 1, 11, 23
Multi-Armed Bandit problem ... 20, 22
naïve assumption 133
NST 40
parameter randomization 173
  per game run 173
  per simulation 173
  per state 175
  per turn 173
progressive history 37
PropNet 72
PropNet optimizations 76
  Opt0 76
  Opt1 78
  Opt2 80

- Opt3 83
RAVE variants 105
- GRAVE 106
- HRAVE 108
- RAVE 37, 105

reasoner 56, 85, 95
SA-MCTS 126
search tree 20
simultaneous move UCT 31
- DUCT 34
- SUCT 32

Stanford GGP project 9, 51
transposition tables 41
tree reuse 47
UCB1 .... 11, 22
UCT 11, 30, 43
- UCT0 44
- UCT1 45
- UCT2 45
- UCT3 46

VGDL 61

Valorization

Please note: the *Regulations for obtaining the doctoral degree* of Maastricht University, dated 1 September 2018, require the addition of a *valorization addendum* to each dissertation. "While the valorisation addendum is included in the thesis, it is not regarded as part of the thesis and must not be taken into account in the assessment of the thesis by the Assessment Committee and the Defence Committee".[1]

Knowledge valorization is "the process of creating value from knowledge, by making knowledge suitable and/or available for social (and/or economic) use and by making it suitable for translation into competing products, services, processes and new activities". This addendum discusses various ways in which the research presented in the thesis might contribute to create social and economic value. We mainly discuss the value that can be created by advancing research on Monte-Carlo Tree Search (MCTS), giving also some insights on how value can be created by using MCTS to advance research towards Artificial General Intelligence (AGI). First, valorization opportunities in game-related domains are discussed. Next, the discussion is extended to other domains. Finally, we argue that valorisation opportunities can arise from research on MCTS that moves in the direction of AGI.

Games

This thesis uses (video) games as a test bed for MCTS, therefore the gaming industry is the first application domain that comes to mind where the presented research has the potential to create value. Video games are getting more and more realistic and sophisticated, demanding smarter AI characters that are able to deal with increasingly complex environments. Moreover, if AI characters show unrealistic behaviors, the interest of the players in the game will decrease and the game will lose its entertainment value. MCTS can be used to model believable AI characters in commercial applications, and there have been already some examples of its success on this task. In the *Spades* mobile phone game by AI Factory MCTS has been used to create engaging AI opponents and allies (Whitehouse *et al.*, 2013), and has later been improved to emulate human play (Baier *et al.*, 2018). Moreover, MCTS has been used in the development of the AI system for the on-line turn-based strategy

[1] `https://www.maastrichtuniversity.nl/support/phds`, retrieved 4 September 2019.

game *Prismata*, by Lunarch Studios (Churchill and Buro, 2015), and in the development of the AI system of *Total War: Rome II*, a strategy game developed by Creative Assembly and published by Sega (Thompson, 2018).

Potential applications of MCTS to games are not only limited to the development of AI characters for the game industry. MCTS can also be useful during the process of game design and game content generation. A possibility is to use it to evaluate generated games. For instance, Browne (2012) computes game evaluation metrics using the performance of UCT-based search on such games. Other approaches apply MCTS to directly generate games or game content. For example, it has been used to generate initial tile placements for Pentominoes puzzles (Browne, 2013) and to generate Sokoban puzzles (Kartal, Sohre, and Guy, 2016). In addition, it has been applied to automatically generate narratives (Kartal, Koenig, and Guy, 2014), which could be used in virtual environments or in video games to dynamically generate new plot lines.

Another application domain for which MCTS can create value are serious games, i.e. games that combine the playfulness of video games with a serious purpose, such as training, education or rehabilitation. An example of a serious game that makes use of MCTS is given by Sanselone *et al.* (2014), which use this search technique to control the behavior of virtual characters in a simulated surgery room environment for medical staff training. Another example is the work of Hocine, Gouaïch, and Cerri (2014), which use MCTS in the development of a game aimed at rehabilitating patients that suffered from a stroke.

Other Domains

The potential to create value from research on MCTS and its enhancements is not limited to games alone. MCTS is a suitable technique for domains that involve planning, optimization and decision making, and there are many examples of successful application of MCTS and its variants in a different number of fields.

In robotics, for instance, MCTS-based algorithms have been used for various purposes, ranging from controlling the movement of robots to managing their perception. Using MCTS variants, Goldhoorn *et al.* (2014) design robots that are able to find and follow people in a real-world scenario, Zech, Xiong, and Piater (2015) propose an optimization method that improves the precision of robotic grasps and Kartal (2015) addresses the multi-robot patrolling problem. Moreover, Patten, Martens, and Fitch (2018) propose an object classification mechanism for robots that operate in an outdoor environment and Best *et al.* (2019) design a decentralized mechanism that controls multiple robots with active perception of the world.

Other applications of MCTS can be found in the medical field. For example, MCTS has been used to schedule elective admissions of patients to the hospital (Van Eyck *et al.*, 2013; Zhu *et al.*, 2014) and to improve the process of selecting personalized health-care services in order to reduce the risk of re-hospitalization of patients (Laschet, 2014). In addition, Pinheiro, Kybic, and Fua (2017) design a mechanism based on MCTS to match graph structures to medical images.

Space exploration is another example of task for which the use of MCTS has been

considered. Hennes and Izzo (2015) apply MCTS to automatically plan interplanetary trajectories, while Song and Gong (2019) use it to find the optimal exploration sequence of near-earth asteroids. In addition, Arora, Fitch, and Sukkarieh (2017) show that MCTS is suitable to design a robot that is able to perform exploration of a Mars-analog environment autonomously, being able to plan its actions on-line according to the perceptions acquired from the environment.

Various applications of MCTS can be found also in logistic (Edelkamp *et al.*, 2016). For instance, there are multiple studies that address transportation problems. Trunda and Barták (2013) show how MCTS can be used for planning in multiple transportation domains. Moreover, Al-Kanj, Powell, and Bouzaiene-Ayari (2016) present an MCTS-based solution for routing a utility truck that restores outages in the power grid, actively collecting information and updating its beliefs about the state of the network. Abdo, Edelkamp, and Lawo (2016), instead, propose to use MCTS to design an algorithm that is able to deal with different variations of the vehicle routing problem with multiple vehicles, instead of designing a specific solution for each variation. A similar problem is addressed by Jiang, Al-Kanj, and Powell (2017), which design a variant of the MCTS algorithm that optimizes the behavior of a single driver navigating a graph while operating on a ride-sharing platform. Finally, Cazenave, Balbo, and Pinson (2009) show how a variant of MCTS, nested Monte-Carlo search, can be applied to urban transportation in order to minimize the waiting time of bus passengers, and Mirheli and Hajibabai (2019) show how to use MCTS to manage parking utilization in such a way that the travelers' cost is minimized and the parking agency's revenue is maximized.

MCTS has also been successfully applied to energy management. It has been used to solve stochastic energy stock management problems (Couëtoux and Doghmen, 2011), and to balance electricity supply and demand in power grids (Golpayegani, Dusparic, and Clarke, 2015).

Many more examples of the application of MCTS variants to concrete problems can be mentioned. These include designing security models (Marecki, Tesauro, and Segal, 2012; Guo, An, and Kolobov, 2015), military simulation and planning (Marks *et al.*, 2013; Teter *et al.*, 2014), protecting natural resources from illegal extraction (Qian *et al.*, 2014), deciphering encrypted text (Hauer, Hayward, and Kondrak, 2014) and forecasting financial volatility of assets (Cazenave and Hamida, 2015). Further examples are managing autonomous driving vehicles (Lenz, Kessler, and Knoll, 2016), managing wildfires (Bertsimas *et al.*, 2017), improving computer-aided retrosynthesis (Segler *et al.*, 2018) maximizing the performance of job scheduling heuristics (Wimmenauer, 2019), and finding optimal material designs (Dieb *et al.*, 2019). All these examples show the wide applicability of MCTS-based algorithms and many more applications are being explored every day.

Artificial General Intelligence

Research in AGI is based on the idea of having a program that is able to perform multiple different tasks in multiple different environments without any human intervention and with the ability to autonomously adapt to each new, possibly previously

unseen task. Although not yet achieving true AGI, programs that are able to cope with tasks of different nature without needing human intervention are suitable to be applied in many real-world scenarios. They are potentially useful for environments in which the type of tasks that the machine has to face might change over time, but they cannot be predicted or even imagined by the programmers. Moreover, such programs might be successful in environments for which it is too time consuming or even physically impossible to re-program the machine for each new type of task. For example, for a robot on a space mission there might be scenarios that the programmers have not thought about, therefore the robot should be able to learn by itself how to cope with them and how to perform the necessary tasks to deal with the unforeseen situation. Moreover, programmers are not able to physically access the robot to re-program it once it has reached space.

The discussion in previous sections highlighted how MCTS is a technique that can be applied successfully to many domains of social and economic interest. Other valorization opportunities created by the research presented in this thesis arise from the possibility of using the proposed domain-independent MCTS enhancements to develop computer programs and systems that are capable of dealing with multiple tasks in various domains at the same time, following the direction of AGI.

Summary

This thesis investigates how search can be utilized to support *Artificial General Intelligence* (AGI) in games. The aim of AGI is creating Artificial Intelligence (AI) that can perform multiple different tasks in multiple different environments, can autonomously manage itself and possesses the ability to adapt to perform any new task that it might have never performed before. Planning is among the competences that an AGI is expected to possess. Given that games can model a wide variety of computationally hard problems, they can be considered a reasonable subset of all the planning tasks that we would like an AGI to be able to perform. Therefore, *general game playing* (GGP), which aims at creating programs that are able to play many (video) games with different properties without requiring any human intervention, is identified as a suitable domain to test search techniques for AGI. In addition, *Monte-Carlo Tree Search* (MCTS) is presented as a successful search technique for domains like GGP, where no specific domain knowledge is available.

Chapter 1 provides an introduction to games, discusses their relevance for AI and AGI and gives a brief description of the most popular search techniques used by game AI programs, among which is MCTS. The following problem statement guides the research.

> **Problem statement**: *How can the performance of Monte-Carlo Tree Search for general game playing be improved?*

To answer the problem statement four research questions have been formulated. They deal with (1) speeding up the interpretation of game rules written in a declarative language, (2) evaluating the use of global or local information to enhance the selection strategy of MCTS, (3) on-line tuning search-control parameters for MCTS, and (4) investigating the effect of search-control parameter randomization in MCTS.

Chapter 2 gives a formal definition of the games considered in the thesis and defines the terms commonly used when referring to tree search. Furthermore, Monte-Carlo methods are discussed, together with their link to MCTS, which is presented next. Finally, the UCT selection strategy is described both for sequential and simultaneous move games and relevant MCTS enhancements are discussed.

Chapter 3 introduces the test environments used to answer the four research questions: the *Stanford General Game Playing* (Stanford GGP) project and the *General Video Game AI* (GVG-AI) project. The first one focuses on GGP for abstract games, while the second on GGP for arcade-style video games. For each of the two environments the chapter introduces the language used to describe the

corresponding games and describes how the execution of a game run is managed. Moreover, the rules of the competition associated with each environment are presented, together with the common implementation details of the agents that are tested in the subsequent chapters of the thesis.

The Stanford GGP project represents the game rules using a declarative language known as the *Game Description Language* (GDL). This means that agents have to implement a mechanism that interprets the game rules written in GDL and computes all the elements that are necessary to reason on the game (i.e., game states, legal moves, etc.). This interpretation process is in general slow and might reduce the number of simulations that an agent can perform. This might hinder the performance of MCTS, which instead benefits from the more accurate statistics that can be collected with a higher number of simulations. This has led to the formulation of the first research question.

> **Research question 1**: *How can the process of interpreting on-line the game rules written in a declarative language be sped up?*

Chapter 4 answers the first research question by investigating an interpreter based on the representation of the GDL game rules as a Propositional Network (PropNet). Results show that a software implementation of the PropNet performs better than the GGP Base Prover, a custom made interpreter for GDL rules. The software implementation of the PropNet increases the number of game simulations by an average of two orders of magnitude with respect to the GGP Base Prover. Moreover, the speed of the PropNet is further increased by applying four optimizations, which, in order, remove PropNet components with constant truth values, remove PropNet propositions that do not have any special meaning for the game, detect and then remove components that will only assume a constant truth value during the game reasoning process, and remove components that have no output and thus no particular meaning for the game. The use of a cache that memorizes results previously computed by the PropNet is shown to increase the overall speed further. However, its use is recommended only for games with a small search space, like Chinese Checkers with 1 player and Tic Tac Toe, for which it increases the speed already in the first search steps. Furthermore, the use of a PropNet based reasoner enables the MCTS agent to reach a win rate close to 100% against an agent that uses the Prover. Finally, the speed of a PropNet-based reasoner can be further increased by at least one order of magnitude by encoding the PropNet on a Field Programmable Gate Array (FPGA) board. It may be concluded that using a PropNet with an optimized structure to represent game rules written in GDL is beneficial for MCTS-based agents, because they are able to perform more simulations in the given time frame and they can profit from hardware acceleration.

Previous research has shown that MCTS also profits from enhancing its selection strategy by increasing the amount of information used to guide the search. The Rapid Action Value Estimation (RAVE) strategy and the Generalized Rapid Action Value Estimation (GRAVE) strategy have been shown to be successful enhancements for the selection phase of MCTS. Both of them bias the selection towards actions that seem to perform generally well in the game. However, GRAVE uses more global information than RAVE to bias action selection in nodes that only have a low

number of visits. This strategy has been shown to perform better than RAVE on some variants of Go and a few other games, therefore it might be successful in GGP as well. This has led to the formulation of the second research question.

> **Research question 2**: *What is the effect of using locally or globally collected information to enhance the selection strategy for Monte-Carlo Tree Search?*

Chapter 5 answers the second research question proposing another variant of RAVE, the History Rapid Action Value Estimation (HRAVE) strategy, which biases action selection always using global information about the actions. It then compares the performance of RAVE, GRAVE and HRAVE on a set of games from the Stanford GGP project, both combined with a random play-out strategy and with the more informed MAST play-out strategy. Results show that, when RAVE variants are combined with a random play-out strategy, the performance of GRAVE is, in the worst case, comparable with the one of RAVE, both when using 1s or 10s play-clock. The performance of HRAVE, instead, is more game dependent, sometimes being better than RAVE or GRAVE and sometimes being worse. Moreover, when RAVE variants are combined with the Move Average Sampling Technique (MAST) used as play-out strategy, GRAVE still seems to be overall better than RAVE. However, its advantage is less than when both strategies are combined with random play-outs, and there are a few games where the combination GRAVE-MAST actually performs worse than RAVE-MAST. Additionally, the combination HRAVE-MAST seems to perform slightly less than both RAVE-MAST and GRAVE-MAST. Over all the experiments, the difference in performance between RAVE, GRAVE and HRAVE is not large. However, the overall win rate of GRAVE is never inferior to the one of RAVE and HRAVE, and it seems the most robust among all the RAVE variants. Therefore, it may be concluded that a strategy that starts biasing action selection with global information and uses more local information the more the nodes have been visited is the most suitable to enhance MCTS for GGP. In addition, and advantage of GRAVE is that it can be switched to a pure RAVE or a pure HRAVE strategy by simply modifying one of its parameters. With respect to the other two variants, this makes it more promising to be tuned on-line with the approach presented in Chapter 6.

Together with RAVE, many other enhancements for the different phases of MCTS have been applied successfully in GGP. Often, MCTS and its enhancements are controlled by multiple parameters that require extensive and time-consuming off-line optimization. Moreover, as the played games are unknown in advance, off-line optimization cannot tune parameters specifically for single games. It has to find values that perform overall well on a predefined set of games, with no guarantee that they will perform successfully also on unseen games. An alternative would be to adjust parameter values while playing each new game, therefore in an on-line fashion. This has led to the formulation of the third research question.

> **Research question 3**: *How can search-control parameters for Monte-Carlo Tree Search be tuned effectively on-line?*

Chapter 6 answers the third research question proposing an on-line tuning method for search-control parameters that enables MCTS to be self-adaptive during game play (SA-MCTS). Seven different strategies were introduced to decide how to allocate the available samples to test the parameter combinations: Multi-Armed Bandit (MAB) allocation, Hierarchical Expansion (HE), Naïve Monte-Carlo (NMC), Linear Side Information (LSI), an Evolutionary Algorithm (EA), the N-Tuple Bandit Evolutionary Algorithm (NTBEA) and the Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES). The performance of on-line parameter tuning has been tested both on the Stanford GGP and the GVG-AI projects. Results show that among the tested allocation strategies to tune parameters on-line, the ones considering a discrete parameter domain and based on evolutionary algorithms perform best. NTBEA seems to have the best performance overall, but EA is also quite close. Results for the Stanford GGP project show that on-line parameter tuning is beneficial both for simple and more informed agents, when two parameters are tuned. The performance decreases when tuning more parameters. However, when tuning four parameters, the performance is still close to the one obtained using fixed default parameter values. Results for the GVG-AI project show that it is harder to tune parameters on-line with much shorter time settings, even when the number of tuned parameters is small. However, it may still be better to tune parameters on-line when fixed parameter settings might be sub-optimal, such as it is seen in the game Modality. It may be concluded that the proposed approach is useful when off-line parameter tuning is infeasible, or in contexts like GGP, both for abstract and real-time games, where parameters cannot be tuned in advance for each game. Moreover, it is useful when off-line tuned values might be sub-optimal for some games, or off-line tuning incurs in the risk of overfitting the values to the set of games selected for the purpose of tuning. It may also be concluded that on-line parameter tuning is robust against different types of opponents.

The success of on-line search-control parameter tuning on some of the tested games might be partially due to the randomization introduced by exploring different parameter combinations. This might be introducing diversification in the search process, making it explore different parts of the tree that would not be explored keeping the parameters fixed. Moreover, previous research has shown that adding randomization to certain components of the search might increase its diversification and improve its performance. In a domain like GGP, that deals with many games with different characteristics, adding more randomization might be a good strategy for some games. This has led to the formulation of the fourth research question.

> **Research question 4**: *What is the effect of randomizing search-control parameters for Monte-Carlo Tree Search?*

Chapter 7 answers this research question evaluating four different strategies that randomize search-control parameters for MCTS in GGP: randomization per game, per turn, per simulation and per state. Moreover, search-control parameter randomization is compared with fixed parameter settings and with on-line parameter tuning both in the framework of the Stanford GGP project and in the framework of the GVG-AI project. For the Stanford GGP project, results show that the randomization strategy that performs best is the one that randomizes parameter values

before each simulation, selecting such values within a predefined reasonable interval. Moreover, results show that for some games randomizing per simulation the value of a single parameter is better than keeping a good value fixed for the whole game. Furthermore, results show that the effect of parameter randomization depends on multiple factors, such as the game being played, which and how many parameters are being randomized and the type of opponent. It may be concluded that, although not always the best solution for all games, randomization within a given set of values is still beneficial in GGP for games where the fixed parameter settings optimized off-line on a predefined set of games are actually performing poorly. Moreover, parameter randomization might be a valid alternative to on-line parameter tuning when the number of parameters to tune is high and time settings are limited, because the problem of tuning them on-line becomes too hard due to the combinatorial complexity.

Chapter 8 concludes the thesis and gives indications for future research directions. The answer to the problem statement is based on the answers to the research questions given above. First, the process of interpreting the game rules written in a declarative language can be sped up by using a PropNet representation of these rules, optimizing the structure of such PropNet, and embedding the PropNet structure on an FPGA. By speeding up the process of interpreting the game rules, the number of simulations that can be performed by MCTS can be increased. Second, the selection strategy of MCTS can be enhanced using both locally and globally collected information about the available actions. In this case, the best approach is using a mix of global and local information, the first for states that have been visited less and the second for states that have been visited more, like the GRAVE selection strategy does. Third, search-control parameters can be tuned on-line and adapted to each new game being played, using the NTBEA strategy to allocate samples for evaluating parameter combinations. Fourth, randomizing search-control parameters within a predefined set of values before each simulation can be used as an alternative to on-line parameter tuning when the number of parameters to tune is high and the time settings are limited.

All the approaches presented in the thesis have been shown to enhance the performance of MCTS for a wide variety of games, without relying on game-specific pre-coded information. Although evaluated only on a subset of all the planning tasks that AGI is aiming to tackle, the games considered in this thesis present a wide variety of characteristics. They include abstract games, video games, deterministic and non-deterministic games, games with a discrete or continuous game flow, with sequential or simultaneous moves, with constant-sum or variable-sum payoffs, and with different numbers of players. Therefore, the presented MCTS enhancements are promising to also support search and planning for AGI.

The research presented in the thesis indicates several areas for future research. These include specific recommendations such as (i) further speeding up the PropNet reasoner and its implementation on an FPGA, (ii) further testing the RAVE variants, using different formulas to compute their parameters and combining them with different play-out strategies, (iii) improving on-line parameter tuning by testing other allocation strategies, increasing their adaptability to each played game and applying them to other domains, and (iv) testing parameter randomization with longer

time constraints and adding a mechanism that decides when and which parameters to randomize. More generic recommendations for future research suggest to test MCTS for AGI on more challenging domains and to focus on designing dynamic approaches that automatically adapt different aspects of the search and of the agent to each game or to each category of games being played.

Curriculum Vitae

Chiara Sironi was born on August 24, 1988 in Desio, Italy. She attended secondary school first at Liceo Ettore Majorana, in Desio from 2002 to 2005, and subsequently at Liceo Marie Curie, in Meda from 2005 to 2007. In 2008, she started studying Computer Science at the University of Milano–Bicocca, in Italy, and in 2011 she received her B.Sc. degree cum laude. Immediately thereafter, she started a master in Computer Science at the same university, and in 2014 she obtained the M.Sc. degree cum laude. During her master studies she also spent eight months as an exchange student at the University of Antwerp, in Belgium. Moreover, she carried out her master's thesis project at the University of Leuven, in Belgium, for five months. From 2015 to 2019 she worked as a Ph.D. researcher at the Department of Data Science and Knowledge Engineering, Maastricht University. The research performed there resulted in several publications and finally this thesis. She also received a best paper nomination at the renowned IEEE Conference on Games (COG) for her paper "Comparing Randomization Strategies for Search-Control Parameters in MCTS". Besides performing scientific tasks, she was involved in supervising students with their thesis work and in guiding them during the practical sessions of courses such as Data Structures and Algorithms, and Software Engineering. Furthermore, she participated in the organization of the 14th IEEE Conference on Computational Intelligence and Games (CIG) in Maastricht in August 2018.

SIKS Dissertation Series

2011

1 Botond Cseke (RUN[1]) *Variational Algorithms for Bayesian Inference in Latent Gaussian Models*

2 Nick Tinnemeier (UU) *Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language*

3 Jan Martijn van der Werf (TUE) *Compositional Design and Verification of Component-Based Information Systems*

4 Hado van Hasselt (UU) *Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference*

5 Bas van der Raadt (VU) *Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.*

6 Yiwen Wang (TUE) *Semantically-Enhanced Recommendations in Cultural Heritage*

7 Yujia Cao (UT) *Multimodal Information Presentation for High Load Human Computer Interaction*

8 Nieske Vergunst (UU) *BDI-based Generation of Robust Task-Oriented Dialogues*

9 Tim de Jong (OU) *Contextualised Mobile Media for Learning*

10 Bart Bogaert (UVT) *Cloud Content Contention*

11 Dhaval Vyas (UT) *Designing for Awareness: An Experience-focused HCI Perspective*

12 Carmen Bratosin (TUE) *Grid Architecture for Distributed Process Mining*

13 Xiaoyu Mao (UVT) *Airport under Control. Multiagent Scheduling for Airport Ground Handling*

14 Milan Lovric (EUR) *Behavioral Finance and Agent-Based Artificial Markets*

15 Marijn Koolen (UVA) *The Meaning of Structure: the Value of Link Evidence for Information Retrieval*

16 Maarten Schadd (UM) *Selective Search in Games of Different Complexity*

17 Jiyin He (UVA) *Exploring Topic Structure: Coherence, Diversity and Relatedness*

18 Mark Ponsen (UM) *Strategic Decision-Making in complex games*

19 Ellen Rusman (OU) *The Mind's Eye on Personal Profiles*

20 Qing Gu (VU) *Guiding service-oriented software engineering - A view-based approach*

21 Linda Terlouw (TUD) *Modularization and Specification of Service-Oriented Systems*

22 Junte Zhang (UVA) *System Evaluation of Archival Description and Access*

23 Wouter Weerkamp (UVA) *Finding People and their Utterances in Social Media*

24 Herwin van Welbergen (UT) *Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior*

25 Syed Waqar ul Qounain Jaffry (VU) *Analysis and Validation of Models for Trust Dynamics*

[1] Abbreviations: SIKS – Dutch Research School for Information and Knowledge Systems; CWI – Centrum voor Wiskunde en Informatica, Amsterdam; EUR – Erasmus Universiteit, Rotterdam; OU – Open Universiteit; RUN – Radboud Universiteit Nijmegen; TUD – Technische Universiteit Delft; TUE – Technische Universiteit Eindhoven; UL – Universiteit Leiden; UM – Universiteit Maastricht; UT – Universiteit Twente; UU – Universiteit Utrecht; UVA – Universiteit van Amsterdam; UVT – Universiteit van Tilburg; VU – Vrije Universiteit, Amsterdam.

26 Matthijs Aart Pontier (VU) *Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots*

27 Aniel Bhulai (VU) *Dynamic website optimization through autonomous management of design patterns*

28 Rianne Kaptein (UVA) *Effective Focused Retrieval by Exploiting Query Context and Document Structure*

29 Faisal Kamiran (TUE) *Discrimination-aware Classification*

30 Egon van den Broek (UT) *Affective Signal Processing (ASP): Unraveling the mystery of emotions*

31 Ludo Waltman (EUR) *Computational and Game-Theoretic Approaches for Modeling Bounded Rationality*

32 Nees-Jan van Eck (EUR) *Methodological Advances in Bibliometric Mapping of Science*

33 Tom van der Weide (UU) *Arguing to Motivate Decisions*

34 Paolo Turrini (UU) *Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations*

35 Maaike Harbers (UU) *Explaining Agent Behavior in Virtual Training*

36 Erik van der Spek (UU) *Experiments in serious game design: a cognitive approach*

37 Adriana Burlutiu (RUN) *Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference*

38 Nyree Lemmens (UM) *Bee-inspired Distributed Optimization*

39 Joost Westra (UU) *Organizing Adaptation using Agents in Serious Games*

40 Viktor Clerc (VU) *Architectural Knowledge Management in Global Software Development*

41 Luan Ibraimi (UT) *Cryptographically Enforced Distributed Data Access Control*

42 Michal Sindlar (UU) *Explaining Behavior through Mental State Attribution*

43 Henk van der Schuur (UU) *Process Improvement through Software Operation Knowledge*

44 Boris Reuderink (UT) *Robust Brain-Computer Interfaces*

45 Herman Stehouwer (UVT) *Statistical Language Models for Alternative Sequence Selection*

46 Beibei Hu (TUD) *Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work*

47 Azizi Bin Ab Aziz (VU) *Exploring Computational Models for Intelligent Support of Persons with Depression*

48 Mark Ter Maat (UT) *Response Selection and Turn-taking for a Sensitive Artificial Listening Agent*

49 Andreea Niculescu (UT) *Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality*

2012

1 Terry Kakeeto (UVT) *Relationship Marketing for SMEs in Uganda*

2 Muhammad Umair (VU) *Adaptivity, emotion, and Rationality in Human and Ambient Agent Models*

3 Adam Vanya (VU) *Supporting Architecture Evolution by Mining Software Repositories*

4 Jurriaan Souer (UU) *Development of Content Management System-based Web Applications*

5 Marijn Plomp (UU) *Maturing Interorganisational Information Systems*

6 Wolfgang Reinhardt (OU) *Awareness Support for Knowledge Workers in Research Networks*

7 Rianne van Lambalgen (VU) *When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions*

8 Gerben de Vries (UVA) *Kernel Methods for Vessel Trajectories*

9 Ricardo Neisse (UT) *Trust and Privacy Management Support for Context-Aware Service Platforms*

10 David Smits (TUE) *Towards a Generic Distributed Adaptive Hypermedia Environment*

11 J.C.B. Rantham Prabhakara (TUE) *Process Mining in the Large: Preprocessing, Discovery, and Diagnostics*

12 Kees van der Sluijs (TUE) *Model Driven Design and Data Integration in Semantic Web Information Systems*

13 Suleman Shahid (UVT) *Fun and Face: Exploring non-verbal expressions of emotion during playful interactions*

14 Evgeny Knutov (TUE) *Generic Adaptation Framework for Unifying Adaptive Web-based Systems*

15 Natalie van der Wal (VU) *Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes.*

16 Fiemke Both (VU) *Helping people by understanding them - Ambient Agents supporting task execution and depression treatment*

17 Amal Elgammal (UVT) *Towards a Comprehensive Framework for Business Process Compliance*

18 Eltjo Poort (VU) *Improving Solution Architecting Practices*

19 Helen Schonenberg (TUE) *What's Next? Operational Support for Business Process Execution*

20 Ali Bahramisharif (RUN) *Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing*

21 Roberto Cornacchia (TUD) *Querying Sparse Matrices for Information Retrieval*

22 Thijs Vis (UVT) *Intelligence, politie en veiligheidsdienst: verenigbare grootheden?*

23 Christian Muehl (UT) *Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction*

24 Laurens van der Werff (UT) *Evaluation of Noisy Transcripts for Spoken Document Retrieval*

25 Silja Eckartz (UT) *Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application*

26 Emile de Maat (UVA) *Making Sense of Legal Text*

27 Hayrettin Gurkok (UT) *Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games*

28 Nancy Pascall (UVT) *Engendering Technology Empowering Women*

29 Almer Tigelaar (UT) *Peer-to-Peer Information Retrieval*

30 Alina Pommeranz (TUD) *Designing Human-Centered Systems for Reflective Decision Making*

31 Emily Bagarukayo (RUN) *A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure*

32 Wietske Visser (TUD) *Qualitative multi-criteria preference representation and reasoning*

33 Rory Sie (OU) *Coalitions in Cooperation Networks (COCOON)*

34 Pavol Jancura (RUN) *Evolutionary analysis in PPI networks and applications*

35 Evert Haasdijk (VU) *Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics*

36 Denis Ssebugwawo (RUN) *Analysis and Evaluation of Collaborative Modeling Processes*

37 Agnes Nakakawa (RUN) *A Collaboration Process for Enterprise Architecture Creation*

38 Selmar Smit (VU) *Parameter Tuning and Scientific Testing in Evolutionary Algorithms*

39 Hassan Fatemi (UT) *Risk-aware design of value and coordination networks*

40 Agus Gunawan (UVT) *Information Access for SMEs in Indonesia*

41 Sebastian Kelle (OU) *Game Design Patterns for Learning*

42 Dominique Verpoorten (OU) *Reflection Amplifiers in self-regulated Learning*

43 (Withdrawn)

44 Anna Tordai (VU) *On Combining Alignment Techniques*

45 Benedikt Kratz (UVT) *A Model and Language for Business-aware Transactions*

46 Simon Carter (UVA) *Exploration and Exploitation of Multilingual Data for Statistical Machine Translation*

47 Manos Tsagkias (UVA) *Mining Social Media: Tracking Content and Predicting Behavior*

48 Jorn Bakker (TUE) *Handling Abrupt Changes in Evolving Time-series Data*

49 Michael Kaisers (UM) *Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions*

50 Steven van Kervel (TUD) *Ontology driven Enterprise Information Systems Engineering*

51 Jeroen de Jong (TUD) *Heuristics in Dynamic Scheduling; a practical framework with a case study in elevator dispatching*

2013

1 Viorel Milea (EUR) *News Analytics for Financial Decision Support*

2 Erietta Liarou (CWI) *MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing*

3 Szymon Klarman (VU) *Reasoning with Contexts in Description Logics*

4 Chetan Yadati (TUD) *Coordinating autonomous planning and scheduling*

5 Dulce Pumareja (UT) *Groupware Requirements Evolutions Patterns*

6 Romulo Gonçalves (CWI) *The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience*

7 Giel van Lankveld (UVT) *Quantifying Individual Player Differences*

8 Robbert-Jan Merk (VU) *Making enemies: cognitive modeling for opponent agents in fighter pilot simulators*

9 Fabio Gori (RUN) *Metagenomic Data Analysis: Computational Methods and Applications*

10 Jeewanie Jayasinghe Arachchige (UVT) *A Unified Modeling Framework for Service Design.*

11 Evangelos Pournaras (TUD) *Multi-level Reconfigurable Self-organization in Overlay Services*

12 Marian Razavian (VU) *Knowledge-driven Migration to Services*

13 Mohammad Safiri (UT) *Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly*

14 Jafar Tanha (UVA) *Ensemble Approaches to Semi-Supervised Learning Learning*

15 Daniel Hennes (UM) *Multiagent Learning - Dynamic Games and Applications*

16 Eric Kok (UU) *Exploring the practical benefits of argumentation in multi-agent deliberation*

17 Koen Kok (VU) *The PowerMatcher: Smart Coordination for the Smart Electricity Grid*

18 Jeroen Janssens (UVT) *Outlier Selection and One-Class Classification*

19 Renze Steenhuizen (TUD) *Coordinated Multi-Agent Planning and Scheduling*

20 Katja Hofmann (UVA) *Fast and Reliable Online Learning to Rank for Information Retrieval*

21 Sander Wubben (UVT) *Text-to-text generation by monolingual machine translation*

22 Tom Claassen (RUN) *Causal Discovery and Logic*

23 Patricio de Alencar Silva (UVT) *Value Activity Monitoring*

24 Haitham Bou Ammar (UM) *Automated Transfer in Reinforcement Learning*

25 Agnieszka Anna Latoszek-Berendsen (UM) *Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System*

26 Alireza Zarghami (UT) *Architectural Support for Dynamic Homecare Service Provisioning*

27 Mohammad Huq (UT) *Inference-based Framework Managing Data Provenance*

28 Frans van der Sluis (UT) *When Complexity becomes Interesting: An Inquiry into the Information eXperience*

29 Iwan de Kok (UT) *Listening Heads*

30 Joyce Nakatumba (TUE) *Resource-Aware Business Process Management: Analysis and Support*

31 Dinh Khoa Nguyen (UVT) *Blueprint Model and Language for Engineering Cloud Applications*

32 Kamakshi Rajagopal (OU) *Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development*

33 Qi Gao (TUD) *User Modeling and Personalization in the Microblogging Sphere*

34 Kien Tjin-Kam-Jet (UT) *Distributed Deep Web Search*

35 Abdallah El Ali (UVA) *Minimal Mobile Human Computer Interaction*

36 Than Lam Hoang (TUE) *Pattern Mining in Data Streams*

37 Dirk Börner (OU) *Ambient Learning Displays*

38 Eelco den Heijer (VU) *Autonomous Evolutionary Art*

39 Joop de Jong (TUD) *A Method for Enterprise Ontology based Design of Enterprise Information Systems*

40 Pim Nijssen (UM) *Monte-Carlo Tree Search for Multi-Player Games*

41 Jochem Liem (UVA) *Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning*

42 Léon Planken (TUD) *Algorithms for Simple Temporal Reasoning*

43 Marc Bron (UVA) *Exploration and Contextualization through Interaction and Concepts*

2014

1 Nicola Barile (UU) *Studies in Learning Monotone Models from Data*

2 Fiona Tuliyano (RUN) *Combining System Dynamics with a Domain Modeling Method*

3 Sergio Raul Duarte Torres (UT) *Information Retrieval for Children: Search Behavior and Solutions*

4 Hanna Jochmann-Mannak (UT) *Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation*

5 Jurriaan van Reijsen (UU) *Knowledge Perspectives on Advancing Dynamic Capability*

6 Damian Tamburri (VU) *Supporting Networked Software Development*

7 Arya Adriansyah (TUE) *Aligning Observed and Modeled Behavior*

8 Samur Araujo (TUD) *Data Integration over Distributed and Heterogeneous Data Endpoints*

9 Philip Jackson (UVT) *Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language*

10 Ivan Salvador Razo Zapata (VU) *Service Value Networks*

11 Janneke van der Zwaan (TUD) *An Empathic Virtual Buddy for Social Support*

12 Willem van Willigen (VU) *Look Ma, No Hands: Aspects of Autonomous Vehicle Control*

13 Arlette van Wissen (VU) *Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains*

14 Yangyang Shi (TUD) *Language Models With Meta-information*

15 Natalya Mogles (VU) *Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare*

16 Krystyna Milian (VU) *Supporting trial recruitment and design by automatically interpreting eligibility criteria*

17 Kathrin Dentler (VU) *Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability*

18 Mattijs Ghijsen (UVA) *Methods and Models for the Design and Study of Dynamic Agent Organizations*

19 Vinicius Ramos (TUE) *Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support*

20 Mena Habib (UT) *Named Entity Extraction and Disambiguation for Informal Text: The Missing Link*

21 Kassidy Clark (TUD) *Negotiation and Monitoring in Open Environments*

22 Marieke Peeters (UU) *Personalized Educational Games - Developing agent-supported scenario-based training*

23 Eleftherios Sidirourgos (UVA/CWI) *Space Efficient Indexes for the Big Data Era*

24 Davide Ceolin (VU) *Trusting Semi-structured Web Data*

25 Martijn Lappenschaar (RUN) *New network models for the analysis of disease interaction*

26 Tim Baarslag (TUD) *What to Bid and When to Stop*

27 Rui Jorge Almeida (EUR) *Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty*

28 Anna Chmielowiec (VU) *Decentralized k-Clique Matching*

29 Jaap Kabbedijk (UU) *Variability in Multi-Tenant Enterprise Software*

30 Peter de Cock (UVT) *Anticipating Criminal Behaviour*

31 Leo van Moergestel (UU) *Agent Technology in Agile Multiparallel Manufacturing and Product Support*

32 Naser Ayat (UVA) *On Entity Resolution in Probabilistic Data*

33 Tesfa Tegegne (RUN) *Service Discovery in eHealth*

34 Christina Manteli (VU) *The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems.*

35 Joost van Ooijen (UU) *Cognitive Agents in Virtual Worlds: A Middleware Design Approach*

36 Joos Buijs (TUE) *Flexible Evolutionary Algorithms for Mining Structured Process Models*

37 Maral Dadvar (UT) *Experts and Machines United Against Cyberbullying*

38 Danny Plass-Oude Bos (UT) *Making brain-computer interfaces better: improving usability through post-processing.*

39 Jasmina Maric (UVT) *Web Communities, Immigration, and Social Capital*

40 Walter Omona (RUN) *A Framework for Knowledge Management Using ICT in Higher Education*

41 Frederic Hogenboom (EUR) *Automated Detection of Financial Events in News Text*

42 Carsten Eijckhof (CWI/TUD) *Contextual Multidimensional Relevance Models*

43 Kevin Vlaanderen (UU) *Supporting Process Improvement using Method Increments*

44 Paulien Meesters (UVT) *Intelligent Blauw. Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden.*

45 Birgit Schmitz (OU) *Mobile Games for Learning: A Pattern-Based Approach*

46 Ke Tao (TUD) *Social Web Data Analytics: Relevance, Redundancy, Diversity*

47 Shangsong Liang (UVA) *Fusion and Diversification in Information Retrieval*

2015

1 Niels Netten (UVA) *Machine Learning for Relevance of Information in Crisis Response*

2 Faiza Bukhsh (UVT) *Smart auditing: Innovative Compliance Checking in Customs Controls*

3 Twan van Laarhoven (RUN) *Machine learning for network data*

4 Howard Spoelstra (OU) *Collaborations in Open Learning Environments*

5 Christoph Bösch (UT) *Cryptographically Enforced Search Pattern Hiding*

6 Farideh Heidari (TUD) *Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes*

7 Maria-Hendrike Peetz (UVA) *Time-Aware Online Reputation Analysis*

8 Jie Jiang (TUD) *Organizational Compliance: An agent-based model for designing and evaluating organizational interactions*

9 Randy Klaassen (UT) *HCI Perspectives on Behavior Change Support Systems*

10 Henry Hermans (OU) *OpenU: design of an integrated system to support lifelong learning*

11 Yongming Luo (TUE) *Designing algorithms for big graph datasets: A study of computing bisimulation and joins*

12 Julie M. Birkholz (VU) *Modi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks*

13 Giuseppe Procaccianti (VU) *Energy-Efficient Software*

14 Bart van Straalen (UT) *A cognitive approach to modeling bad news conversations*

15 Klaas Andries de Graaf (VU) *Ontology-based Software Architecture Documentation*

16 Changyun Wei (UT) *Cognitive Coordination for Cooperative Multi-Robot Teamwork*

17 André van Cleeff (UT) *Physical and Digital Security Mechanisms: Properties, Combinations and Trade-offs*

18 Holger Pirk (CWI) *Waste Not, Want Not! - Managing Relational Data in Asymmetric Memories*

19 Bernardo Tabuenca (OU) *Ubiquitous Technology for Lifelong Learners*

20 Lois Vanhée (UU) *Using Culture and Values to Support Flexible Coordination*

21 Sibren Fetter (OU) *Using Peer-Support to Expand and Stabilize Online Learning*

22 Zhemin Zhu (UT) *Co-occurrence Rate Networks*

23 Luit Gazendam (VU) *Cataloguer Support in Cultural Heritage*

24 Richard Berendsen (UVA) *Finding People, Papers, and Posts: Vertical Search Algorithms and Evaluation*

25 Steven Woudenberg (UU) *Bayesian Tools for Early Disease Detection*

26 Alexander Hogenboom (EUR) *Sentiment Analysis of Text Guided by Semantics and Structure*

27 Sándor Héman (CWI) *Updating compressed column-stores*

28 Janet Bagorogoza (UVT) *Knowledge Management and High Performance; The Uganda Financial Institutions Model for HPO*

29 Hendrik Baier (UM) *Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains*

30 Kiavash Bahreini (OU) *Real-time Multimodal Emotion Recognition in E-Learning*

31 Yakup Koç (TUD) *On the robustness of Power Grids*

32 Jerome Gard (UL) *Corporate Venture Management in SMEs*

33 Frederik Schadd (TUD) *Ontology Mapping with Auxiliary Resources*

34 Victor de Graaf (UT) *Geosocial Recommender Systems*

35 Jungxao Xu (TUD) *Affective Body Language of Humanoid Robots: Perception and Effects in Human Robot Interaction*

2016

1 Syed Saiden Abbas (RUN) *Recognition of Shapes by Humans and Machines*

2 Michiel Christiaan Meulendijk (UU) *Optimizing medication reviews through decision support: prescribing a better pill to swallow*

3 Maya Sappelli (RUN) *Knowledge Work in Context: User Centered Knowledge Worker Support*

4 Laurens Rietveld (VU) *Publishing and Consuming Linked Data*

5 Evgeny Sherkhonov (UVA) *Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers*

6 Michel Wilson (TUD) *Robust scheduling in an uncertain environment*

7 Jeroen de Man (VU) *Measuring and modeling negative emotions for virtual training*

8 Matje van de Camp (UVT) *A Link to the Past: Constructing Historical Social Networks from Unstructured Data*

9 Archana Nottamkandath (VU) *Trusting Crowdsourced Information on Cultural Artefacts*

10 George Karafotias (VU) *Parameter Control for Evolutionary Algorithms*

11 Anne Schuth (UVA) *Search Engines that Learn from Their Users*

12 Max Knobbout (UU) *Logics for Modelling and Verifying Normative Multi-Agent Systems*

13 Nana Baah Gyan (VU) *The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach*

14 Ravi Khadka (UU) *Revisiting Legacy Software System Modernization*

15 Steffen Michels (RUN) *Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments*

16 Guangliang Li (UVA) *Socially Intelligent Autonomous Agents that Learn from Human Reward*

17 Berend Weel (VU) *Towards Embodied Evolution of Robot Organisms*

18 Albert Meroño Peñuela (VU) *Refining Statistical Data on the Web*

19 Julia Efremova (TUE) *Mining Social Structures from Genealogical Data*

20 Daan Odijk (UVA) *Context & Semantics in News & Web Search*

21 Alejandro Moreno Célleri (UT) *From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground*

22 Grace Lewis (VU) *Software Architecture Strategies for Cyber-Foraging Systems*

23 Fei Cai (UVA) *Query Auto Completion in Information Retrieval*

24 Brend Wanders (UT) *Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach*

25 Julia Kiseleva (TUE) *Using Contextual Information to Understand Searching and Browsing Behavior*

26 Dilhan Thilakarathne (VU) *In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains*

27 Wen Li (TUD) *Understanding Geo-spatial Information on Social Media*

28 Mingxin Zhang (TUD) *Large-scale Agent-based Social Simulation - A study on epidemic prediction and control*

29 Nicolas Höning (TUD) *Peak reduction in decentralised electricity systems - Markets and prices for flexible planning*

30 Ruud Mattheij (UVT) *The Eyes Have It*

31 Mohammad Khelghati (UT) *Deep web content monitoring*

32 Eelco Vriezekolk (UT) *Assessing Telecommunication Service Availability Risks for Crisis Organisations*

33 Peter Bloem (UVA) *Single Sample Statistics, exercises in learning from just one example*

34 Dennis Schunselaar (TUE) *Configurable Process Trees: Elicitation, Analysis, and Enactment*

35 Zhaochun Ren (UVA) *Monitoring Social Media: Summarization, Classification and Recommendation*

36 Daphne Karreman (UT) *Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies*

37 Giovanni Sileno (UVA) *Aligning Law and Action - a conceptual and computational inquiry*

38 Andrea Minuto (UT) *Materials that Matter - Smart Materials meet Art & Interaction Design*

39 Merijn Bruijnes (UT) *Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect*

40 Christian Detweiler (TUD) *Accounting for Values in Design*

41 Thomas King (TUD) *Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance*

42 Spyros Martzoukos (UVA) *Combinatorial and Compositional Aspects of Bilingual Aligned Corpora*

43 Saskia Koldijk (RUN) *Context-Aware Support for Stress Self-Management: From Theory to Practice*

44 Thibault Sellam (UVA) *Automatic Assistants for Database Exploration*

45 Bram van de Laar (UT) *Experiencing Brain-Computer Interface Control*

46 Jorge Gallego Perez (UT) *Robots to Make you Happy*

47 Christina Weber (UL) *Real-time foresight - Preparedness for dynamic innovation networks*

48 Tanja Buttler (TUD) *Collecting Lessons Learned*

49 Gleb Polevoy (TUD) *Participation and Interaction in Projects. A Game-Theoretic Analysis*

50 Yan Wang (UVT) *The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains*

2017

1 Jan-Jaap Oerlemans (UL) *Investigating Cybercrime*

2 Sjoerd Timmer (UU) *Designing and Understanding Forensic Bayesian Networks using Argumentation*

3 Daniël Harold Telgen (UU) *Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines*

4 Mrunal Gawade (CWI) *Multi-core Parallelism in a Column-store*

5 Mahdieh Shadi (UVA) *Collaboration Behavior*

6 Damir Vandic (EUR) *Intelligent Information Systems for Web Product Search*

7 Roel Bertens (UU) *Insight in Information: from Abstract to Anomaly*

8 Rob Konijn (VU) , *Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery*

9 Dong Nguyen (UT) *Text as Social and Cultural Data: A Computational Perspective on Variation in Text*

10 Robby van Delden (UT) *(Steering) Interactive Play Behavior*

11 Florian Kunneman (RUN) *Modelling patterns of time and emotion in Twitter #anticipointment*

12 Sander Leemans (TUE) *Robust Process Mining with Guarantees*

13 Gijs Huisman (UT) *Social Touch Technology - Extending the reach of social touch through haptic technology*

14 Shoshannah Tekofsky (UVT) *You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior*

15 Peter Berck (RUN) *Memory-Based Text Correction*

16 Aleksandr Chuklin (UVA) *Understanding and Modeling Users of Modern Search Engines*

17 Daniel Dimov (UL) *Crowdsourced Online Dispute Resolution*

18 Ridho Reinanda (UVA) *Entity Associations for Search*

19 Jeroen Vuurens (UT) *Proximity of Terms, Texts and Semantic Vectors in Information Retrieval*

20 Mohammadbashir Sedighi (TUD) *Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility*

21 Jeroen Linssen (UT) *Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)*

22 Sara Magliacane (VU) *Logics for causal inference under uncertainty*

23 David Graus (UVA) *Entities of Interest — Discovery in Digital Traces*

24 Chang Wang (TUD) *Use of Affordances for Efficient Robot Learning*

25 Veruska Zamborlini (VU) *Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search*

26 Merel Jung (UT) *Socially intelligent robots that understand and respond to human touch*

27 Michiel Joosse (UT) *Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors*

28 John Klein (VU) *Architecture Practices for Complex Contexts*

29 Adel Alhuraibi (UVT) *From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT*

30 Wilma Latuny (UVT) *The Power of Facial Expressions*

31 Ben Ruijl (UL) *Advances in computational methods for QFT calculations*

32 Thaer Samar (RUN) *Access to and Retrievability of Content in Web Archives*

33 Brigit van Loggem (OU) *Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity*

34 Maren Scheffel (OU) *The Evaluation Framework for Learning Analytics*

35 Martine de Vos (VU) *Interpreting natural science spreadsheets*

36 Yuanhao Guo (UL) *Shape Analysis for Phenotype Characterisation from High-throughput Imaging*

37 Alejandro Montes Garcia (TUE) *WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy*

38 Alex Kayal (TUD) *Normative Social Applications*

39 Sara Ahmadi (RUN) *Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR*

40 Altaf Hussain Abro (VU) *Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems*

41 Adnan Manzoor (VU) *Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle*

42 Elena Sokolova (RUN) *Causal discovery from mixed and missing data with applications on ADHD datasets*

43 Maaike de Boer (RUN) *Semantic Mapping in Video Retrieval*

44 Garm Lucassen (UU) *Understanding User Stories - Computational Linguistics in Agile Requirements Engineering*

45 Bas Testerink (UU) *Decentralized Runtime Norm Enforcement*

46 Jan Schneider (OU) *Sensor-based Learning Support*

47 Jie Yang (TUD) *Crowd Knowledge Creation Acceleration*

48 Angel Suarez (OU) *Collaborative inquiry-based learning*

2018

1 Han van der Aa (VU) *Comparing and Aligning Process Representations*

2 Felix Mannhardt (TUE) *Multi-perspective Process Mining*

3 Steven Bosems (UT) *Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction*

4 Jordan Janeiro (TUD) *Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks*

5 Hugo Huurdeman (UVA) *Supporting the Complex Dynamics of the Information Seeking Process*

6 Dan Ionita (UT) *Model-Driven Information Security Risk Assessment of Socio-Technical Systems*

7 Jieting Luo (UU) *A formal account of opportunism in multi-agent systems*

8 Rick Smetsers (RUN) *Advances in Model Learning for Software Systems*

9 Xu Xie (TUD) *Data Assimilation in Discrete Event Simulations*

10 Julienka Mollee (VU) *Moving forward: supporting physical activity behavior change through intelligent technology*

11 Mahdi Sargolzaei (UVA) *Enabling Framework for Service-oriented Collaborative Networks*

12 Xixi Lu (TUE) *Using behavioral context in process mining*

13 Seyed Amin Tabatabaei (VU) *Computing a Sustainable Future*

14 Bart Joosten (UVT) *Detecting Social Signals with Spatiotemporal Gabor Filters*

15 Naser Davarzani (UM) *Biomarker discovery in heart failure*

16 Jaebok Kim (UT) *Automatic recognition of engagement and emotion in a group of children*

17 Jianpeng Zhang (TUE) *On Graph Sample Clustering*

18 Henriette Nakad (UL) *De Notaris en Private Rechtspraak*

19 Minh Duc Pham (VU) *Emergent relational schemas for RDF*

20 Manxia Liu (RUN) *Time and Bayesian Networks*

21 Aad Slootmaker (OU) *EMERGO: a generic platform for authoring and playing scenario-based serious games*

22 Eric Fernandes de Mello Araujo (VU) *Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks*

23 Kim Schouten (EUR) *Semantics-driven Aspect-Based Sentiment Analysis*

24 Jered Vroon (UT) *Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots*

25 Riste Gligorov (VU) *Serious Games in Audio-Visual Collections*

26 Roelof Anne Jelle de Vries (UT) *Theory-Based and Tailor-Made: Motivational Messages for Behavior Change Technology*

27 Maikel Leemans (TUE) *Hierarchical Process Mining for Scalable Software Analysis*

28 Christian Willemse (UT) *Social Touch Technologies: How they feel and how they make you feel*

29 Yu Gu (UVT) *Emotion Recognition from Mandarin Speech*

30 Wouter Beek (VU) *The "K" in "semantic web" stands for "knowledge": scaling semantics to the web*

2019

1 Rob van Eijk (UL) *Web privacy measurement in real-time bidding systems. A graph-based approach to RTB system classification*

2 Emmanuelle Beauxis Aussalet (CWI, UU) *Statistics and Visualizations for Assessing Class Size Uncertainty*

3 Eduardo Gonzalez Lopez de Murillas (TUE) *Process Mining on Databases: Extracting Event Data from Real Life Data Sources*

4 Ridho Rahmadi (RUN) *Finding stable causal structures from clinical data*

5 Sebastiaan van Zelst (TUE) *Process Mining with Streaming Data*

6 Chris Dijkshoorn (VU) *Nichesourcing for Improving Access to Linked Cultural Heritage Datasets*

7 Soude Fazeli (TUD) *Recommender Systems in Social Learning Platforms*

8 Frits de Nijs (TUD) *Resource-constrained Multi-agent Markov Decision Processes*

9 Fahimeh Alizadeh Moghaddam (UVA) *Self-adaptation for energy efficiency in software systems*

10 Qing Chuan Ye (EUR) *Multi-objective Optimization Methods for Allocation and Prediction*

11 Yue Zhao (TUD) *Learning Analytics Technology to Understand Learner Behavioral Engagement in MOOCs*

12 Jacqueline Heinerman (VU) *Better Together*

13 Guanliang Chen (TUD) *MOOC Analytics: Learner Modeling and Content Generation*

14 Daniel Davis (TUD) *Large-Scale Learning Analytics: Modeling Learner Behavior & Improving Learning Outcomes in Massive Open Online Courses*

15 Erwin Walraven (TUD) *Planning under Uncertainty in Constrained and Partially Observable Environments*

16 Guangming Li (TUE) *Process Mining based on Object-Centric Behavioral Constraint (OCBC) Models*

17 Ali Hurriyetoglu (RUN) *Extracting actionable information from microtexts*

18 Gerard Wagenaar (UU) *Artefacts in Agile Team Communication*

19 Vincent Koeman (TUD) *Tools for Developing Cognitive Agents*

20 Chide Groenouwe (UU) *Fostering technically augmented human collective intelligence*

21 Cong Liu (TUE) *Software Data Analytics: Architectural Model Discovery and Design Pattern Detection*

22 Martin van den Berg (VU) *Improving IT Decisions with Enterprise Architecture*

23 Qin Liu (TUD) *Intelligent Control Systems: Learning, Interpreting, Verification*

24 Anca Dumitrache (VU) *Truth in Disagreement - Crowdsourcing Labeled Data for Natural Language Processing*

25 Emiel van Miltenburg (VU) *Pragmatic factors in (automatic) image description*

26 Prince Singh (UT) *An Integration Platform for Synchromodal Transport*

27 Alessandra Antonaci (OU) *The Gamification Design Process applied to (Massive) Open Online Courses*

28 Esther Kuinderman (UL) *Cleared for take-off: Game-based learning to prepare airline pilots for critical situations*

29 Daniel Formolo (VU) *Using virtual agents for simulation and training of social skills in safety-critical circumstances*

30 Vahid Yazdanpanah (UT) *Multiagent Industrial Symbiosis Systems*

31 Milan Jelisavcic (VU) *Alive and Kicking: Baby Steps in Robotics*

32 Chiara Sironi (UM) *Monte-Carlo Tree Search for Artificial General Intelligence in Games*