FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Guiding Monte Carlo Tree Search simulations through Bayesian Opponent Modeling in The Octagon Theory

Hugo Miguel Pimenta Lopes Fernandes

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Eugénio Oliveira (PhD)

Co-Supervisor: Pedro Nogueira (MSc)

July 18, 2013

Guiding Monte Carlo Tree Search simulations through Bayesian Opponent Modeling in The Octagon Theory

Hugo Miguel Pimenta Lopes Fernandes

Mestrado Integrado em Engenharia Informática e Computação

Approved by:

July 18, 2013

Abstract

Board games present a very challenging problem in the decision-making topic of Artificial Intelligence. Although classical tree search approaches have been successful in various board games, such as Chess, these approaches are still very limited by modern technology when applied to higher complexity games such as Go. In light of this, it was not until the appearance of Monte Carlo Tree Search (MCTS) methods that higher complexity games became the main focus of research, as solution perspectives started to appear in this domain.

This thesis builds on the current state-of-the-art in MCTS methods, by investigating the integration of Opponent Modeling with MCTS. The goal of this integration is to guide the simulations of the MCTS algorithm according to knowledge about the opponent, obtained in real-time through *Bayesian Opponent Modeling*, with the intention of reducing the number of irrelevant computations that are performed in purely stochastic, domain-independent methods. For this research, the two player deterministic board game *The Octagon Theory* was used, as its rules, fixed problem length and board configuration, present not only a difficult challenge for both the creation of opponent models and the execution of the MCTS method itself, but also a clear benchmark for comparison between algorithms. Through the analysis of a performed computation on the game-tree complexity, the large board version of the game is believed to be in the same complexity class of Shogi and the 19x19 version of Go, turning it into a suitable board game for research in this area.

Throughout this report, several MCTS policies and enhancements are presented and compared with not only the proposed variation, but also standard *Monte Carlo* search and the best known *greedy* approach for *The Octagon Theory*. The experiments reveal that a combination of *Move Groups*, *Decisive Moves*, *Upper Confidence Bounds for Trees (UCT)*, *Limited Simulation Lengths* and an *Opponent Modeling* based simulation policy turn a former losing MCTS agent into the best performing one in a domain with estimated game-tree complexity of 10^{293}, even when the provided computational budget is kept low. Under the tested conditions, the best found MCTS-based approach was capable of surpassing the best known *greedy* approach, increasing the baseline win rate in a performed *round-robin* tournament by approximately 26%.

Resumo

Os jogos de tabuleiro apresentam um problema de tomada de decisão desafiador na área da Inteligência Artificial. Embora abordagens clássicas baseadas em árvores de pesquisa tenham sido aplicadas com sucesso em diversos jogos de tabuleiro, como o Xadrez, estas mesmas abordagens ainda são limitadas pela tecnologia actual quando aplicadas a jogos de tabuleiro de maior complexidade, como o Go. Face a isto, os jogos de maior complexidade só se tornaram no foco de pesquisa com o aparecimento de árvores de pesquisa baseadas em métodos de Monte Carlo (*Monte Carlo Tree Search - MCTS*), uma vez que começaram a surgir perspectivas de solução neste domínio.

Este projecto de dissertação tem como objectivo expandir o estado de arte actual relativo a MCTS, através da investigação da integração de modelação de oponentes (*Opponent Modeling*) com MCTS. O propósito desta integração é guiar as simulações de um algoritmo típico de MCTS através da obtenção de conhecimento acerca do adversário, utilizando modelação de oponentes Bayesiana (*Bayesian Opponent Modeling*), com o intuito de reduzir o número de computações irrelevantes que são executadas em métodos puramente estocásticos e independentes de domínio. Para esta investigação, foi utilizado o jogo de tabuleiro deterministico *The Octagon Theory*, pois as suas regras, dimensão fixa do problema e configuração do tabuleiro apresentam não só um complexo desafio na criação de modelos de oponentes e na execução de MCTS em si, mas também um meio claro de classificação e comparação (*benchmark*) entre algoritmos. Através da análise de um estudo efectuado sobre a complexidade do jogo, acredita-se que o jogo, quando jogado na maior versão do tabuleiro, se encontra na mesma classe de complexidade do *Shogi* e da versão 19x19 do Go, transformando-se num jogo de tabuleiro adequado para investigação nesta área.

Ao longo deste relatório, diversas políticas e melhoramentos relativos a MCTS são apresentados e comparados não só com a variação proposta, mas também com o método básico de Monte Carlo e com a melhor abordagem *greedy* conhecida no contexto do *The Octagon Theory*. Os resultados desta investigação revelam que a adição de *Move Groups*, *Decisive Moves*, *Upper Confidence Bounds for Trees (UCT)*, *Limited Simulation Lengths* e *Opponent Modeling* transformam um agente MCTS previamente perdedor no melhor agente, num domínio com uma complexidade da árvore de jogo (*game-tree complexity*) estimada de 10^{293}, mesmo quando o orçamento computacional atribuído ao agente é mínimo. Nas condições testadas, a melhor abordagem baseada em MCTS encontrada superou a melhor abordagem *greedy* conhecida, aumentado o *win rate* obtido pela abordagem definida como base num torneio *round-robin* em aproximadamente 26%.

Acknowledgements

I would like to thank everyone who directly or indirectly contributed to the realization of this work.

In particular, I would like to thank both my supervisors, Prof. Eugénio Oliveira and Pedro Nogueira, for their continuous support throughout the course of this work. Furthermore, I would like to thank Prof. Eugénio Oliveira for introducing and motivating my passion for Artificial Intelligence during my years at the university, and Pedro Nogueira for proposing and guiding my work with great enthusiasm, availability and encouragement. I am certain this thesis is a much better work thanks to their supervision.

I would also like to thank Dan Huffman, the creator of The Octagon Theory, for inventing and clarifying my questions and doubts about the game.

Finally, I would like to thank all my family and friends for always listening to my ideas and for all the games we played together when trying to better understand the game.

Hugo Fernandes

"We don't stop playing because we grow old; we grow old because we stop playing."

George Bernard Shaw

Contents

List of Figures

LIST OF FIGURES

List of Tables

Abbreviations

AI	Artificial Intelligence
AMAF	All Moves As First
DFA	Deterministic Finite Automaton
EPH	Evolutionary Planning Heuristics
FAST	Feature-Average Sampling Technique
FUSE	Feature UCT SElection
FPU	First Play Urgency
GGP	General Game Playing
IPC	International Planning Competition
MAB	Multi-Armed Bandit
MAST	Move-Average Sampling Technique
MCTS	Monte Carlo Tree Search
MIP	Mixed Integer Programming
MRW	Monte Carlo Random Walk
PAST	Predict-Average Sampling Technique
POMDP	Partial Observable Markov Decision Processes
RAVE	Rapid Action Value Estimation
RTS	Real-time Strategy Games
TOT	The Octagon Theory
TSP	Traveling Salesman Problem
UCB	Upper Confidence Bound
UCT	Upper Confidence Bounds for Trees
Web	World Wide Web

Chapter 1

Introduction

This chapter introduces the context of the thesis. A general introduction to the targeted research area is first given, followed by the motivation and objectives of this work. In the end, a document outline is presented, briefly describing the contents of each chapter in the document.

1.1 Context

Since the early days of computing, scientists from various fields attempted to develop ways of enabling computers to solve complex problems that transcend the human brain directly [1] (i.e. problems that cannot be solved without the aid of computers). With this idea in mind, the concept of creating a highly intelligent brain quickly emerged, leading to the creation of a new academic research field: *Artificial Intelligence*. Decision-making has always been one of the most important branches in Artificial Intelligence (AI) [2], as intelligent agents should not only be able to gain knowledge, but also to use it to make autonomous decisions. Decision-making problems exist everywhere in the modern society [3]: from economical backgrounds [4], such as the underlying competition present in any industry, to sociological [5] and psychological [6] problems, such as interaction between human-beings, and even in entertainment environments, such as sports [7] and games.

Game theory [8] is a field of AI that consists in the applicability of decision-making strategies in scenarios of competition and cooperation, such as economic-based scenarios. The underlying entertainment factor that games provide and their clear rules turn them into a popular test-bed for any decision-making problem, as games provide a simple and efficient way for testing solutions: players either win or not. Although games can be classified according to several properties [9], one specific property is widely used to differentiate the problems underlying the games: *determinism*. A deterministic game is a game with perfect information, such as Chess or Go. In a deterministic game, players are aware of every aspect involved in the game-playing process, and are only limited by their intelligence, or luck, if applicable. In a non-deterministic game, the information is not perfect, and thus players only possess partial knowledge about the game, or their opponent, throughout the course of the game.

Board games present a simple and entertaining mean of competition between opponents, focused solely on the intelligence and decision-making capability of their intervenients when confronted with players of different characteristics and playing levels. Since board games were always popular throughout history, most work done regarding game-theory was accomplished on deterministic board games [10]. The main advantage in using board games for decision-making research is that they greatly vary in complexity (2.3.3), allowing for comparison between methods, depending on the complexity class [11] they fall under. Although many games such as Chess can be targeted by classical AI approaches [10], higher complexity ones, such as Go, cannot. As such, the focus of game theory was mainly in perfecting the classical AI approaches for games that could be solved. However, with the breakthrough of *Monte Carlo Tree Search* (MCTS) methods (3.2)[1], the main focus of research in the field of game theory moved from the already approachable games to higher complexity ones, as new solution perspectives emerged [12]. Most of the current research in AI is still applied to these games and methods, and new enhancements are constantly being proposed.

1.2 Motivation

Although Monte Carlo methods (3.1) have been used to solve complex numerical problems in various fields of science, such as Mathematics and Physics, their popularity has been on par with other search methods. However, with the appearance of MCTS methods and their success in higher complexity games, such as Go, when compared to the previously used methods [13], namely *Neural Networks* [14], *Temporal Difference Learning* [15] and standard Monte Carlo itself, everything changed. In fact, within one year of the birth of MCTS, the first version of the solver *MoGo* was able to surpass every other solver found in the literature for the 9x9 board version of Go [16]. A comparison between the approximate Elo Ratings[2] [17] of the best solvers used for 9x9 Go, and the year they were used in, is presented in table 1.1 [13].

Table 1.1: Comparison between the Elo Rating of the best developed solvers and their approaches for 9x9 Go, from the year 2006 to 2010

Year	Program	Approach	Elo Rating
2006	INDIGO	Pattern database, Monte Carlo simulation	1400
2006	GNU Go	Pattern database, α-β search	1800
2006	Many Faces	Pattern database, α-β search	1800
2006	NeuroGo	Temporal difference learning, Neural network	1850
2007	RLGO	Temporal difference search	2100
2007	MoGo	MCTS (RAVE variation)	2500

[1]Throughout this report, numbers between curly braces refer to a chapter or section included in the report.
[2]A method for calculating and estimating the relative strength of players in two-player games.

2007	CrazyStone	MCTS (RAVE variation)	2500
2008	Fuego	MCTS (RAVE variation)	2700
2010	Many Faces	MCTS (RAVE variation)	2700
2010	Zen	MCTS (RAVE variation)	2700

The procedure used for calculating Elo Ratings is based on the idea that the chance of a player beating another player is a function of the difference in the current rating of both players (see [17] for a more thorough description). With this, the probability of the best 9x9 Go solver that does not use a MCTS approach winning against the best Go solver (i.e. the probability of an agent with 2100 Elo rating beating an agent with 2700 rating) is estimated to be approximately 2% [17]. Although the advantage of MCTS methods is clear when applied to games, MCTS has been used in various domains, such as real-world planning, control and optimization problems, obtaining significant results (3.2.10).

At the present time, many complex games for which no competitive solutions had yet been found now have MCTS-based agents capable of playing them at world class level (3.2.9). However, most of MCTS research is still focused on deterministic board games with unknown game lengths, such as Go. While there is still room for improvement in this domain, the fact that most of the work done is domain-dependent allied to the large difference in length of the problems being solved hinders comparison between approaches. In light of this, research in other domains where specific domain knowledge is not as relevant and the length of the problem is kept consistent has been incited, as the effectiveness and comparison of different approaches becomes simpler and clearer.

1.3 Objectives

Based on the current state-of-the-art, it is clear that higher complexity games relying on variations of MCTS for decision-making are still far from having good, competitive solutions. This problem derives from the fact that MCTS methods require a large computational load to obtain successful results in complex domains, and the current technology is still far from meeting these requirements. In light of this, the most crucial factor of success in MCTS is the usage of adequate policies to minimize irrelevant computations and enhance the accuracy of the method.

Although many approaches have been studied and proposed in this field (3.2), namely during the *selection* step of the algorithm (3.2.1), most implementations found in the literature still rely on *random sampling* (3.2.3) for simulating playouts during the algorithm, leading to a large amount of unrealistic play styles and irrelevant computations. In order to diminish this problem and enhance the *simulation* step, several approaches based on guiding the playouts according to domain specific knowledge and heuristics have been researched. However, these approaches were found to lead the MCTS algorithm to steer away from its natural process, causing numerous problems (3.2.3). Although solutions and mechanisms to prevent these problems are constantly being proposed (3.2), there is still a clear lack of research when it comes to integrating solutions to non-deterministic games, such as Opponent Modeling (3.3), with deterministic ones, even though these approaches could prove to be an efficient alternative for guiding simulations in a MCTS scenario. In fact, the success of Opponent Modeling in games such as Poker (3.3) suggests that the same considerations could be used to analyze the opponent throughout the course of a game and simulate its actions according to its perceived strategy. With this, the main goal of this thesis is to study the viability of such integration, by using Opponent Modeling to guide the simulation step of MCTS in a fixed-length deterministic game, believed to be in the same complexity class of Shogi and the 19x19 version of Go (2.3). Hence, the main research questions targeted in this thesis are:

A1. *What is the game complexity of The Octagon Theory?*

A2. *Can an opponent model for the provided rule-set be accurately defined by extracting experimental results from played games between humans?*

B1. *How viable are the standard Monte Carlo Tree Search approaches when playing the game?*

B2. *Can Opponent Modeling and specific domain knowledge improve these approaches?*

1.4 Outline

This report is organized as follows. The first (current) chapter provides a general introduction to the topics addressed in this thesis. The second chapter presents the rules, complexity and previous work done regarding the game to be used as a case-study for these research questions (The Octagon Theory), comparing it to other popular games. The third chapter presents the literature review, including both the state-of-the-art and related work in this topic, as well as background information on the considered algorithms. In the fourth chapter, the approach proposed in this thesis and the considered MCTS approaches for comparison are presented. In the fifth chapter, the experimental results obtained during the parameter tuning and selection policy processes are presented and discussed, and a comparison between approaches is performed. Finally, the last chapter completes this report with the conclusions on the research done and future work perspectives.

Chapter 2

The Octagon Theory

In this chapter, the board game used as a test-bed throughout the thesis is first presented, followed by an analysis and comparison of its *state-space* and *game-tree* complexities. Finally, previous work done in this game context is discussed and domain-specific knowledge found during this research is demonstrated.

2.1 Introduction

The Octagon Theory (TOT) is a deterministic two-player board game available for *iOS* and *Web*. TOT presents an interesting challenge for AI, as its simple rules, when allied to the complex board configuration, turn it into a game in which attacking strategies and defensive strategies are very similar. This similarity greatly ramps the difficulty up, as obtaining models of players and characterizing moves as attacking and defending moves is a very difficult problem. The consistent length of the problem turns TOT into a promising test-bed for comparison between algorithms, as every solution perspective has to consider the same amount of moves in every match, regardless of the approach.

2.2 Rules

TOT is a turn-based *pushing* game in which two players fight for the control of an octagonal board by using a limited selection of pieces over a predetermined number of turns, pushing their opponent's pieces off the edges of the board (including a hole placed in its middle). Although TOT can be played in boards of three different sizes, the only changes in the rules related to the size of the board are the starting amount of pieces per player and the number of turns to be disputed until the game ends. The larger version of the board, used as focus of this research, is presented in Fig. 2.1.

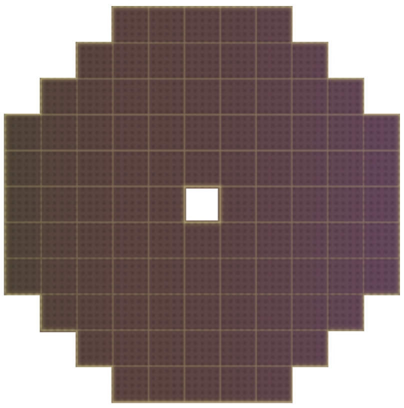

Figure 2.1: Large version of the board in The Octagon Theory

The winner of the game is the player who has more pieces on the board after all turns have been played. If both players have the same amount of pieces, the game ends in a tie. Although the game is essentially a fixed-length game, it is theoretically possible for a game played on the larger version of the board to end prematurely due to the higher ratio of turns to board positions. This situation only occurs if, at a certain point in the game, the acting player is unable to place any piece on the board (i.e. there are no free positions remaining). If this happens, the player with greater control of the board at that specific moment is declared the winner. However, it is worth noting that this situation is extremely unlikely, as it is only possible if both players purposely cause it, ignoring the outcome of the game in the process.

Each player has four different kinds of pieces with increasing pushing capabilities. The four pieces and their respective starting amounts for the large version of the board are presented in Fig. 2.2. Each piece is represented as an octagon and contains one or more straight lines from the center to the edges, representative of the piece's pushing capabilities. When a piece is placed on the board in any of the eight possible orientations (achieved by rotating the piece), any opponent pieces in neighbor positions pointed at by the placed piece are pushed back one position, provided the positions behind them are free or off the board. If a pushed piece does not have a free position behind it, the push occurs on the last piece of the stack of pieces, as long as all pieces belong to the player whose piece is being pushed. This kind of push is referred to as *cannoning*. An example of the *pushing* and *cannoning* processes that occur after placement of an *8-piece* (i.e. a piece with 8 pushing directions) on a board is shown in Fig. 2.3.

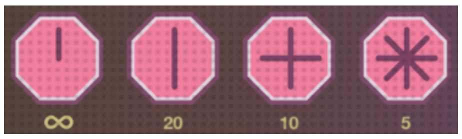

Figure 2.2: Starting amounts of pieces per player in The Octagon Theory

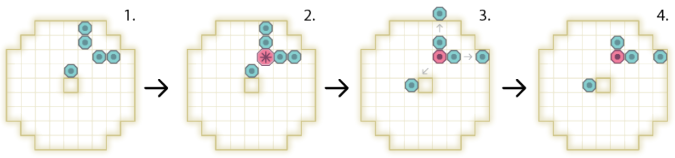

Figure 2.3: Example of the *pushing* and *cannoning* processes after placement of an *8-piece* in The Octagon Theory

2.3 Complexity

In order to gain some insight on which approaches should be explored in a TOT game-playing scenario, a study on both the state-space and game-tree complexity of the three versions of the board was conducted, enabling comparison with different games and domains found in the literature. These two metrics were chosen as they are the most commonly applied metrics when it comes to decision-based games [10], both relevant to the problem at hand. State-space complexity is more relevant in evaluation and classification scenarios, being strongly related with Opponent Modeling, while game-tree complexity is more relevant in search scenarios, strongly related with Monte Carlo Tree Search methods.

2.3.1 State-Space Complexity

The state-space complexity of a game represents the number of legal game states reachable from the initial one. If the state-space complexity is low, the likelihood of the same state to appear in multiple games is higher, enabling players to use previously gained knowledge when determining an action for a repeated state. In extreme cases with a very low state-space complexity, predefined moves can even be stored for any state. This situation is apparent in solved games such as Connect-Four [18]. In most classic board games, this kind of method is impossible, as the state-space complexity is too large [10].

In TOT, the initial game state is an empty board. As the game is played between two players *P1* and *P2*, each position of the board can be in one of three different states: occupied by a *P1* piece, occupied by a *P2* piece, or empty. Thus, for a given board with N positions, the state-space (SS) is given by:

$$SS = 3^N \tag{2.1}$$

The state-space complexity of the three board versions can be seen in Table 2.1.

As shown in Table 2.3, the state-space complexity of the large board version of TOT is similar to the one of Chess. Thus, finding equivalent board-states throughout the game is very uncommon, turning state detection and classification into an *impractical* problem with the current technology [10].

Table 2.1: State-space complexity of the three board versions in The Octagon Theory

Board version	Board size	State-space complexity
Small	36	10^{17}
Medium	68	10^{33}
Large	96	10^{46}

2.3.2 Game-Tree Complexity

The game-tree complexity of a game represents the number of nodes that need to be visited to determine the value of the initial game position through a full-width tree search [10]. A full-width game tree considers all the reachable nodes in any depth level. For most games, accurately computing (or even estimating) the game-tree complexity is a very difficult task [10]. As such, a rough estimate based on the average game length and branching factor is typically computed instead [10].

Since TOT is a fixed-length game (excluding the possible premature ending condition on the large version of the board mentioned in 2.2), the average game length is easily defined according to the used version of the board. However, the rules of the game, namely *pushing*, cause the branching factor to vary throughout the game, as the strategy of both players influences the number of possible moves in each turn. With this in mind, the game-tree complexity of TOT was estimated through a set of 100,000 simulated games between pseudo-random players (i.e. players that randomly bias their move selections) on each version of the board.

For each board size, 100,000 simulations were run. In each simulation, the number of unique moves (i.e. moves that leave the board in a unique state) per turn were recorded. The remaining (non-unique) moves were not considered in order to prevent an *artificial increase* of the problem complexity. The results of these simulations are displayed in Fig. 2.4.

As shown in Fig. 2.4, the branching factor initially grows in every version of the board, as more pieces emerge, leading to a higher amount of unique moves (e.g. by pushing a single piece in eight possible directions). However, while the branching factor stabilizes on the smaller versions of the board, as the board fills up and pieces start being pushed off more often, the larger version of the board suffers a decrease in unique moves as the game approaches the end. This decrease derives from the fact that the number of turns disputed on the large version of the board is much higher than on the lower versions (70 versus 20 in the smaller version and 30 on the medium one), causing a large amount of pieces to be blocked from further pushes as more pieces start being stacked together.

For each simulated game, the game-tree complexity of the game was also calculated. These results are shown in Table 2.2.

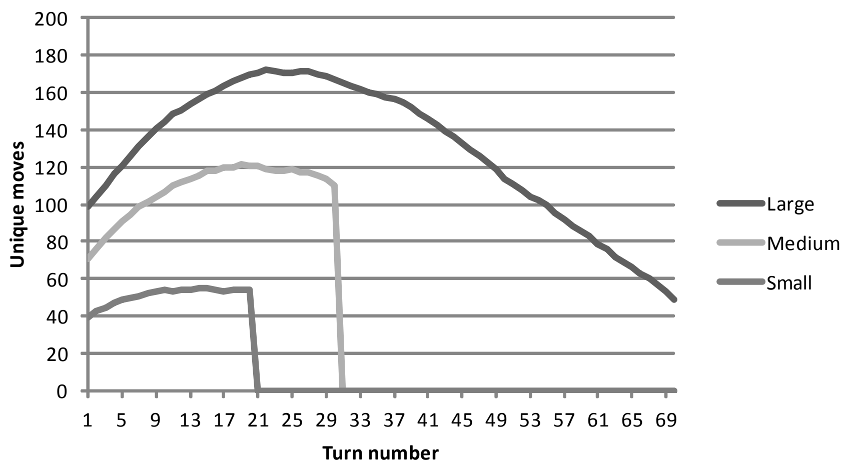

Figure 2.4: Average number of unique moves per play after 100,000 simulations in various versions of The Octagon Theory

Table 2.2: Game-tree complexity of the three board versions in The Octagon Theory

Board version	Board size	Game-tree complexity
Small	36	10^{69}
Medium	68	10^{121}
Large	96	10^{293}

2.3.3 Comparison with other popular games

In order to understand how complex TOT is, and what methods have been researched using games with similar complexities, a comparison of the board size, state-space complexity and game-tree complexity between the three board versions of TOT and other games was performed. The approximated results can be seen in Table 2.3.

Table 2.3: Comparison between The Octagon Theory boards and other popular games, ordered by Game-Tree complexity

Game name	Board size	State-space complexity	Game-tree complexity
Stratego	92	10^{115} [19]	10^{535} [19]
Go (19x19)	361	10^{171} [20]	10^{360} [10]
The Octagon Theory (large)	**96**	$\mathbf{10^{46}}$	$\mathbf{10^{293}}$
Thurn and Taxis (2 players)	56	10^{66} [21]	10^{240} [21]
Shogi	81	10^{71} [22]	10^{226} [22]
Go (13x13)	169	10^{79} [20]	

Amazons	100	10^{40} [23]	10^{212} [23]
Havannah	271	10^{127} [24]	10^{157} [24]
Abalone	61	10^{25} [25]	10^{154} [25]
Xiangqi	90	10^{48} [10]	10^{150} [10]
Chess	64	10^{47} [26]	10^{123} [26]
The Octagon Theory (medium)	**68**	$\mathbf{10^{33}}$	$\mathbf{10^{121}}$
Go (9x9)	81	10^{38} [20]	
Hex (11x11)	121	10^{57} [24]	10^{98} [24]
Gomoku (15x15)	225	10^{105} [10]	10^{70} [10]
The Octagon Theory (small)	**36**	$\mathbf{10^{17}}$	$\mathbf{10^{69}}$
Othello	64	10^{28} [10]	10^{58} [10]
Lines of Action	64	10^{23} [27]	10^{64} [27]
Nine Men's Morris	42	10^{10} [10]	10^{50} [10]
Fanorona	45	10^{21} [28]	10^{46} [28]
Qubic	64	10^{30} [10]	10^{34} [10]
Awari	12	10^{12} [10]	10^{32} [10]
Checkers	32	10^{20} [10]	10^{31} [10]
Congkak-6	42	10^{15} [24]	10^{33} [24]
Domineering (8x8)	64	10^{15} [24]	10^{27} [24]
Connect Four	42	10^{13} [10]	10^{21} [10]
Kalah	14	10^{13} [24]	10^{18} [24]
Pentominoes	64	10^{12} [24]	10^{18} [24]

By analyzing this table, it is clear that the game-tree complexity of the large board version of TOT is located in the upper range limit of complexities found in the literature. In fact, the large version of the board is located in between Shogi and the 19x19 version of Go, suggesting the large version of TOT belongs in the *EXPTIME-Complete* problems class, for which no solution perspective exists yet [11]. Furthermore, since TOT is a fixed-length game, the game-tree complexity is consistent across different games, unlike in games with varying game-ending conditions (e.g. a game of Chess can end after only four moves).

2.4 Previous Work

At this point in time, all known agents capable of playing TOT were developed by members of the *aigamedev* community and follow the same greedy approach, based on the official AI included in the TOT AI modders kit, described in [29]. This approach essentially consists in evaluating game states by crossing a weight matrix with each players pieces, selecting moves that lead to the maximum positional gain (or minimum loss) over the opponent in each turn. As such, for a given board-state (B), the chosen move is the move that satisfies the following condition:

$$\Pi_B = \underset{x\in[1,n],y\in[1,n],p\in[1,4],o\in[1,8]}{\arg\max} [f_W(B + a_{x,y,p,o}) - f_W(B)] \tag{2.2}$$

where n is the width/height of the board, a is the resulting action of placing a piece (p) with orientation (o) in a position of the board (x, y) and f is the *evaluation function* (i.e. the function that crosses a weight matrix (W) with a specific board-state). Despite its simplicity, the highest level AI (i.e. the AI with the best tuned W matrix found) is capable of defeating most human players.

2.5 Domain Knowledge

As TOT is still in its early stages in life, domain knowledge found so far is very limited. However, throughout this research, some interesting common patterns and considerations relative to the larger version of the board (focus of this study) emerged.

The most relevant finding is the player with the starting advantage. Prior to this research, the first player to move was believed as the one with an initial upper hand, even if such advantage was considered irrelevant. However, the first player to move was actually found to be in a considerable disadvantage. Out of all the performed experiments during this research, only 32.710% of the games were won as player 1, while 46.806% of them were won as player 2 (the remaining 20.484% were ties). This disadvantage derives from the fact that the first move of the game is the only move in the entire game that is not a *counter-move*. Although the importance of one single move in a game that is only finished after 140 moves might seem minimal, it was actually found to be a crucial factor of success between agents of similar level.

When played at high levels, the winner of a game of TOT is the player who is able to affect its opponent's pieces in such a way that their own pieces end up playing against their owner many turns down the road (e.g. by blocking the opponent from pushing pieces stacked behind them until the rest of the game). In fact, out of all the games recorded between the best performing agents, over 95% of the moves (excluding the initial one) were *pushes* and *kills* (i.e. pushes that throw a piece off the board). Furthermore, the amount of pieces placed in positions with no neighbouring pieces (here referred to as *free moves*) represent less than 1% of the total recorded moves. The remaining moves, referred here as *presses*, consist of pieces stacked against other pieces without pushing them (either by placing them against a player-owned piece, orienting them towards an empty space, or placing them against a stack of pieces including pieces from both players).

The countering nature of TOT was found to deprecate the usage of weighting matrices to estimate the value of each position of the board, when played at a higher level than the one proportioned by greedy agents. In fact, basing game-state evaluation at intermediate states of the game in MCTS-based agents according to such approaches was found to greatly diminish the agents' effectiveness, even when dealing with small computation budgets (i.e. lower thinking time per move). Out of all the games recorded between the best performing agents, the estimated value of every position of the board (here seen as the minimum amount of consecutive moves required by the opponent to push this specific piece off the board) consistently changed to irregular values (i.e. different values across different games). However, it is worth noting that the safest positions throughout the entire game tend to be the ones closer to the middle hole and on its line of attack (i.e. the line on the board that follows the required orientation to push a piece inside the hole), while the positions near the edges of the board typically become more important as the game progresses.

An interesting finding relative to the game complexity is that the game is actually much more restricted than it seems to be. If a player happens to make a mistake at an early stage of the game, when playing against a player of similar level, that mistake can incite a chain of actions that quickly leads to a *butterfly effect*. In fact, out of all the games recorded between agents with similar playing level, no agent was ever able to recover from a four piece or higher disadvantage, as its opponent was able to force *piece trades* until the game ended. This situation turns the selection and simulation policies of MCTS-based agents in a critical factor, as shown in the experiments (5).

Chapter 3

Related Work

This chapter presents the results coming out of the performed literature review on the domain of the problem focused by this thesis, providing an analysis on the related work, as well as background knowledge. For each discussed method, a global description is first given, followed by a description and analysis of the most relevant variations found in the literature.

3.1 Monte Carlo Methods

Monte Carlo methods are a set of algorithms used to approximate an unknown distribution through random sampling. These methods are typically used in very complex problems, when obtaining an exact solution with a deterministic algorithm is infeasible [30]. Although the general flow of the methods can be different, depending on the application domain, all Monte Carlo based methods follow the same steps.

Monte Carlo methods start by specifying the input domain (i.e. what is and is not possible to do). Once the domain has been established, inputs are randomly chosen, according to a uniform probability distribution over the domain. Afterwards, for each selected input, the outcome of the input is obtained through a deterministic computation of the results. By gathering and aggregating all the obtained results, a problem solution may be estimated. As with any probability-based method, the higher the uniformity and the chosen inputs, the better the estimation. One simple application of this kind of algorithm, could be, for example, determining the probability of obtaining heads as a result of a coin-flip. If someone flips a (non-rigged) coin randomly, the ratio between heads and tails will tend towards 50%. Obviously, if the coin is flipped only once, this ratio will either be 0% or 100%, but if the coin is flipped one million times, the ratio *should* be much closer. As this convergence to the real value of the distribution is the goal of any Monte Carlo method, these methods require a large amount of samples in order to be accurate.

One of the many applications of Monte Carlo methods ([31]) is solving (or playing) complex games, as firstly demonstrated by Abramson [32]. Games are essentially a set of states, in which transition between states is the outcome of the players actions (i.e. the inputs). Therefore, if a player knows the value of each input in each state, the game is solved. The problem with this approach lies in the fact that determining these values in higher complexity games is infeasible with the current technology. As such, the quality of a solver for these kind of games essentially represents the quality of its estimation. The idea behind Monte Carlo methods when applied to games is to play very large amounts of random games (i.e. produce random inputs throughout the various game-states), computing values for the quality of each move. In most board games, the result can be represented as a zero-sum result. Thus, the quality of a move can be estimated by considering the final result of every simulated game reached by following that (and any subsequent) moves.

Monte Carlo methods have some very strong advantages when applied to games, as shown by Sheppard in Scrabble [33] and Ginsberg in Bridge [34], who reached the top rankings of the world with such approaches. Since these methods are entirely based on random sampling, they do not rely on domain knowledge to estimate values of moves or in the exploration and estimation of nodes in very large trees, as in the case of the mainly used tree search methods. In fact, the only requirement for implementing a Monte Carlo based solver for any game is to implement the game rules. Once the solver knows the rules, it simply has to generate random moves among those rules to compute the results. However, Monte Carlo methods also have a great disadvantage compared to regular tree search methods. Since Monte Carlo methods are entirely based on random sampling, they have no knowledge of the game and tactics whatsoever [35]. The main problem with having no domain knowledge is that, while tree search methods recognize and classify moves based on what they can lead to (such as tactics), or how the game will play out, Monte Carlo methods simply select the move with the highest outcome, whether it is actually a good move or not [36], possibly making too many mistakes or even falling into traps.

3.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [37] is a product of the integration of Monte Carlo search methods with tree search methods. Therefore, MCTS is a best-first search algorithm that creates and asymmetrically expands a game tree based on the outcome of random simulations. Once enough samples have been drawn, a decision is made based on the estimated values of the tree nodes. As such, the effectiveness of MCTS algorithms is greatly dependent on the overall accuracy of these nodes, being higher as coverage of the tree increases (e.g. by increasing the number of performed iterations until a decision is made).

The basic MCTS algorithm is divided in four different sequential steps [38]:

1. *Selection*: Select an expandable node of the tree to explore.

2. *Expansion*: Expand the selected node by adding one or more child nodes to it and select a child node.

3. *Simulation*: Randomly simulate a game starting at the selected node until an ending condition is found.

4. *Back-propagation*: Back-propagate the result across the path followed on the tree.

These steps run continuously, for as long as they are allowed to, building and updating the nodes (i.e. states) and connections (i.e. actions) of the tree that is ultimately used for the *final selection* of the action to perform.

3.2.1 Selection

The first step in any MCTS algorithm is selecting a path for upgrading the tree. In this step, a child selection policy is recursively applied to the tree, starting at the root node (i.e. the node representing the current game state), until a leaf node is reached. Leaf nodes may either represent terminal states of the game or expandable nodes. Expandable nodes are all the non-terminal nodes in the tree that still require further expansion to reach a terminal state. An example of the selection of a node is presented in Fig. 3.1.

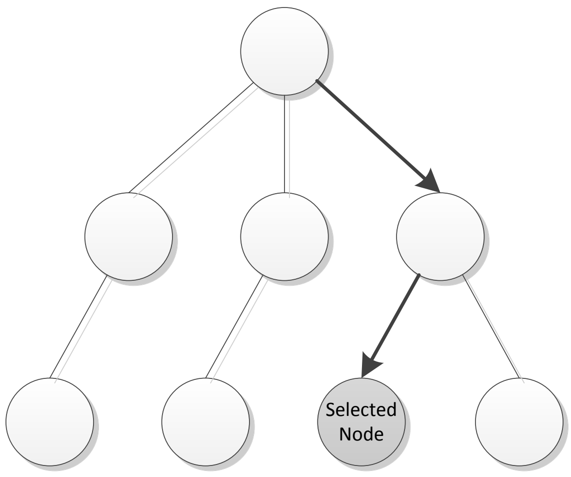

Figure 3.1: Selection step in a Monte Carlo Search Tree

The child selection policy depends on the implementation, and it controls the balance between exploration and exploitation, closely related to the *exploration-exploitation dilemma* [39] frequently encountered in *Reinforcement Learning* methods [40]. Exploration consists in selecting nodes with previously found bad scores to explore, possibly turning the bad scores into more favorable ones. Exploitation consists in selecting nodes with promising scores, in order to approximate their score to the actual value and increase the confidence of performing such actions. Balancing exploration and exploitation is a crucial task, as the score of a node is only as good as the process that led to it. If a node was not extensively explored (due to to a poorly sampled distribution), its score is most likely inaccurate, as it can lead to a large amount of child nodes with completely different outcomes. Several approaches have been researched in order to improve the selection policy.

Childs et al. [41] proposed the definition of *move groups* when selecting nodes in a tree. The idea behind move groups is that if not all nodes correspond to different moves (i.e. moves leading to different outcomes), they should be grouped together. Although Go was used for the original demonstration, the same principle can be applied to any game with similar possibilities. For example, in Tic-tac-toe, although the first player has nine positions to play a piece, these nine positions only represent three different moves (corner, edge or center), as the board could simply be rotated afterwards to obtain the same result. By grouping these nine positions in just three moves, the dimension of the MCTS game-tree is greatly reduced. A different scenario that can occur is reaching equivalent game states through different sequences of moves. Although the moves leading to the state are different, the ending result is the same. This situation is referred to as a *transposition*, and can also be approached as in the case of move groups [41].

One of the greatest challenges in the selection process is selecting the first moves. Since the tree has to be created, the first moves are bound to have the least amount of statistical information, as they are farther from the terminal nodes, having less time to compute results before a decision has been made. Gelly and Silver [42] investigated using offline functions to define a default selection policy by seeding the MCTS tree with previously computed information. As most games tend to start with similar moves, or at least avoid bad moves, having previous information on these moves can greatly speed up the initial process. Although using computed information keeps consistency with the method, the MCTS tree may also be seeded with heuristical information. Kozelek [43] successfully used historical knowledge from exploration done in previous games for the game Arimaa.

Besides using previously gathered information for the initial selection process, *opening books* were also used to improve initial performance in MCTS. The first sequence of moves in any game is called the *opening*. In most classic popular games, such as Chess, openings were extensively studied, and players typically start with popular ones (i.e. the move sequences that statistically lead to an early advantage). Opening books are essentially databases of known openings with proven value, and can even be generated by the MCTS itself [44, 45, 46], prior to playing. By following previously defined openings, the selection process can be focused on gaining deeper knowledge on the tree right from the beginning, choosing the first moves according to the known openings.

In a typical selection policy, nodes are usually visited at least once before starting to be explored. Although this information is valuable when choosing between exploration and exploitation, the required time to do so in complex games might be too large at first, greatly reducing the quality of the first moves, due to a lack of exploitation. *First play urgency* (FPU) is a modification to the selection policy of MCTS, proposed by Gelly and Wang [47] to minimize the risk of the first moves. With FPU, the selection policy initially considers unexplored nodes and explored nodes as different problems, boosting the importance of exploitation until a secure-enough move has been found.

As the selection policy is the process in charge of selecting what nodes should be explored, it is also the process in charge of finding *decisive moves* (i.e. moves that end the game) and *anti-decisive moves* (i.e. moves that prevent the opponent from ending the game) [48]. Although these moves can be found using pure sampling techniques [49], doing so might contradict the default selection policy, causing the algorithm to overlook them. In order to prevent this, decisive move analysis can be performed before the default selection policy is applied, at the cost of computational time. Teytaud and Teytaud [48] demonstrated a great performance increase for the game Havannah, even with the added computational cost.

Guillaume et al. [50] proposed a technique for guiding MCTS, called *progressive bias*, by adding domain specific heuristic knowledge to the selection policy. Progressive bias consists in balancing heuristic information with computed information, depending on the number of visits each node has. As the statistical information of a node is not admissible when it has too few visits, a higher importance is given to the heuristic value. As the number of node visits increases, the heuristic value progressively loses importance, while the statistical information gains accuracy and importance.

3.2.2 Expansion

Once a node has been selected, one or more child nodes are added to it, assuming the node is non-terminal and there are still unexplored paths. If the node is in fact terminal, or every path has already been covered, expansion is skipped, as the selection corresponds to an update, and not an expansion. An example of the expansion step is shown in Fig. 3.2.

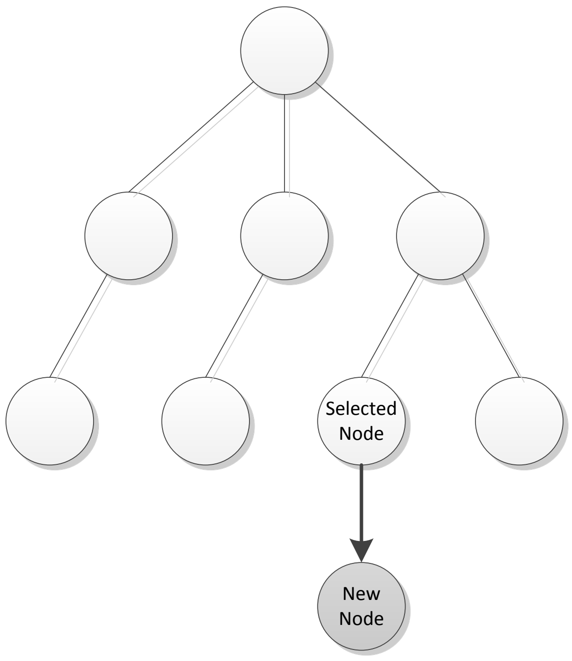

Figure 3.2: Expansion step in a Monte Carlo Search Tree

The only varying factor between implementations when it comes to the expansion policy is the number of child nodes to be added. This number typically varies according to the domain and computational budget, ranging from one single node to every possible node. Some implementations, such as *Mango* [50], a program for the game Go, vary the number of nodes to be expanded according to the number of node visits. The advantage of this approach is that if a node is consistently visited, depending on the domain, fully expanding it straightaway might outperform waiting for future iterations of the algorithm.

3.2.3 Simulation

Simulation is here intended as the task responsible for the computational simulation of an entire game, starting from the selected node. An example of the simulation step is displayed in Fig. 3.3.

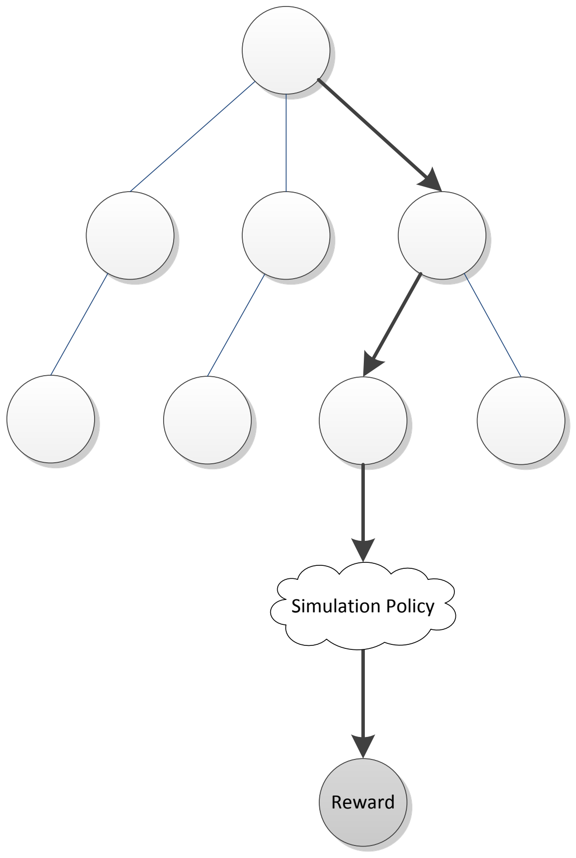

Figure 3.3: Simulation step in a Monte Carlo Search Tree

The default simulation policy in MCTS is the same as in regular Monte Carlo approaches: random sampling. The main advantage of this approach is that since it is simple and domain independent, its time complexity is inherently lower than domain knowledge based policies. Furthermore, since Monte Carlo methods draw samples from a random uniform distribution, the default simulation policy converges to full coverage of the game tree. However, random moves are not without a cost. The problem of random moves is that they do not model actions of the opponents accurately. As such, solely relying on random moves to simulate games is unrealistic, and leads to possible unnecessary computational costs. In order to solve this problem, several approaches have been researched with the intention of guiding MCTS by integrating domain knowledge in the simulation policy [51].

Gelly et al. [52] proposed integrating move patterns into the simulation policy, using the game of Go as an example. In certain domains, such as Go, players tend to perform consistent sequences of moves in a row after the first one has been applied. Clear examples of these moves can be taking pieces in Checkers, or exchanging material in Chess. Once a pattern begins, the following moves usually follow the pattern (unless it is dropped). As such, the simulation can be biased towards the moves within the pattern, instead of purely relying on random sampling. Coulom [53] extended this idea, by proposing the computation of Elo Ratings [17] for each recognized move pattern, also using Go as an example. By defining ratings for each move, the tree can further be guided by taking into account the rating of the opponent, as the likelihood of the opponent playing moves common to its rating is higher than playing patterns characteristic of weaker/stronger players.

One approach to guide the simulation is to inject heuristics based on the domain knowledge directly into the simulation policy. However, as discussed by Silver and Tesauro [54], these heuristics should be as simple and fast as possible, as the goal is not for them so solve the problem, but to roughly guide the simulation process. Silver and Tesauro also propose a method of *simulation balancing*, much like its selection counterpart, progressive bias [50]. Just as in the selection case, discussed in 3.2.1, simulation balancing gradually decreases the reliance in heuristics as the tree accuracy increases.

Rimmel and Teytaud [55] proposed using *tiles* as a domain independent improvement to regular simulation policies, obtaining good results for the game Havannah. Tiles are essentially groups of simulations that contain certain actions performed by the same player. By grouping all the simulations into tiles, an average reward value (i.e. expected gain of running the simulation) can be estimated. For example, in the case of Havannah, if two consecutive moves are executed by the same player in different simulations, those simulations are grouped into a specific tile and their average reward value is computed. When deciding on future simulations, those with actions contained in tiles that led to higher rewards can be prioritized, as they are expected to feed more knowledge to the tree.

Learning approaches can also be taken into account when defining a simulation policy. *General Game Playing* (GGP) [56] is a competition aimed at developing agents that can play games effectively just from their formal description (i.e. without human intervention). Björnsson and Finnsson [57] obtained world class level in this competition with a simulation policy capable of learning without the aid of domain knowledge, described as *Move-Average Sampling Technique* (MAST) [58]. MAST maintains discrete representations of every explored state and action, averaging the reward obtained by each action as simulations execute it. Subsequent simulations consider these values in order to bias the chosen actions towards those with better expectancy. Finnsson and Björnsson continued their research in this field [59], proposing two extensions to MAST, described as *Predict-Average Sampling Technique* (PAST) and *Feature-Average Sampling Technique* (FAST). PAST also considers the game state in which actions were taken when calculating their reward, instead of individual actions, such as in MAST, while FAST proposes the addition of a pre-learning method capable of extracting domain specific features and estimating their importance from the game definition.

3.2.4 Back-propagation

Once a simulation has been completed (i.e. has reached a terminal state), the simulation results need to be back-propagated, from the terminal state, to the first action in the sequence (i.e. the current root of the search tree). For each node that was visited in order to reach the terminal state, the back-propagation process increments its visit count and updates its average reward value, according to the outcome of the simulation. An example of the back-propagation of an obtained reward is presented in Fig. 3.4.

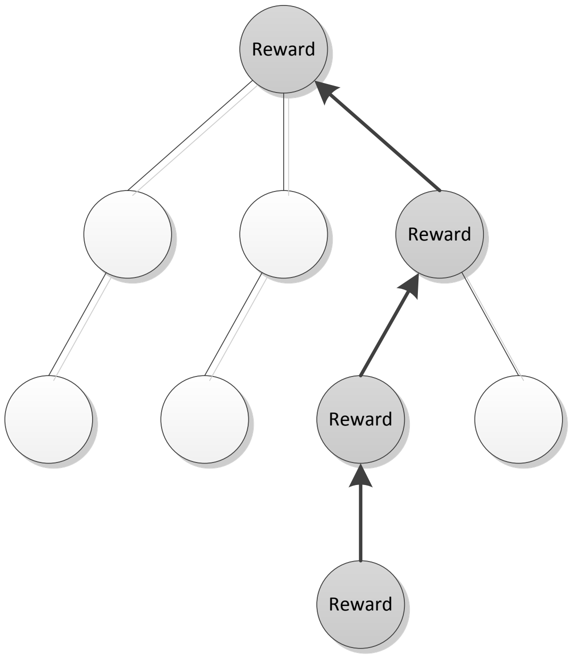

Figure 3.4: Back-propagation step in a Monte Carlo Search Tree

This reward value may be discrete in the case of zero-sum games, or continuous in games where final score is also important [21]. Although the reward value is deterministic, it can also be tweaked during back-propagation to improve future simulations of the tree.

Xie and Liu [60] observe that the importance of simulations varies throughout the course of the game. In fact, simulations done at an earlier time in the game tend to be less accurate than later ones, since they have to traverse considerably more nodes until a terminal state is reached, lowering accuracy. To counteract this, Xie and Liu propose the addition of a weighting factor to the nodes when back-propagating their results [60]. By increasing this weighting factor throughout the course of the game, nodes updated through simulations with higher accuracy gain more importance.

Tom and Müller [61] proposed the addition of a score bonus for better wins in discrete games. The idea behind this addition is that close wins usually come from states where both players were, to some extent, balanced (i.e. as close to win as the opponent). As such, close wins might represent a higher risk of play, as some unexplored nodes just before the win might turn it into a loss.

Kocsis et al. [62] proposed the addition of a decaying reward to the back-propagation step, alongside UCT (3.2.7). The decaying reward factor is a value that should be multiplied by every updated node, and decayed as the node proximity to the root of the search tree increases. By adding this weight, nodes leading to a win that are closer to the initial move are considered more important than later ones. Although this policy is aimed at games where early wins are considered better wins than later ones, implementing it in a fixed-length game might also be beneficial, as the accuracy of the nodes is closely related to their proximity to the root of the search tree.

3.2.5 Final selection

Once the computational budget is reached, the search process halts, and the best found action (i.e. the best child node) is selected, according to some criteria. There are four criteria for this selection process [21]:

- Max child: Select the root child node that produces the highest reward value.
- Robust child: Select the node with the highest visit count.
- Max-Robust child: Select the node with both the highest value and visit count. If no such node exists, resume the search process until a suitable node is found [63].
- Secure child: Select the node representing the lowest risk for the player.

As with any step, choosing the criteria for final move selection should be done according to the domain and goal of the MCTS process.

3.2.6 Move Pruning

Most games tend to have various kinds of actions, ranging from very good moves, such as winning moves, to very bad moves. Although humans naturally filter obvious bad moves immediately when playing, these moves do exist, and as such, are considered by any complete search tree, greatly increasing the search space with no added value. *Move pruning* is a technique for minimizing the exploration of such nodes, by cutting off any nodes corresponding to heuristically bad actions. By cutting these nodes, large amounts of search space can be spared, as each single node typically leads to a very large amount of nodes. The most popular method of move pruning is an extension of the original *Minimax* algorithm [64], called *alpha-beta pruning* [65], which greatly increases performance over the original method.

As MCTS methods typically target very complex domains and require a search tree for their basic functionality, addition of move pruning strategies to the base algorithm can be a crucial factor for attaining success. Although some pruning strategies can be implemented without needing any domain knowledge, having a reliable evaluation function for identifying potential bad moves is often needed. When such function does not exist, move pruning may be split in two different kinds [66]: *soft pruning* and *hard pruning*. The difference between soft pruning and hard pruning strategies is that, while hard pruning strategies permanently prune moves (i.e. permanently remove nodes from the tree), soft pruning ones only do so temporarily, allowing the moves to be re-evaluated at a later time.

Progressive unpruning and *progressive widening* are two similar soft pruning mechanisms, proposed by Guillaume et al. [50] and Coulom [53] respectively, for use in MCTS methods. The idea of using soft pruning instead of hard pruning, is that heuristic knowledge can be abused to immediately reduce the size of the search tree. Once the defined (pruned) tree starts increasing the accuracy of the nodes, progressive unpruning/widening unprunes some of the previously soft pruned nodes, eventually allowing exploration of every node, if enough computational budget is allowed. MoGo [16] successfully adopted progressive unpruning/widening strategies to gain a small boost in performance for the game Go.

Huang et al. [67] proposed two additional pruning strategies, described as *absolute pruning* and *relative pruning*, reporting an increase in performance on their Go program LinGo when using relative pruning. Absolute pruning consists in pruning every action of the tree instead of the most visited one, once it becomes clear that no other node could surpass the most visited one in number of visits. Relative pruning defines an upper bound for the number of visits to a node, pruning the remaining nodes once a node reaches the bound. As both strategies present a clear bias toward most visited nodes, any MCTS implementation that adheres to robust-child policies (3.2.5) can, theoretically, benefit from such strategies.

Domain based pruning techniques may also prove to be very useful, assuming they are accurate enough. Huang et al. [67] were able to use the concept of territory in Go to prune certain parts of the search tree, slightly increasing global performance of their Go program LinGo. He et al. [68] used domain specific knowledge for predicting the opponent's strategies in the game Dead End, increasing their win rate by over 50% when compared to their initial, already optimized, UCT (3.2.7) approach. This significant increase suggests that integration of Opponent Modeling (3.3) strategies may prove to be beneficial, even when applied to deterministic games.

3.2.7 Upper Confidence Bounds for Trees (UCT)

Upper Confidence Bounds for Trees (UCT) is currently the most popular MCTS algorithm, and it is widely used by computer Go programs. The most crucial factor for obtaining successful results with MCTS methods in very complex domains is a good exploration/exploitation ratio (3.2.1). UCT was proposed by Kocsis et al. [62, 69], as a MCTS method that uses an *Upper Confidence Bound* function (UCB1) as a tree policy.

The idea behind UCT is that, in any MCTS method, selecting moves to explore is essentially a *Multi-armed bandit* (MAB) problem. In a MAB problem [70], a gambler is faced with multiple slot machines, in which each machine has a limited amount of money (i.e. a reward value), and a specific distribution for returning the money. However, as the bandit does not know the value of the machines or their distributions, the average reward of each machine is unknown. The goal of the bandit is to, given a certain amount of initial money (i.e. the budget), maximize the rewards received from the machines. This problem produces a clear exploration/exploitation dilemma, as while the bandit wants to invest only in the machine with the highest average reward, any of the machines could be that machine. In MCTS, the selection of a move clearly follows this process. In this case, the MCTS player (bandit) is faced with multiple actions (slot machines), and wants to maximize its reward (wins) with an initial budget (computational budget). In order to do this, the player has to exploit the most promising actions, while also investing in less promising ones, to ensure a better action is not overlooked.

In a typical UCT strategy, the value of the nodes begins by being calculated according to the default selection policy [63]. Once the number of visits of a node crosses a predefined threshold, the calculation of its value swaps to a UCB1 based function. For a given set of child nodes reachable from a parent node j, the selected child node is the node i that maximizes the following equation:

$$UCT = \overline{X_i} + C\sqrt{\frac{\ln n_j}{n_i}} \tag{3.1}$$

where $\overline{X_i}$ is the normalized average reward of node i, n_j is the visit count for node j, n_i the visit count for node i and C is a positive constant. If more than one child node have the same UCT value, the winner is usually selected randomly [69]. The constant C is the *exploration constant*, and can be adjusted to increase or decrease the ratio between exploration and exploitation. Although a starting value of C = 2 is suggested in the literature, this parameter is typically experimentally tuned [49], as the optimal value is largely dependent on the domain, computational budget and the MCTS implementation.

3.2.8 All Moves As First (AMAF)

All Moves As First (AMAF) is a commonly used heuristic for enhancing MCTS algorithms, as it can be combined with most tree policies, such as UCT (3.2.7) [42]. In many games, such as Go, players can often play interchangeable sequences of moves leading to the same result. For example, if the player is deciding between three different positions for placing pieces on the board and none of those positions are occupied by the opponent, the player can place the three pieces sequentially in three different orders, obtaining the same result (i.e. the same game state). In most MCTS algorithms, these three moves would be updated differently, whether they were first played, or played at a later time, even though they reached the same state. The basic idea behind AMAF is to update every move as if it was the first. This way, if a move is played in a deeper position of the search tree but it is also a possible immediate move, the reward of the move is updated as if it was played first, since doing so is in fact possible. Depending on the domain, AMAF heuristics can prove to be very beneficial [13], and several variations have been researched for improving performance in specific cases [71].

The most popular AMAF heuristic is *Rapid Action Value Estimation* (RAVE), and it has been used to successfully improve the MCTS process in various games [72, 73]. RAVE is tuned to be used in conjunction with UCT (3.2.7), as it requires both the value sets (i.e. reward value and visit count) from UCT and AMAF for each node. By having these values, RAVE blends them together, obtaining the final reward value of each node. The blending process is similar to the α-AMAF heuristic [42], but considers the visit count of the updated node to balance the importance (i.e. weight) of both the UCT and AMAF values. As the visit count of a certain node rises, the score provided by the AMAF heuristic loses importance, eventually leading to full UCT. The rate of convergence to UCT depends on the domain and is typically tuned manually.

Although RAVE is typically formally applied to every node of the search tree, extensions for choosing candidate nodes for this computation have been researched. Lorentz [74] proposed a variation of RAVE, defined as *Killer RAVE*, in which only the nodes representing the most important actions are updated using RAVE. Tom [75] proposed an attempt of a more robust version of RAVE, defined as *RAVE-max*, by introducing a stochastic component to the calculation of the rewards of the nodes to avoid degenerate cases in some domains.

3.2.9 Applications in games

With the appearance of MCTS, the focus of research in games switched to higher complexity games. This switch created a new benchmark for AI [12]: *Computer Go* [76]. Although most research in this domain has been done by using Go as an example, several games, such as classical board games, modern single player games [21], multi-player games [77], and even real-time games [78, 79] have also been approached. However, the results [49] show MCTS is still clearly inferior to α-β pruning approaches when used in games with low enough complexity to be targeted by such methods.

The first Go program capable of beating a professional player in a handicapped match of the 9x9 version of the board was *MoGo* [16], using the RAVE variation of MCTS (3.2.8). The same approach was later improved for the Go program *Fuego* [80], leading to the first win against a professional player in an even match, as white (i.e. first to play). The first Go program to win a tournament while using MCTS was *CrazyStone* [81], by also approaching the problem with a AMAF variation of MCTS (3.2.8). Since then, RAVE approaches continued to be researched and improved, originating new, improved solvers [13].

Despite the clear bias towards deterministic games, MCTS has also been applied to non-deterministic games, namely Poker [82]. When applied to these problems, one extra level of nodes is added to the top of the tree, representing the possible non-deterministic cases as pseudo-actions [21]. Van den Broeck et al. [83] propose modifications to the selection and back-propagation strategies in MCTS methods to exploit the uncertainty of sampled values. Ponsen et al. [84] integrate Opponent Modeling (3.3) with MCTS, using previously learned models to guide regular UCT (3.2.7) in Poker.

3.2.10 Applications in non-game domains

Although most MCTS applications are game related, MCTS has been used for solving problems in various domains, such as mathematics, physics, economics, and even power management. The most common problems targeted by MCTS in these domains are planning, scheduling and optimization problems.

In planning problems, MCTS has been successfully used to learn plans in very complex problems where near-optimal solutions are known. Silver and Veness [85] proposed an algorithm for online planning in large *Partial Observable Markov Decision Processes* (POMDP), by combining the standard Monte Carlo method with MCTS. Gaudel et al. [86] proposed a MCTS variation, defined as *Feature UCT SElection* (FUSE), used for *Feature Selection* [87], by solving the problem as if it were a single player game. Walsh et al. [88] describe a mechanism for integrating MCTS with *Reinforcement Learning* [40], when faced with exponentially large state spaces.

The utility of MCTS in scheduling has been demonstrated by Nakhost and Müller [89], who successfully developed a *Monte Carlo Random Walk* (MRW) planner for all of the domains targeted by the *4th International Planning Competition* (IPC-4). Silver and Veness [85] obtained good performance on the *rocksample* problem (rock exploration in Mars), by using regular UCT (3.2.7). Pinson et al. [90] demonstrated how nested MCTS with memorization outperformed other methods for the *Bus Scheduling Problem* [91], when given enough computational budget. Chaslot et al. [92] demonstrated the usage of MCTS in *Production Management Problems*, outperforming *Evolutionary Planning Heuristics* (EPH) in every scenario, widening the gap as the complexity rose higher.

MCTS has also been applied in (combinatorial) optimization problems, obtaining near optimal solutions in various cases. However, scalability in these problems is still an issue with some variations. Rimmel et al. [93] used a nested MCTS with *time windows* in the *Traveling Salesman Problem* [94] (TSP), obtaining equivalent solutions to the typical optimization methods, albeit showing clear scalability problems in complex domains. Sabharwal et al. [95] demonstrated how a very highly optimized *Mixed Integer Programming* [96] (MIP) solver can be further optimized in several problems by using UCT (3.2.7).

3.3 Opponent Modeling

Opponent Modeling is the process of estimating an opponent's strategy (i.e. understand its play style and flaws), in order to obtain the upper-hand by predicting future actions and exploiting their weaknesses. In most (non-luck-based and non-solved) competitive games, two factors for winning the game exist [97]: knowing the game and understanding the opponent. The winner of any of these games is the player who is able to balance these two factors in a way that surpasses its opponent.

If a player is simply concerned about understanding the game, a weaker opponent might be able to exploit this and win. A clear example of this are most commercial Chess programs. Although these programs are capable of looking ahead like no human player can [98], by creating extensive game trees with α-β pruning approaches and performing millions of computations per second, a human who understands how they think can easily beat them because he knows its weaknesses. On the other hand, the same human might be a very bad player when facing other humans, because he does not understand them or possesses limited knowledge about the game. In fact, most professional players of any game, including sports, analyze their opponents before facing them, by watching how they played in previous games (e.g. what strategies did they use and what did they overlook), in order to find an opportunity to gain an advantage. This process is defined as building the *Opponent Model* [99].

Opponent Modeling has always existed as, even if subconsciously, it is the foundation of the human thought process in any sort of competition [97]. However, it was not until the 1990's that serious research in this domain started [100]. The sudden rise in interest was caused by the exploration of non-deterministic games, namely Poker. In fact, the lack of perfect information in non-deterministic games represents a clear Opponent Modeling problem. In these games, players know their state and their available actions, but they do not know if they are winning or not, because they do not know how well their opponents are doing. In order to estimate their position, players typically build models of their opponents and try to infer their previous actions in similar situations with the current situation. For example, in Poker, players do not know their opponent's cards, but if they know the players well enough, they *might* be able to perceive how well they are doing. This process is usually done by correlating factors that happened in previous similar situations (e.g. number of *calls* and *raises*) and their results (win, loss) with the current situation.

3.3.1 Building Opponent Models

Before knowledge about an opponent can be used to make decisions during a game, the opponent model has to be built. Creation of these models can be done exclusively *online* (i.e. as the game progresses), but most strategies also build them *offline* (i.e. before the actual game starts), in order to start the game with some knowledge of what can happen or not.

Online models are dynamic, and are built during the course of the game. Whenever the opponent executes an action, the player using Opponent Modeling classifies the action through some previously defined criteria (e.g. good, bad, aggressive, passive), and starts to build its own model of the opponent. As the opponent keeps playing, the model keeps being updated, and the player may use its information to guide its own actions. Although building an opponent model dynamically is fully accurate to the player's beliefs, as the model is solely based on the actual opponent, doing so requires extensive play before the gathered information is actually useful. This situation is very disadvantageous as players typically play few times against a large number of opponents, and not many times against one single opponent.

Offline models are static by nature, and are typically built through extensive analysis of various games, prior to playing. By analyzing a large amount of games between many different players, certain patterns begin to emerge. With these patterns, *user/player profiles* [101] can be built, differentiating various kinds of players. For example, in Poker, Sklansky [102] characterizes players according to their aggressiveness, number of played hands and number of bets/calls. The effectiveness of such models is greatly dependent on the quality of the data [103] used during the building process, and the used criteria for differentiating the actions in the domain. If a model to be used by a computer to play against other computers is built according to game logs between humans, the model might be inaccurate, as is it is based on opponents it will never play against. Moreover, if a model to be used when playing against very good opponents is based on game logs between weak opponents, the model will probably be unsatisfactory.

Building models, online or offline, is essentially a learning process. As such, any *Machine learning* [104] method can be used to autonomously learn opponent models.

3.3.2 Learning Opponent Models

Once a solid set of models has been defined, every opponent has to be identified according to the models. This identification process is essentially a *Machine Learning* process, in which a proximity to each player profile is defined for the opponent [105]. This proximity is important, as players do not necessarily fall under one single model. In fact, depending on the game flow, players can *jump* between models [97]. For example, an aggressive player that is on the verge of losing a game due to its aggressiveness might adopt a more defensive position to try and win the game back. By continuously running the learning process, players can adapt to these situations.

When a game starts, the opponent is typically considered as a *neutral player*. As such, the opponent is given an (equal) initial probability for every model. Every time the opponent makes a move, the current state of the game and the performed action are related to each model in the set, approaching the opponent to some models (i.e. raising the probability) while repulsing it from other models (i.e. lowering the probability). Depending on the method and the domain, further distinctions can be made, such as balancing the importance of actions according to the game state or the amount of moves played until the action was executed.

By analyzing the proximity between the opponent and the various models during a game, future moves may be predicted [106]. If no characteristics or models have been previously defined, *clustering algorithms* [107] may be used to identify and classify groups from the moves available to the player and relate them to future moves. However, in the case of most games, such as Poker, this process is typically accomplished by using a probabilitic function to approximate the model to a previously defined one. Although any probabilistic function can be used, variations of Bayes' theorem are typically chosen [97]. As the probability of executing certain actions is previously set for each defined model, calculating the probability of an opponent to execute a certain action is a matter of relating the agent with the various models and their inherent actions [108].

Baker and Cowling [109] use Bayes' theorem to evaluate the probability of a player utilizing a specific play style and use an *Anti-Player* (i.e. a player obtained *offline* to counter a specific play style) to choose the actions that were found to lead to a higher win ratio.

Southey et al. [110] propose a learning variation based on Bayes' theorem, found to quickly identify the player it is playing against from a set of opponent models containing games from the player. This suggests the method may be good enough to be used in real time against other players, as long as the models obtained *offline* are adequate.

Vidal and Durfee [111] introduce the *K-level agents* problem when learning certain multi-agent systems, closely related to Opponent Modeling. When a player is learning an opponent's strategy, there is no guarantee that the opponent is not also learning the player's strategy, basing its own strategy on the one pursued by the player. Moreover, if the player perceives this, its initial action might even have been a *trap*, tricking its opponent into thinking the player is following a certain strategy, when it is actually countering its learning process. This situation may happen recursively and on various levels, suggesting some players may fall under an *adaptive player profile*, as their strategy is essentially mutable throughout the game. Although players may follow more sophisticated learning policies to target these situations, their effectiveness is greatly influenced by the underlying domain and the player's certainty regarding its opponents' strategies [112].

3.3.3 Applications

The main application for Opponent Modeling is currently in non-deterministic games. However, general deterministic games have also been researched, by integrating Opponent Modeling with other heuristics or search methods.

Within non-deterministic games, the game that benefits the most from Opponent Modeling is Poker [113, 114], as it is essential to achieve high performance in this domain [115]. In fact, *Poki* [115], one of the strongest programs for Poker in the current state-of-the-art is entirely based in Opponent Modeling, using a *Neural Network* [116] for model learning and action prediction.

Del Giudice et al. [117] used Opponent Modeling for a scaled down version of the game Kriegspiel (partial observable Chess), obtaining good results in move prediction.

Richards and Amir [118] used Opponent Modeling for guiding simulations while playing the game Scrabble against other computers, obtaining a statistical improvement over a similar approach that does not use Opponent Modeling.

Schadd et al. [119] experimented with Opponent Modeling in *Real-time Strategy Games* (RTS), using the game Spring as an example. From the experiments, the approach was found to be capable of, for the most part, identifying the opponent's strategy fast enough (i.e. while it is still possible to counter it) from a simple set of strategies.

Carmel and Markovitch [120] attempted to use Opponent Modeling for handling encounters with other agents in multi-agent systems, by representing the problem as a repeated two-player game and learning simple *Deterministic Finite Automatons* (DFAs) to model the actions. However, this approach led to problems when learning minimal DFAs *autonomously* (i.e. without the aid of *teachers* to verify their correctness).

Chapter 4

Guiding MCTS through Opponent Modeling

This chapter presents the approach proposed in this thesis. The considered enhancements and variations in a MCTS scenario are first presented, followed by the used procedure when integrating Opponent Modeling during the simulation step.

4.1 Monte Carlo Tree Search

In order to evaluate the effectiveness of the proposed approach, the most relevant MCTS variations and enhancements found in the literature (3.2) for similar domains were first applied. This section presents the enhancements and variations that were considered for validating the effectiveness of the proposed approach (5) (i.e. the ones that were found to produce promising results), describing their usability in the The Octagon Theory.

4.1.1 Selection

The default selection policy used in this study was a *random selection policy* (i.e. a policy that randomly selects child nodes for exploration). Furthermore, regular UCT (3.2.7) was also considered, as it was shown to improve effectiveness of MCTS-based approaches regardless of the domain.

In TOT, every version of the board resembles an octagon (i.e. a regular polygon with eight sides). While the small version of the board is a regular octagon (i.e. an octagon with eight lines of reflective symmetry and rotational symmetry of eighth order), the remaining board versions are irregular octagons of equivalent opposite side lengths and rotational symmetry of fourth order. Thus, on the larger version of the board, the 96 positions can be defined from a pool of only 16 different ones, as shown on Fig. 4.1. In light of this, *move groups* (3.2.1) can not only be used to reduce the number of equivalent moves during initial stages of the game, but also extended to heuristically prune moves during the expansion phase (4.1.2).

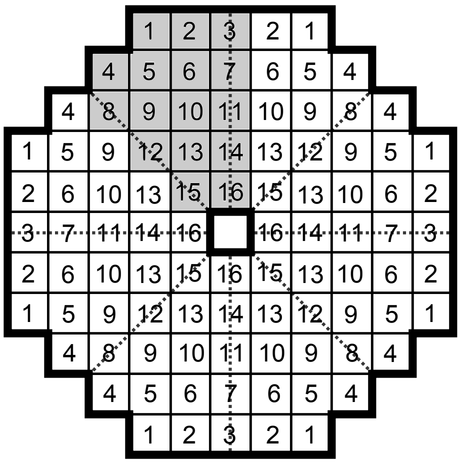

Figure 4.1: Symmetric positions on the large version of the board of The Octagon Theory

As TOT is a fixed-length game, *decisive moves* (3.2.1) can also be considered during the simulation step. In fixed-length games, nodes of the game tree representing decisive moves are all at the same level (i.e. as non-expandable leaves of the tree). Once the game reaches a certain point in time, fully determining *winning sequences* (i.e. sequences of moves that lead to decisive moves regardless of the opponent's moves) and *losing sequences* might be possible within the given computational budget, effectively replacing the standard selection policy until the end of the game. In a game-playing scenario, the computational budget is typically defined as the thinking time that is allowed per action, usually described in the rules of the game (or variant of the game).

4.1.2 Expansion

The default expansion policy used in this study was a *full expansion policy* (3.2.2). When using this policy, every possible unique move is added as a child node of the state leading to it.

Although *move groups* (3.2.1) were proposed as a selection policy enhancement, they can be extended further as an expansion policy enhancement when used as a *hard pruning* (3.2.6) technique. The combination of these two approaches is here referred to as *move abstraction*, and was the additional expansion policy researched.

As previously mentioned (2.5), over 99% of the recorded moves were *pushes*, *kills* and *presses*. Only less than 1% of the recorded moves were *free moves*, as players tend to place pieces next to other pieces. However, due to the size of the board, these moves actually represent the majority of moves in each turn. In light of this, if each *free move* is given the same importance as the remaining move types, the expansion policy will be clearly unbalanced towards the move type that is typically less performed. When using *move abstraction*, only one *free move* per symmetric board position (Fig. 4.1) is considered during expansion. Furthermore, if two different moves are *locally symmetric*, only one move is considered. In the case of pushes, three different positions have to be considered per pushed piece: the position in which the piece was placed (x_p, y_p), the starting position of the pushed piece (x_i, y_i) and the final position of the pushed piece (x_f, y_f). With this, two pushes $\{x_{p1}, y_{p1}, x_{i1}, y_{i1}, x_{f1}, y_{f1}\}$ and $\{x_{p2}, y_{p2}, x_{i2}, y_{i2}, x_{f2}, y_{f2}\}$ are locally symmetric if:

$$\begin{gathered}\{f_S(m_B(x_{p1},y_{p1})), f_S(m_B(x_{i1},y_{i1})), f_S(m_B(x_{f1},y_{f1}))\} \\ = \\ \{f_S(m_B(x_{p2},y_{p2})), f_S(m_B(x_{i2},y_{i2})), f_S(m_B(x_{f2},y_{f2}))\}\end{gathered} \tag{4.1}$$

where f is the *symmetric evaluation function* (i.e. the function that crosses a symmetry matrix (S) with a specific portion of the board) and m is the *sub-matrix function* (i.e. the function that returns the sub-matrix of a board B around position $\{x,\ y\}$). In this case, the considered sub-matrix was a 3x3 matrix, as it was found to offer the best gain to performance-loss ratio during an initial test phase. The observed performance-loss was attributed to the fact that, due to the size of the board (2.2), the amount of locally symmetric states of higher dimension represent a negligible portion of the state-space (2.3.1). However, the size of the sub-matrix can be increased depending on the rules of the domain. Furthermore, although the method also works in the case of *compound pushes* (i.e. multiple pushes per piece), these pushes were not considered, as the loss in performance of doing so for such an unlikely occurrence was also found to be too high. Instead, the only filter applied to *compound pushes* is a *minimal working piece* filter. With this, if placing a certain type of piece would cause multiple pushes, only the *lowest quality piece* (2.2) capable of performing the pushes is considered (e.g. the most powerful piece is only considered if there is at least one diagonal push and a straight push at the same time, or if there is a *compound push* that could be covered by a weaker piece and the player no longer has it).

4.1.3 Simulation

The default simulation policy used as a baseline in this study was *random sampling*. As such, during every simulation, moves from both the player and the opponent are randomly chosen.

When dealing with domains with large branching factors such as TOT, the length of the simulations (i.e. the number of nodes traversed until an end-game condition is met) may be too high to ensure sufficient coverage of the game tree in limited thinking times. Since low game tree coverage leads to less-informed decisions, the length of the simulations can be artificially reduced to boost the number of performed iterations. In TOT, for a given turn (n) of a game played on a board size with a number of turns (T_f) as its game-ending condition, the artificial ending turn (T_a) for a limited simulation length (L) is given by:

$$T_a = min(T_f, T_n + L). \tag{4.2}$$

If simulation policies that cause simulations to end in non-terminal states are used, the back-propagation policy can be adjusted to offer lesser reward values or to follow domain-specific heuristics to assess the potential of the state. However, for this study, no changes were made to the back-propagation policy when dealing with limited simulation lengths, in order to compare their raw effectiveness against unlimited ones.

4.1.4 Back-propagation

TOT is a zero-sum game with no added bonus on *heavier* wins (i.e. wins by a larger margin). As such, the chosen back-propagation policy was a simple discrete function with $V = \{-1, 0, 1\}$ for {losses, ties, wins}, as suggested in the literature (3.2.4).

4.1.5 Final Selection

As this research is focused on a domain with limited thinking time (as most games are), the *Max-Robust child* criteria (3.2.5) cannot be successfully applied without budgeting additional thinking time for additional search rounds. In light of this, only the remaining three criteria (*Max child*, *Robust child* and *Secure child*) (3.2.5) were considered.

4.2 Integrating Opponent Modeling

In a typical MCTS algorithm, the simulation policy is entirely random (3.2.3). While this policy does converge to full coverage of the game tree, leading to fully accurate decisions when enough computational budget is given, it also leads to very unrealistic playouts. Furthermore, since MCTS is targeted at problems that cannot be approached by traditional AI methods, MCTS-based solutions are not expected to be given enough computational budget to fully complete, as if such thing were possible in realistic time-spans, so would classical AI approaches, such as full search trees.

In TOT, the weakness of a fully random-based simulation policy is apparent. Given the large branching factor (2.3.2) and the amount of turns required in each game (2.2), a fully-random playout is very unlikely to mirror an actual playout (i.e. a sequence of moves that would actually be performed by an intelligent player). In light of this, most of the performed simulations are not only useless, but also detrimental to the global performance of the MCTS agent performing them. In fact, not only does the agent spend time and resources performing irrelevant simulations that would never be followed by their opponent, it also factors the outcome of these simulations in its decision-making process. For example, in a game of TOT, although *free moves* represent the majority of possible moves in each turn, they were found to be chosen less than 1% of the time (2.5). As such, not only would a typical (random) simulation policy spend most of its resources simulating playouts with very unlikely sequences containing a large number of *free moves*, but it would also consider the outcome of such playout as important as the one of a much more likely move sequence.

As previously mentioned (3.2.3), several simulation policy enhancements have been researched to diminish the unrealistic playouts problem, by biasing simulations with domain-specific knowledge, or even weighting the importance of the simulations according to domain-specific heuristics such as the *fitness* of intermediate game-states or proximity to terminal conditions. However, these approaches do lead to a problem commonly found in many traditional search approaches: *heavy playouts* (3.2.3). A clear example of this is the one of traditional *minimax* approaches. Although these approaches have been found to produce high level of play in games such as Chess, they are easily defeated by players who understand how they work and counter their traditional *best play* flow, which overlooks seemingly *bad plays*.

The proposed simulation policy enhancement described in this section focuses on guiding the simulations according to knowledge about the opponent obtained *online* (i.e. during the course of the game) and minimizing the number of unrealistic playouts. As the entire process should be kept as simple as possible (3.2.3) (i.e. it should only act as a guide for the MCTS process and not a solver on its own), the used model must be simple enough to be kept in all nodes of the tree, and the algorithm used to update such model according to the opponent's actions and estimated actions must be as simple as possible (i.e. *steal* as little computation time as possible from the MCTS algorithm). In light of this, the chosen approach was based on *Bayesian Opponent Modeling* (3.2.3).

4.2.1 Building the Opponent Model

When developing a mechanism capable of randomly biasing simulations according to the opponent's behaviour, the characteristics that are to be used when making such distinction have to be defined first.

In non-deterministic games, such as Poker, Opponent Modeling is typically used to predict future actions, as the amount of inputs (e.g. *call*, *raise*, *bet*, *fold*) is sufficiently low to base the entire algorithm around *player profiles*, using *Anti-Players* (3.3). However, in board games such as TOT, the number of inputs is considerably higher (2.2), turning move prediction into an unrealistic expectation. As such, moves have to be characterized according to general domain-specific considerations (i.e. generally followed in every game by every player).

In order to identify which parameters could be used to characterize moves and players in TOT, a series of game logs from a round-robin between 12 different players were specifically recorded. Out of these 12 players, 10 were human players, while the remaining 2 were simple greedy AIs: a defensive one and an aggressive one. The youngest player was 16 years old at the time, while the oldest player was 81 years old. The average age was 43.7 years (SD = 24.3) and the men to women ratio was 1:1. It is also worth noting that the players had no considerable past experience with board games (i.e. they were all casual players). Due to the length of the each game (2.2), only 1 game was recorded between each pair of players. However, all human players were given a *warm up* time in which they could play against each-other *off-the-record* to get used to the rules of the game and develop their own strategy. Although the outcome of the games was obviously recorded, the goal of this experiment was to obtain game logs to analyze how players react to certain situations and understand their general play style throughout the course of a game when facing different opponents. However, it is worth noting the two most experienced players were able to defeat every other player, including the AIs, and their own match was only decided by a 2 piece difference. One player was also able to predict the AIs strategy and defeat the defensive

AI, but was unable to defeat the aggressive one. The remaining players lost to both AIs, but every player won at least a game against another human player.

Through the analysis of the recorded game logs, it is clear that players primarily choose their moves according to the move type that better suits their strategy at each given moment. In fact, it is apparent that the proximity of the positions of the board to the edges and middle hole have no impact on the player's decisions throughout the majority of the game, as the *quality* of these positions is greatly dependent on the board-state (2.5). The only moment in which players were found to base their decisions according to the positions of the board was at the beginning of the game before any confrontation started. During this initial moment, defensive players were found to place pieces in positions distant to the edges of the board, without influencing their opponent's pieces, while aggressive players quickly started pushing pieces around. As soon as the first push occurred, every player swapped strategy from its perceived *opening strategy* to its general playing one.

In order to better understand the players' strategies, one game from each player was chosen, and an open interview about their thought process was conducted. In each interview, the game was replayed move-by-move, and the players were asked to explain what led them to perform the move they selected. When faced with a certain board-state, every player was found to follow the same thought process. This general thought process is presented in Fig. 4.2.

The first thing every player does when deciding which move to make is inspect the board for possible *immediate kills* and assess whether or not they would like to perform a *kill*. This decision is typically based on the player's score in relation to the opponent, the player's *aggressiveness* and *look-ahead*. This same process is iteratively followed for *pushes*, *presses* and *free moves*. The most interesting consideration regarding this thought process is that the players' strategies across different games can be largely defined by three single factors: the likelihoods of performing a *kill* or *push* when such movements exist, and the likelihood of *avoiding* proximity to other pieces when they do not. Throughout the course of the game, these characteristics fluctuate according to the player's standing. For example, when a player was found to be losing, its aggressiveness (i.e. likelihood of performing *kills* over other movements, or *pushes* if no *kills* existed) was found to rise higher as the game approached the end, in an attempt to turn the game around. Likewise, players who found themselves with a significant advantage towards the end of the game gradually steered away from *kills*, as maintaining the advantage was valued higher than performing possibly risky moves. In light of this, each player (M) can be characterized by the probability vector:

$$M = \{\phi, \gamma, \delta\} \tag{4.3}$$

where ϕ represents the probability of selecting a *kill* when such moves exist, γ represents the probability of selecting a *push* when such moves exist and *kills* are not considered (either because they do not exist or because they were discarded as an option), and δ represents the probability of selecting a *free move* over a *press* when all other moves were ignored, or nonexistent. This model is expected to perform well in a MCTS scenario, as it provides relative knowledge about a player

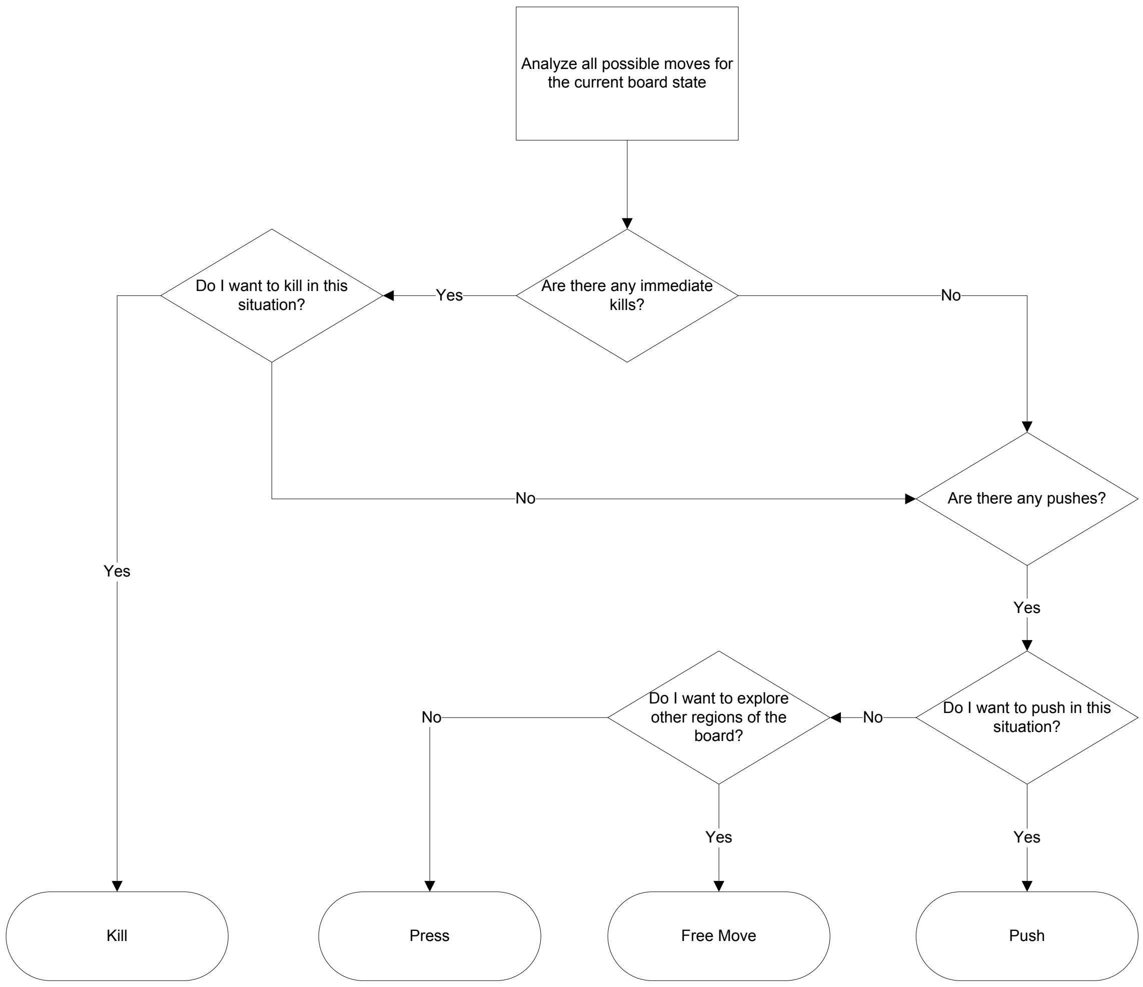
Analyze all possible moves for the current board state
Are there any immediate kills?
Yes
Do I want to kill in this situation?
No
Are there any pushes?
No
Yes
Yes
Do I want to push in this situation?
No
Do I want to explore other regions of the board?
No
Yes
Yes
Kill
Press
Free Move
Push

Figure 4.2: General thought process of players in The Octagon Theory

while keeping its simplicity (i.e. only three additional values must be kept per node of the tree).

4.2.2 Guiding MCTS simulations

At the start of the game, the opponent is given an initial player model $\{\phi_0,\ \gamma_0,\ \delta_0\}$. Although this model can be initialized as *neutral* (3.3), the effectiveness of the MCTS algorithm is naturally higher when the starting model is closer to the real model of the opponent. As such, the initial model can theoretically be tuned with past knowledge about the opponents (3.2.3) or through experimental *offline* matches against various agents when knowledge about the opponent is nonexistent.

The goal of the proposed mechanism is to guide the simulations of MCTS. As such, the model does not need to be precise if such precision requires additional computational budget that could be instead used to perform more simulations (3.2.3). Therefore, the model should require minimal space, as it will be stored in every node of the tree, and updating the model should be accomplished through a simple operation.

In the proposed mechanism, the model is directly used to infer the likelihood of a player performing a certain type of move. As such, the likelihood of performing each type of move at a certain point in time can be estimated if the model is accurate. With this in mind, the proposed mechanism uses a variation of *Bayes' theorem*. According to *Bayes' theorem* [121], the probability of a player following a certain model ($M = \{\phi,\ \gamma,\ \delta\}$) after performing a move of type ($\tau \in \{kill, push, press, freemove\}$) is given by:

$$P(M|\tau) = \frac{P(\tau|M)P(M)}{P(\tau)} = \frac{P(\tau|M)P(M)}{\sum\limits_i(\tau|M_i)P(M_i)} \tag{4.4}$$

where $P(\tau|M)$ is known from the model itself ($\{\phi,\ \gamma,\ \delta\}$). When a set of models is predefined, the presented equation can be used to increase - or decrease - the proximity of the player to each model $P(M)$ (3.3.2). However, in this case, predefining models would require the proximity to every model to be kept per node, causing the algorithm to run into memory problems for additional accuracy that is merely used as a guide. As such, the proposed mechanism keeps one single dynamic model at each node, and the model is updated according to the performed moves. With this, $P(M)$ represents the probability of the model that is kept by the player being correct, and $P(\neg M)$ the probability of the model being incorrect. Thus, the binary variation of *Bayes' theorem* [121] can be used to establish an approximation:

$$P(M|\tau) = \frac{P(\tau|M)P(M)}{P(\tau|M)P(M) + P(\tau|\neg M)P(\neg M)} \tag{4.5}$$

If the player is not found to play according to the model, the model is adjusted (i.e. altered to meet a higher value of $P(M)$), as the distribution of the simulations will be changed. Thus, in each turn (n), the following formula is used to recursively update the model:

$$P(M|\tau_n) = \frac{P(\tau_n|M)P(M|\tau_{n-1})}{P(\tau_n|M)P(M|\tau_{n-1}) + P(\tau_n|\neg M)P(\neg M|\tau_{n-1})} \tag{4.6}$$

As the player can still *jump* (3.3.2) through different models, a *certainty rate* (C) (i.e. how certain the player is that the action belongs to a model) is typically defined. By not ignoring the possibility of the opponent jumping, the algorithm will still focus a part of its process exploring nodes that are not likely to be followed by the opponent model. This situation is similar to the one found in RAVE variations (3.2.8), and it is employed to prevent the player from overlooking moves that would be played by anyone, even when the opponent model is not followed. When facing opponents with fixed strategies, the optimal value for the C constant is expected to be higher, as players do not roam around their pre-defined strategy. However, when facing mutable players (3.3), this value is expected to be considerably lower, as keeping it high leads to a bias towards a certain type of moves, misleading the player when such bias does not exist. Much like in the case of the initial model, this value can also be experimentally tuned according to the goals of the algorithm. As the certainty about a model (*P(M)*) steadily increases when an opponent is focusing on a certain play style, an upper bound of 0.95 is suggested in the literature [109]. By limiting this value, the agent can quickly recover (i.e. alter the opponent model) even if the opponent was not believed to ever change its playing style.

As the game progresses and the opponent performs more moves, the accuracy of the model increases. However, the model is actually useful right from the beginning, as it is not only used when the opponent makes a move, but during *every* move, whether the move was performed by the opponent or simulated by the player. During the normal course of an MCTS algorithm, several simulations are performed per turn. In each of these simulations, the proposed approach uses the model to bias the opponent's random move selection towards moves that fall under its characteristics. However, the model is not kept static throughout an entire simulation. Instead, each simulation keeps its own copy of the model, and updates the model according to the move selection. In fact, if the opponent were to play a move sequence $\{m_1, m_2, m_3\}$, its model would be updated accordingly. As such, it is only natural that the same process is applied during simulations. With this in mind, every simulation becomes increasingly more useful as it always follows a certain logic (versus an entirely random sequence of moves that would never be played by a player of any level). A simple example of this process and all the possible exceptions is shown in Fig. 4.3.

As shown in Fig. 4.3, the opponent model is constantly updated. When the simulation was started, the root node (i.e. the node representing the current state of the game) had an opponent model *PM0*, which was obtained throughout the course of the game. At this point, there are 3 different possible moves (e.g. push, free move, press). Since these moves are all different, performing one of them would cause the opponent model to be updated to either *PM1*, *PM2* or *PM3*. However, since the third move was selected, the opponent model used by the simulation becomes *PM3* from that point on. Since the following move is a player move (and not an opponent's move), no changes occur to the opponent model. In the last case, although the node corresponds to an op-

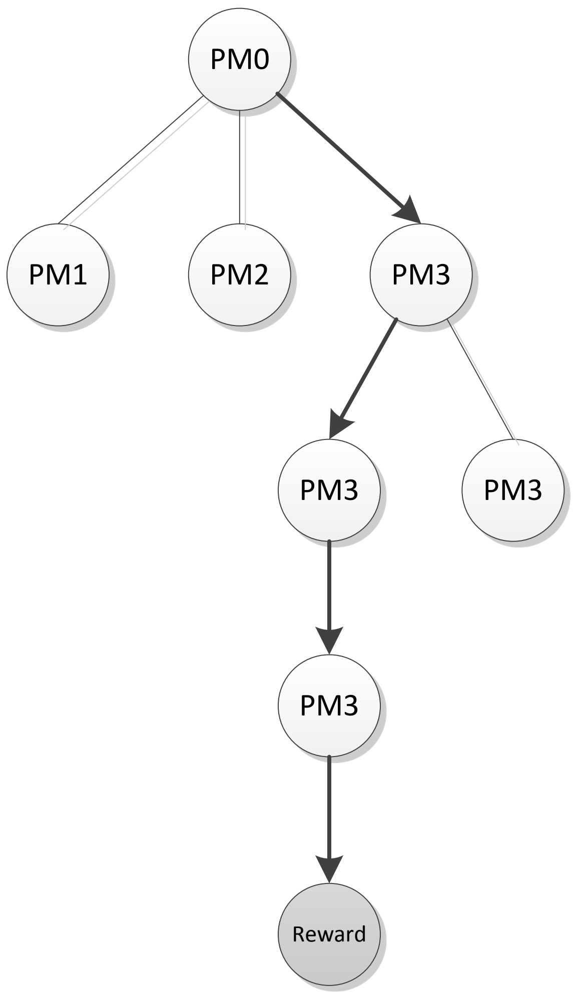

Figure 4.3: Example of a simulation policy using Opponent Modeling

ponent's move, the opponent is forced to play that move (i.e. there are no other possible moves). As such, the opponent model is also *PM3* as the move was not explicitly chosen, but forcibly played. Once the simulation ends and the outcome is determined, the MCTS algorithm continues and subsequent iterations will start at the root node once again (with opponent model *PM0*).

Chapter 5

Experimental Results

In this chapter, the results of the performed experiments for all the researched and developed agents, used to evaluate the effectiveness of the proposed approach (4) are presented and discussed. The experimental setup is first described, followed by the results of the experiments performed to tune the parameters and select the best policies. Finally, various approaches, including the best known (greedy) solver, are compared with every other approach through a round-robin tournament.

5.1 Experimental Setup

The following results were obtained on a single 3.4 *GHz* processor, with a limited maximum heap size of 8192 *MB* and no graphical accelerations or parallelizations.

For every experiment, 100 test games were conducted between each pair of agents under comparison. Out of these games, each agent played 50 of them as player 1 and 50 as player 2. In order to keep the results consistent with the literature, the computational budget was defined as the time required by a standard MCTS agent to perform 20,000 iterations per move on average throughout the course of a game. As such, the standard thinking time was set to 3 seconds on the machine used for the experiments, leading to 7 minute games. Although varying the thinking time and the number of test games in all the experiments could provide interesting results, these values had to be kept low to ensure enough experiments could be performed in a realistic time-span. Even so, the simulation process used to obtain the results presented in this section took longer than two months of continuous running, during which the machine was only periodically restarted to refresh the system resources.

5.2 Parameter Tuning

This section presents and discusses the performed experiments for tuning the developed policies (4) parameters.

5.2.1 UCT

As previously mentioned (3.2.7), the *exploration constant* C is typically experimentally tuned for the domain, and a starting value of $C = 2$ is suggested in the literature. Furthermore, it has been observed that decreasing this value (i.e. favoring exploitation) typically leads to better performance in domains with larger branching factors, while increasing it (i.e. favoring exploration) enhances performance in lower complexity ones [49].

Table 5.1 shows the win, loss and tie rates for different values of C against a baseline UCT agent with equal *exploration-exploitation ratio* ($C = 2$).

Table 5.1: UCT parameter tuning

C	0.2	0.4	0.6	0.8	1.0	1.5	2.5	3.0	4.0
Win	25%	40%	40%	38%	**46%**	26%	35%	26%	22%
Loss	30%	30%	27%	29%	**27%**	44%	40%	41%	53%
Tie	45%	30%	33%	33%	**27%**	30%	25%	33%	25%

As seen from the results, the importance of exploitation was found to be higher than that of exploration. The best found value for C was 1.0, winning 46% of the matches against the baseline agent, which was only able to win 27% of the matches. However, it is worth noting that values in the [0.4, 0.8] range also obtained promising results and could prove to be more efficient when facing different opponents (versus the baseline agent).

5.2.2 Limited Simulation Length

The simulation length parameter (L) (4.1.3) was tuned by following a similar process to the one used for tuning the UCT agent, albeit with the regular thinking time of 3 seconds per move (versus a number of fixed iterations).

When using the larger version of the board, TOT is played over 70 turns (i.e. 140 moves). As such, a regular MCTS agent has to traverse a maximum number of 140 tree nodes to reach the end of the game. However, this number linearly decreases as the game progresses. In light of this, different agents for every $L \in [100, 10]$ in increments of 10 were paired against their limit-free counterpart.

The results of this experiment, using the limit-free agent as baseline, are presented in Fig. 5.1.

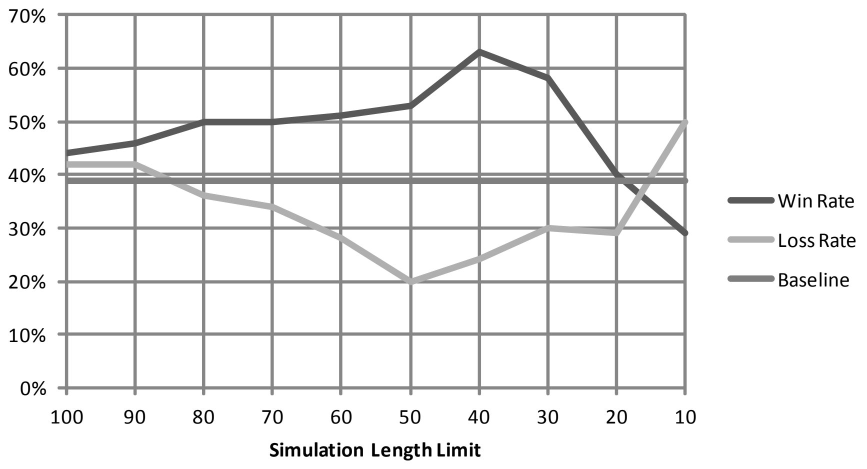

Figure 5.1: MCTS Limited Simulation Length parameter tuning

As shown in Fig. 5.1, the best found value for *L* was 40, winning 63% of the matches against its unlimited version, which was only able to win 24% of the matches. An interesting observation is the fact that while the win rate of the limited length agent rises steadily over the [100, 50] interval, it suffers a large performance loss right after peaking, reaching lower win rates and higher loss rates than the regular agent. This sudden decrease suggests that although the additional iterations boost the overall performance of the algorithm, limiting the length of the simulations too much essentially leads to a greedy behaviour. This causes the agent to lose considerably more often despite making more informed decisions regarding what it considers the end of the game.

In order to compare the effectiveness of limiting the length of the simulations in a MCTS scenario with a Monte Carlo one, the same process was followed to tune the value of (*L*) (4.1.3), using regular Monte Carlo agents. The results of this experiment are presented in Fig. 5.2.

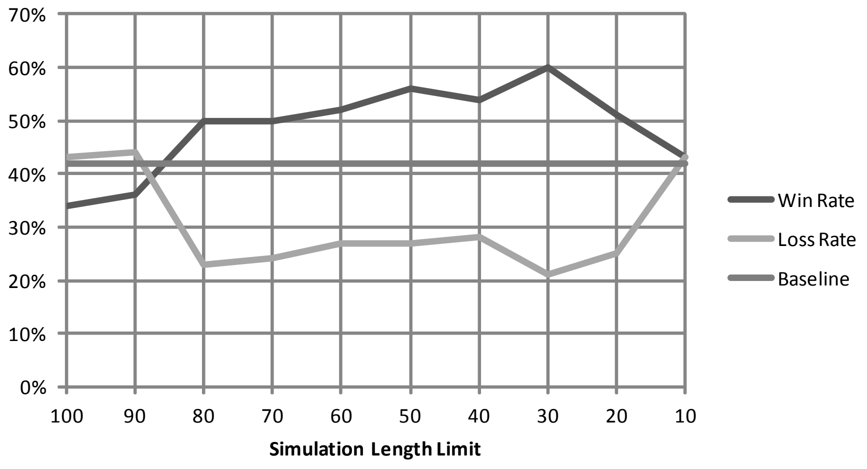

Figure 5.2: Monte Carlo search Limited Simulation Length parameter tuning

As shown in Fig. 5.2, limiting the length of the simulations of a standard Monte Carlo search produces similar results to those of a MCTS approach. However, the performance gain across the various lengths in this scenario is clearly lower than the gain found in the MCTS one. This lower performance difference suggests that since the MCTS agent keeps a search tree, the additional samples drawn from limited simulation numbers are better used and have a greater impact than in the tree-less version of the algorithm.

5.2.3 Opponent Modeling

As previously mentioned (4.2), both the initial opponent model ($\{\phi_0, \gamma_0, \delta_0\}$) and *certainty rate* C can be experimentally tuned according to the goals of the agent using Opponent Modeling for guiding the simulations of the MCTS algorithm. As the main problem of this thesis is to assess the effectiveness of such approach against a wide range of algorithms, both parameters were tuned for general game playing, by using the game logs obtained from the round-robin between 10 human players and 2 greedy AIs (4.2.1). However, it is worth noting that these parameters could be tuned towards a specific goal (e.g. defeat the majority of MCTS approaches) by ignoring additional behaviours.

The effectiveness of the integration of Opponent Modeling is naturally higher if the initial opponent model is closer to the opponent's real model at the start of the game. As such, the initial player model was defined by rounding the initial probability of selecting each particular move type at the first moment that the specific move type existed (e.g. the probability of selecting a *kill* the first time it is possible during the game). Additionally, since the first player forcibly starts with a *free move* (2.5), this move is ignored. However, it is worth noting that the same player can have a different strategy when playing first than when playing second, generating two different player models. In this case, as the sample is too small to produce any statistical significance, one single initial model was defined for both players. With this, the initial opponent model was defined as:

$$\{\phi_0, \gamma_0, \delta_0\} = \{0.75, 0.65, 0.5\} \tag{5.1}$$

When tuning the value of the *certainty rate* C, two aspects should be taken into consideration. On the one hand, the value of C should be able to quickly approximate the initial opponent model to the real (unknown) model. On the other hand, this value should not be too *aggressive*, as doing so might lead to unwanted behaviour (e.g. quickly assuming the opponent is going to use the same strategy throughout the entire game). With this in mind, the value of C was tuned by performing *cross-validation* [122] with two different sets of matches obtained during the *round-robin* (4.2.1).

After extensive analysis of the game logs obtained during the *round-robin* (4.2.1), two sets of five different matches where different strategies were employed were formed. For each match, an agent with the previously tuned policies and parameters (UCT and Limited Simulation Lengths) and the proposed simulation policy performed a normal round of the algorithm for each move of the game. For each move, the value of C regarding that move was considered a success if the resulting opponent model caused the coverage of the selected move by the opponent (i.e. the number of visits on the node that was ultimately selected by the opponent) to be in the 10% *top-end* of moves (i.e. in the 10% most covered moves). The *score* of the *C* value for one game was considered as the ratio between the number of successes and 70 (i.e. the sum of all moves). Thus, if a certain *C* value ensured the move selected by the opponent in each play was in the top 10% 35 times, its score for that value and player is 50%.

For each set of matches, the 10 obtained scores of *C* (5 for each player side) were averaged and used on the other set for *cross-validation*. The results of this experiment for the tested values of *C* are presented in Table 5.2.

Table 5.2: Opponent Modeling C parameter tuning

C	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Coverage	65%	71%	59%	72%	64%	**75%**	69%

As shown in Table 5.2, the best found value for *C* was 0.8. For this value, approximately 75% of the performed moves were represented in the top 10% of most visited nodes in each tested turn. Although there is no guarantee that this value is in fact the best value, due to reduced size of the used sample, the high coverage found across all values suggests that using an Opponent Modeling based simulation policy is a promising alternative to other policies.

5.3 Policy Selection

This section presents and discusses the performed experiments for electing which researched policies (4) should be followed.

5.3.1 Final Selection

The final selection policy (3.2.5) was tested by performing a round-robin between the three mentioned criteria. The results of the round-robin are presented in Table 5.3.

Table 5.3: Results of a round-robin of 100 rounds played between three different final selection criteria

Policy	Max Child	Robust Child	Secure Child
Win	43.0%	26.0%	10.5%
Loss	16.5%	28.5%	34.5%
Tie	40.5%	45.5%	55.0%

As shown in Table 5.3, *Max Child* was found to be the best of the three tested criteria, winning 43% of the matches. Although *Secure Child* presented the worst results, it was capable of forcing over 50% of the matches to end in a tie. This outcome suggests that using a hybrid final selection policy could prove to be an interesting strategy (e.g. by using one criteria to gain an advantage and a different one to hold it).

5.3.2 Decisive Moves

As previously discussed (4.1.1), *decisive moves* can effectively replace the standard *selection policy* in domains where the ending conditions are all at the same tree level, once the number of performed iterations in a single turn is high enough to cover the entire sub-tree without exceeding the provided computational budget.

For the established thinking time of 3 seconds per move, the implemented MCTS agent is capable of fully exploring the game tree once its length is equal or less than 3, as long as the branching factor stays below 88 on the last 3 moves. With this in mind, a standard MCTS agent was matched against an agent that uses the same selection policy as its opponent until the last 3 moves of the game, switching to a pure decisive move policy afterward, as long as the number of unique moves at that point is lower than 88. This additional condition was added due to the fact that, although the branching factor is estimated to be lower than that at the end of the game (Fig. 2.4), there is no formal proof that a game cannot reach a higher branching factor.

Unlike every other experiment, the test games for this experiment were started at turn 50 on board-states obtained from standard MCTS *mirror matches* (i.e. matches between the same agent) that led to a tie. This process was chosen as restricting the number of turns allows the results to be focused on the latest portion of the game, removing unnecessary *noise* that could otherwise exist (e.g. bad openings that condemned the entire match). The results of this experiment are presented in Table 5.4.

Table 5.4: Results of 100 games of a hybrid MCTS agent with decisive moves against a standard MCTS agent

	As Player 1	As Player 2	Total
Win	32%	78%	55%
Loss	38%	10%	24%
Tie	30%	12%	21%

As shown in Table 5.4, the addition of *decisive moves* was found to improve the performance of the agent when playing as player 2. Although the second player does have an advantage over the first one when playing a full game, this advantage is diminished by starting the games at later stages, as the board is no longer empty. The fact that the player does not benefit from the addition of *decisive moves* when playing as player 1 suggests that one single move (versus two moves as player 2) is not enough to make a difference at the end of the game. However, the positive results as player 2 suggest that both players should benefit from the addition of *decisive moves* as the computational budget increases.

5.3.3 Move Abstraction

In order to test the effectiveness of *move abstraction* (4.1.2), a simple Monte Carlo search agent with this added abstraction mechanism was paired against its raw version. The results of this experiment are presented in Table 5.5.

Table 5.5: Results of a Monte Carlo agent with added move abstraction versus a standard Monte Carlo agent

	As Player 1	As Player 2	Total
Win	32.00%	52.00%	42.00%
Loss	44.00%	32.00%	38.00%
Tie	24.00%	16.00%	20.00%

As shown in Table 5.5, the addition of *move abstraction* appears to increase the global performance of the agent. Although this increase is relatively low, the fact that it exists suggests that the additional computations required when determining which moves to prune provide enough value to the agent over the course of the game. In order to gain further insight on the affect of these additional computations over the course of the game, the average number of iterations performed in each turn of the games leading to the results shown in Table 5.5 were registered. These results are presented in Fig. 5.3.

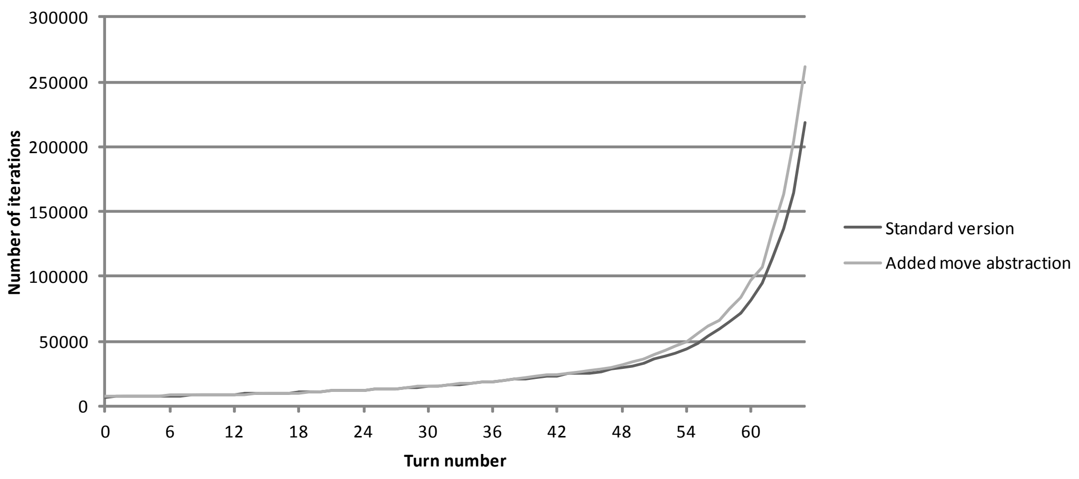

Figure 5.3: Average number of performed iterations per turn during the course a game by a standard Monte Carlo agent and the same agent with the addition of move abstraction

As shown in in Fig. 5.3, the average number of iterations performed per turn increased by approximately 11% during the first 136 moves. The final 4 moves were not considered, as both agents were using *decisive moves* (3.2.1), being able to determine the outcome of every possible move sequence until the end of the game at that point (i.e. within 3 seconds) without having to rely on simulations. Although both variations produce similar results for the most part of the game, the reduction in possible moves enables the agent with this addition to gain the upper-hand once the game is close enough to the end to ramp up the number of iterations per turn. The fact that this addition is significant even in the case of an agent with 3 seconds of thinking time and complete simulation policy (i.e. simulating every game until the end), suggests a greater increase in performance when dealing with larger computational budgets or limited simulation lengths. However, as both agents are still too similar and the sample size is error-prone due to its reduced dimension, the same experiment was performed in a MCTS scenario. The results of this experiment are presented in Table 5.6.

Table 5.6: Results of a MCTS agent with added move abstraction versus a standard MCTS agent

	As Player 1	As Player 2	Total
Win	24.00%	56.00%	40.00%
Loss	48.00%	20.00%	34.00%
Tie	28.00%	24.00%	26.00%

As shown in Table 5.6, the addition of move abstraction in a MCTS scenario also improved the global performance of the agent. However, the increase in performance in this scenario is greater than the one found in the Monte Carlo one. In order to compare the influence of such addition in the number of performed iterations per turn during the course of the game to the one of its Monte Carlo search counterpart, the average number of iterations performed throughout the games leading to the results shown in Table 5.6 were registered in the same manner. The results of this experiment are presented in Fig. 5.4.

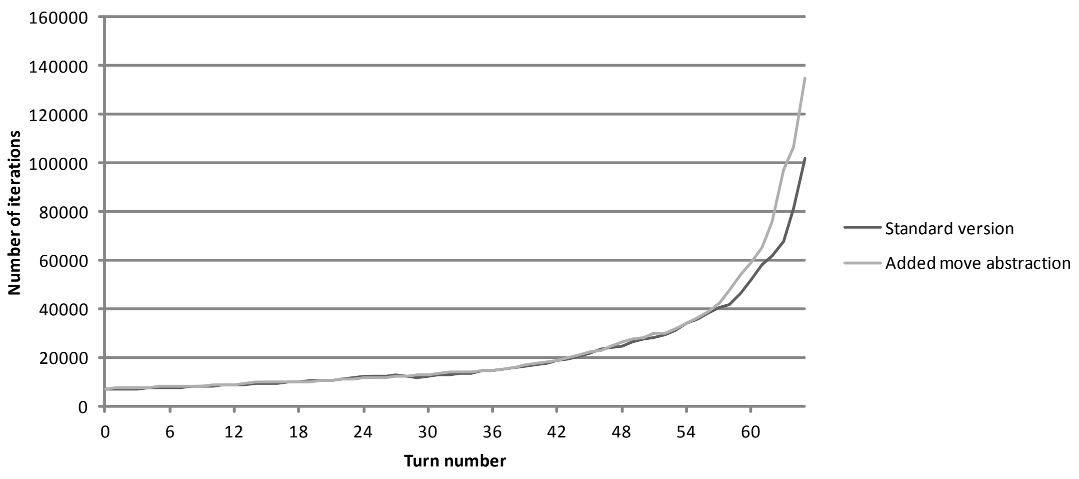

Figure 5.4: Average number of performed iterations per turn during the course a game by a standard MCTS agent and the same agent with the addition of move abstraction

As shown in in Fig. 5.4, the average number of iterations performed per turn increased by approximately 10% during the first 136 moves. Although both variations produce similar results for the most part of the game, as in the previously observed case, the increase in performed iterations towards the end of the game in the case of the agent with the *move abstraction* mechanism is once again superior to that of its raw version. In fact, the gap between both agents is even higher in the MCTS scenario than in the previous Monte Carlo search one. This difference occurs due to the fact that the actual move pruning process on the agent with the *move abstraction* mechanism enables a quicker expansion of the tree towards terminal states (i.e. the effective width of the tree starting at the current state is smaller), requiring less computational time when determining subsequent unique moves. As in the case of the Monte Carlo search experiment, this increase in performance suggests an even greater increase once the computational budget is raised or the length of the simulations is limited.

When comparing the average number of iterations performed in both scenarios (Figs. 5.3 and 5.4), an interesting observation can be made. Although the only difference between both approaches is the addition of a search tree in the case of MCTS, the average number of iterations performed per turn in this scenario decreased by approximately 30% during the first 136 moves. Even though the MCTS agent does not suffer a great loss in performance during the early phases of the game, the gap between both approaches becomes evident and progressively rises as the game approaches the end. This variation is directly related to the additional computations required by the search tree, which grows faster as the game progresses and the expansion to simulation ratio increases. In fact, at the start of the game, since one single simulation has to traverse 140 nodes to reach a result and back-propagate it across the entire tree, the computational time spent during the expansion process is extremely low when compared to the one spent performing the entire simulation, essentially turning the MCTS agent into a simple Monte Carlo agent who happens to spend a small amount of time expanding some nodes. However, as the game progresses, the amount of nodes that need to be traversed to reach a result greatly decreases, causing an increase on the frequency of the remaining operations of the agent, such as selection and expansion. Even so, the handicap created by the decrease in the number of performed iterations is surpassed by the insight passed onto the agent by the search tree, as the agent is still able to outperform the tree-less version (5.4).

5.4 Final Results

Once the parameters were tuned and the policies chosen, a round-robin with 100 rounds was conducted between the best found agents in each approach. For this experiment, the best known *Greedy* approach, standard Monte Carlo search (*MC*) and standard *MCTS* were included as baseline agents. For both Monte Carlo versions, agents with an added *L* adhere to the *Limited Simulation Length* policy, while agents with added *MA* make use of the *Move Abstraction* mechanism. Furthermore, every agent uses a *Max Child Policy* and *Decisive Moves*. The agent *UCT-OM* considers every enhancement discussed, including the integration of Opponent Modeling with MCTS proposed in this thesis. The results of the round-robin are presented in Table 5.7.

Table 5.7: Win rate of the various agents after a round-robin of 100 rounds

Agent	*UCT-OM*	*UCT-L-MA*	*UCT-L*	*Greedy*	*MCTS-L-MA*	*MC-L-MA*	*MC-L*	*MCTS-L*	*MCTS*	*MC*	Average win rate
UCT-OM	-	40%	47%	86%	67%	69%	72%	74%	88%	90%	70.33%
UCT-L-MA	32%	-	51%	75%	55%	55%	61%	57%	87%	92%	62.78%
UCT-L	35%	26%	-	69%	53%	58%	60%	52%	82%	85%	57.78%
Greedy	8%	10%	14%	-	55%	45%	61%	53%	78%	75%	44.33%
MCTS-L-MA	10%	20%	32%	30%	-	38%	41%	48%	55%	61%	37.22%
MC-L-MA	11%	7%	29%	30%	36%	-	42%	38%	50%	54%	33.00%
MC-L	8%	11%	24%	22%	38%	38%	-	36%	50%	57%	31.56%
MCTS-L	11%	17%	26%	24%	40%	25%	25%	-	48%	63%	31.00%
MCTS	0%	0%	0%	0%	14%	12%	10%	18%	-	42%	10.67%
MC	0%	0%	0%	0%	16%	9%	12%	17%	38%	-	10.22%

As shown in this experiment, every UCT version was able to surpass the best known greedy approach, even with the limited thinking time of 3 seconds. However, it is clear that the addition of Opponent Modeling in the simulation phase (4.2) greatly increased the performance of the agent over the typically used *random sampling* simulation policy (3.2.3). This addition leads to an interesting observation. While the performance of the agent was greatly increased when facing the best opponents, no significant increase was noticed when facing the standard versions of MCTS and Monte Carlo search and comparing it to the remaining UCT approaches. This indifference is to be expected, as while the addition of Opponent Modeling is capable of reducing the number of irrelevant simulations and perceive how the opponent plays, its usefulness is greatly diminished when facing opponents who do not seem to follow any particular strategy, as they do not follow any particular model (i.e. the agent cannot read a model which does not exist). In fact, since the standard approaches of MCTS and Monte Carlo search perform so poorly under these conditions (i.e. with such a limited thinking time), their playing style is highly mutable, as they cannot consistently run enough iterations to function properly.

The addition of *move abstraction* in limit-free agents was not found to produce a considerable increase in performance relative to their standard counterparts (5.3.3). However, its benefits are clear when coupled with a limited agent. Together, these two additions were able to improve every agent even further, especially in the case of UCT, which was already capable of greatly increasing the performance of MCTS on its own.

Although the number of performed simulations per match-up had to be kept low (5.1), it is worth noting that the results are consistent throughout the various agent pairings, establishing a clear progression towards the best found approach. This progression occurs not only when using the average win rate as metric, but also the win rate against the remaining approaches. As such, it is safe to assume that the UCT-based approaches surpass the best known greedy approach, and the best found approach is indeed the superior one.

Chapter 6

Conclusions

In this chapter, the conclusions of this research are presented. The initially defined research questions are answered according to the obtained results, and an outlook on open questions for future research in this area is given.

6.1 Goals

A1. *What is the game complexity of The Octagon Theory?*

In order to provide an answer to this research question, a comparison of the board size, state-space complexity and game-tree complexity between the three board versions of TOT and other well-known games was performed (2.3). Although determining the state-space complexity of TOT is a simple task, accurately determining the game-tree complexity is not, as the rules of the game were found to cause the branching factor to vary throughout the game. As such, the game-tree complexity of TOT was estimated through a set of 100,000 simulated games between pseudo-random players on each version of the board.

From the results, both the state-space and game-tree complexity of the large board version of TOT were found to be located in the upper range limit of complexities found in the literature (between Shogi and the 19x19 version of Go), with values of 10^{46} and 10^{293} respectively, suggesting the large board version of TOT belongs in the complexity class *EXPTIME-Complete*, for which no solution perspective exists yet [11].

A2. *Can an opponent model for the provided rule-set be accurately defined by extracting experimental results from played games between humans?*

Through extensive analysis of game logs obtained from a round-robin tournament between 10 human players and 2 greedy AIs (4.2), it is clear that, unlike most games found in the literature (3.2.9), the board configuration of TOT does not allow moves to be characterized according to the positions they affect or their placement on the board. In fact, basing game-state evaluation at intermediate states of the game in MCTS-based agents according to such approaches was found to greatly diminish the agent's effectiveness, and an alternative approach had to be researched.

By analyzing the state of the board in the various games, it is evident that in order to accurately determine the *quality* of the state, the *real distance* of every piece to the closest border (i.e. through a possible move sequence) has to be determined. However, as doing so adds a new level to the game playing algorithm (i.e. a path finding problem), modeling opponents in this manner is an unrealistic expectation in a MCTS scenario, as the goal of the model is to guide the *simulation* process without sacrificing the global performance of the algorithm. In light of this, a simpler immediate model representing the likelihood of the players to perform certain move types was obtained (4.2.1). Although this model is assumed to be too simplistic to act as a base of an entire Opponent Modeling approach, such as most Poker agents (3.2.9), it was found to provide sufficient knowledge at the expense of only three additional values per board-state, when used in a MCTS scenario.

B1. *How viable are the standard Monte Carlo Tree Search approaches when playing the game?*

Given the results of the final round-robin (5.4), both standard MCTS and Monte Carlo search approaches were found to perform poorly under the tested conditions (5.1). However, it is clear that this low performance derives from the fact that, as the provided thinking time is so limited, both agents cannot consistently run enough iterations to function properly.

Although standard MCTS was found to be one of the worst playing agents (i.e. with lower win rate and higher loss rate), the simple addition of domain-independent enhancements such as limiting the length of the simulations and the addition of a UCB1 selection policy (i.e. UCT) were able to greatly increase the performance of the agent, turning the former losing MCTS agent into one of the best found agents (5.4), surpassing the best known greedy approach (2.4).

B2. *Can Opponent Modeling and specific domain knowledge improve these approaches?*

Given the results of the final round-robin (5.4), both Opponent Modeling and specific domain knowledge were found to significantly enhance the performance of the best domain-independent approach.

Although the addition of domain knowledge during the *expansion* phase (4.1.2) in limit-free agents was not found to produce a considerable increase in performance relative to their domain-independent counterparts, its benefits are clear when coupled with a limited agent. Together, these two additions were able to improve every agent even further (5.4), especially in the case of UCT.

By using *Bayesian Opponent Modeling* to guide the simulations of the best performing agent (4.2), an even greater increase in performance could be obtained over the typically used *random sampling* policy (3.2.3). This significant increase suggests that the addition of Opponent Modeling to the *simulation* step of MCTS can indeed reduce the number of unrealistic simulations and perceive how the opponent plays, allowing the MCTS-based agent to make more informed decisions in each moment and reduce the number of irrelevant computations performed during an entire game.

6.2 Future Research

Although the proposed approach was able to turn a former losing MCTS agent into the best performing one, there is still a clear dependency on enhancements that aid the agent in the starting moments of the game, as the number of performed iterations per turn is lower and the branching factor keeps increasing. This suggests that the attribution of a game-based computational budget, instead of the researched turn-based one, could lead to an interesting challenge. Under these rules, the player would not only face a game theory problem, but also a resource allocation task when determining which moves should be prioritized (i.e. given more thinking time), depending on the course of the game. In addition, researching and using a hybrid final selection policy (5.3.1) could prove to be an interesting strategy under these conditions.

As the integration of Opponent Modeling in the *simulation* step of MCTS was found to improve the performance of the approach, many new questions regarding the used parameters (4.2) and the parameter tuning process used (5.2) emerged. Although an upper bound for the value of the *certainty rate* (C) is suggested in the literature as a mechanism of protection against *jumps* (3.3.2), this method is not exactly optimal. Considering the definition of an upper bound is only used to prevent these cases, an interesting research topic could be turning the value of C into a variable value, defining it according to the opponent model. With this, a possible improvement to the typical models suggested in the literature could be the addition of a *likelihood of jumping over time* factor, altering the *certainty rate* according to the state of the game and the perceived model of the opponent.

References

[1] P. McCorduck. *Machines who think: A personal inquiry into the history and prospects of artificial intelligence*. W.H. Freeman San Francisco, 1979.

[2] S.J. Russell, P. Norvig, J.F. Canny, J.M. Malik, and D.D. Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Englewood Cliffs, NJ, 1995.

[3] W. Edwards. The theory of decision making. *Psychological bulletin*, 51(4):380, 1954.

[4] H.A. Simon. Theories of decision-making in economics and behavioral science. *The American Economic Review*, 49(3):253–283, 1959.

[5] J.M. Eisenberg et al. Sociologic influences on decision-making by clinicians. *Annals of Internal Medicine*, 90(6):957, 1979.

[6] S. Plous. *The psychology of judgment and decision making.* Mcgraw-Hill Book Company, 1993.

[7] J. Baker, J. Cote, and B. Abernethy. Sport-specific practice and the development of expert decision-making in team ball sports. *Journal of applied sport psychology*, 15(1):12–25, 2003.

[8] D.M. Kreps. Game theory and economic modelling. *OUP Catalogue*, 2011.

[9] M.J. Osborne and A. Rubinstein. *A course in game theory.* MIT press, 1994.

[10] Victor L. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, 1994.

[11] Christos M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[12] Kirk L. Kroeker. A new benchmark for artificial intelligence. *Commun. ACM*, 54(8):13–15, August 2011.

[13] S. Gelly and D. Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[14] M. Enzenberger et al. Evaluation in go by a neural network using soft segmentation. In *10th advances in computer games conference*, pages 97–108, 2003.

[15] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[16] C.S. Lee, M.H. Wang, G. Chaslot, J.B. Hoock, A. Rimmel, O. Teytaud, S.R. Tsai, S.C. Hsu, and T.P. Hong. The computational intelligence of mogo revealed in taiwan's computer go tournaments. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(1):73–89, 2009.

[17] P.C.H. Albers and H. de Vries. Elo-rating as a tool in the sequential estimation of dominance strengths. *Animal Behaviour*, pages 489–495, 2001.

[18] Victor Allis. A knowledge-based approach to connect-four. the game is solved: White wins. In *Master's thesis, Vrije Universiteit*, page 4, 1988.

[19] A.F.C. Arts. Competitive play in stratego. Master's thesis, Department of Knowledge Engineering, Maastricht University, 2010.

[20] John Tromp and Gunnar Farnebäck. Combinatorics of go. In H.Jaap Herik, Paolo Ciancarini, and H.H.L.M.(Jeroen) Donkers, editors, *Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 84–99. Springer Berlin Heidelberg, 2007.

[21] F. C. Schadd. Monte-carlo search techniques in the modern board game thurn and taxis. Master's thesis, Department of Computer Science, Maastricht University, 2009.

[22] Shi-Jim Yen, Jr-Chang Chen, Tai-Ning Yang, and Shun chin Hsu. Computer chinese chess. *ICGA Journal*, 27(1):3–18, 2004.

[23] P. P. L. M. Hensgens. A knowledge-based approach of the game of amazons. Master's thesis, Department of Computer Science, Maastricht University, 2001.

[24] H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Jack van Rijswijck. Games solved: Now and in the future. *Artif. Intell.*, 134(1-2):277–311, 2002.

[25] Pascal Chorus. Implementing a computer player for abalone using alpha-beta and monte-carlo search. Master's thesis, Department of Knowledge Engineering, Maastricht University, 2009.

[26] Claude E. Shannon. Xxii. programming a computer for playing chess. *Philosophical Magazine Series 7*, 41(314):256–275, 1950.

[27] M.H.M. Winands. *Informed Search in Complex Games*. SIKS dissertation series. Universitaire Pers Maastricht, 2004.

[28] Maarten P. D. Schadd, Mark H. M. Winands, Jos W. H. M. Uiterwijk, H. Jaap van Den Herik, and Maurice H. J. Bergsma. Best play in fanorona leads to draw. *New Mathematics and Natural Computation (NMNC)*, 04(03):369–387, 2008.

[29] richardk. The ai in the octagon theory, July 2012. url: http://aigamedev.com/open/interview/the-octagon-theory/ (last visited on 2013-05-16).

[30] M.H. Kalos and P.A. Whitlock. *Monte carlo methods*. Wiley-VCH, 2009.

[31] P.A. Nogueira, R. Rodrigues, and E. Oliveira. Guided emotional state regulation: Understanding and shaping players' affective experiences in digital games. In *Proceedings of the 12th International Conference on Entertainment Computing (ICEC)*, 2013 (To appear).

[32] B. Abramson. Expected-outcome: a general model of static evaluation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 12(2):182 –193, feb 1990.

[33] Brian Sheppard. World-championship-caliber scrabble. *Artif. Intell.*, 134(1-2):241–275, January 2002.

[34] Matthew L. Ginsberg. Gib: Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research*, 14:303–358, 2001.

[35] Cameron Browne. The Dangers of Random Playouts. *ICGA Journal*, 34(1):25–26, March 2011.

[36] Ingo Althöfer. On board-filling games with random-turn order and monte carlo perfectness. In H.Jaap Herik and Aske Plaat, editors, *Advances in Computer Games*, volume 7168 of *Lecture Notes in Computer Science*, pages 258–269. Springer Berlin Heidelberg, 2012.

[37] Tristan Cazenave and Bernard Helmstetter. Combining tactical search and monte-carlo in the game of go. In *IN: CIG'05*, pages 171–175, 2005.

[38] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In Christian Darken and Michael Mateas, editors, *AIIDE*. The AAAI Press, 2008.

[39] L. Rejeb, Z. Guessoum, and R. M'Hallah. The exploration-exploitation dilemma for adaptive agents. In *Proceedings of the 5th European Workshop on Adaptive Agents and Multi-Agent Systems (AAMAS05)*. Citeseer, 2005.

[40] R.S. Sutton and A.G. Barto. *Introduction to reinforcement learning*. MIT Press, 1998.

[41] B.E. Childs, J.H. Brodeur, and L. Kocsis. Transpositions and move groups in monte carlo tree search. In *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, pages 389 –395, dec. 2008.

[42] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, ICML '07, pages 273–280, New York, NY, USA, 2007. ACM.

[43] T. Kozelek. Methods of mcts and the game arimaa. *Master's thesis, Charles University in Prague*, 2009.

[44] ThomasR. Lincke. Strategies for the automatic construction of opening books. In Tony Marsland and Ian Frank, editors, *Computers and Games*, volume 2063 of *Lecture Notes in Computer Science*, pages 74–86. Springer Berlin Heidelberg, 2001.

[45] Pierre Audouard, Guillaume Chaslot, Jean-Baptiste Hoock, Julien Perez, Arpad Rimmel, and Olivier Teytaud. Grid coevolution for adaptive simulations: Application to the building of opening books in the game of go. In *Proceedings of the EvoWorkshops 2009 on Applications of Evolutionary Computing: EvoCOMNET, EvoENVIRONMENT, EvoFIN, EvoGAMES, EvoHOT, EvoIASP, EvoINTERACTION, EvoMUSART, EvoNUM, EvoSTOC, EvoTRANSLOG*, EvoWorkshops '09, pages 323–332, Berlin, Heidelberg, 2009. Springer-Verlag.

[46] Julien Kloetzer. Monte-carlo opening books for amazons. In *Proceedings of the 7th international conference on Computers and games*, CG'10, pages 124–135, Berlin, Heidelberg, 2011. Springer-Verlag.

[47] S. Gelly and Y. Wang. Exploration exploitation in go: UCT for Monte-Carlo go. *Twentieth Annual Conference on Neural Information Processing Systems (NIPS 2006)*, 2006.

[48] F. Teytaud and O. Teytaud. On the huge benefit of decisive moves in monte-carlo tree search algorithms. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 359 –364, aug. 2010.

[49] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[50] Guillaume M. J-B. Chaslot, Mark H. M. Winands, H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4:343–357, 2008.

[51] P. Drake and S. Uurtamo. Move ordering vs heavy playouts: Where should heuristics be applied in monte carlo go. In *Proceedings of the 3rd North American Game-On Conference*. Citeseer, 2007.

[52] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Research Report RR-6062, INRIA, 2006.

[53] R. Coulom. Computing elo ratings of move patterns in the game of go. In *Computer games workshop*, 2007.

[54] D. Silver and G. Tesauro. Monte-carlo simulation balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 945–952. ACM, 2009.

[55] Arpad Rimmel and Fabien Teytaud. Multiple overlapping tiles for contextual monte carlo tree search. In *Proceedings of the 2010 international conference on Applications of Evolutionary Computation - Volume Part I*, EvoApplicatons'10, pages 201–210, Berlin, Heidelberg, 2010. Springer-Verlag.

[56] M. Genesereth, N. Love, and B. Pell. General game playing: Overview of the aaai competition. *AI magazine*, 26(2):62, 2005.

[57] Y. Björnsson and H. Finnsson. Cadiaplayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15, 2009.

[58] H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In *The Twenty-Third AAAI Conference on Artificial Intelligence*, pages 259–264, 2008.

[59] H. Finnsson and Y. Björnsson. Learning simulation control in general game-playing agents. In *Proc. 24th AAAI Conf. Artif. Intell., Atlanta, Georgia*, pages 954–959, 2010.

[60] F. Xie and Z. Liu. Backpropagation modification in monte-carlo game tree search. In *Intelligent Information Technology Application, 2009. IITA 2009. Third International Symposium on*, volume 2, pages 125–128. IEEE, 2009.

[61] D. Tom and M. Müller. A study of uct and its enhancements in an artificial game. *Advances in Computer Games*, pages 55–64, 2010.

[62] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. *Machine Learning: ECML 2006*, pages 282–293, 2006.

[63] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th international conference on Computers and games*, CG'06, pages 72–83, Berlin, Heidelberg, 2007. Springer-Verlag.

[64] P.H. Rabinowitz. *Minimax methods in critical point theory with applications to differential equations*, volume 65. Amer Mathematical Society, 1986.

[65] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1976.

[66] B. Bouzy. Move-pruning techniques for monte-carlo go. *Advances in Computer Games*, pages 104–119, 2006.

[67] J. Huang, Z. Liu, L. Benjie, and X. Feng. Pruning in uct algorithm. In *Technologies and Applications of Artificial Intelligence (TAAI), 2010 International Conference on*, pages 177–181. IEEE, 2010.

[68] S. He, Y. Wang, F. Xie, J. Meng, H. Chen, S. Luo, Z. Liu, and Q. Zhu. Game player strategy pattern recognition and how uct algorithms apply pre-knowledge of player's strategy to improve opponent ai. In *Computational Intelligence for Modelling Control & Automation, 2008 International Conference on*, pages 1177–1181. IEEE, 2008.

[69] L. Kocsis, C. Szepesvári, and J. Willemson. Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1, 2006.

[70] A. Mahajan and D. Teneketzis. Multi-armed bandit problems. *Foundations and Applications of Sensor Management*, pages 121–151, 2008.

[71] D.P. Helmbold and A. Parker-Wood. All-moves-as-first heuristics in monte-carlo go. In *Proceedings of the 2009 International Conference on Artificial Intelligence, ICAI*, pages 605–610, 2009.

[72] A. Rimmel, F. Teytaud, and O. Teytaud. Biasing monte-carlo simulations through rave values. *Computers and Games*, pages 59–68, 2011.

[73] B. Bouzy, M. Métivier, and D. Pellier. Mcts experiments on the voronoi game. *Advances in Computer Games*, pages 96–107, 2012.

[74] R. Lorentz. Improving monte–carlo tree search in havannah. *Computers and Games*, pages 105–115, 2011.

[75] D. Tom. *Investigating UCT and RAVE: Steps towards a more robust method.* PhD thesis, University of Alberta, 2010.

[76] M. Müller. Computer go. *Artificial Intelligence*, 134(1):145–179, 2002.

[77] J. Nijssen and M. Winands. Enhancements for multi-player monte-carlo tree search. *Computers and Games*, pages 238–249, 2011.

[78] N.G.P. Den Teuling. Monte-carlo tree search for the simultaneous move game tron. *Univ. Maastricht, Netherlands, Tech. Rep*, 2011.

[79] Pierre Perick, David L St-Pierre, Francis Maes, and Damien Ernst. Comparison of different selection strategies in monte-carlo tree search for the game of tron. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 242–249. IEEE, 2012.

[80] M. Enzenberger, M. Muller, B. Arneson, and R. Segal. Fuego—an open-source framework for board games and go engine based on monte carlo tree search. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):259–270, 2010.

[81] R. Coulom. Monte-carlo tree search in crazy stone. In *Proceedings of Game Programming Workshop*, volume 2007, 2007.

[82] J. Rubin and I. Watson. Computer poker: A review. *Artificial Intelligence*, 175(5):958–987, 2011.

[83] G. Van den Broeck, K. Driessens, and J. Ramon. Monte-carlo tree search in poker using expected reward distributions. *Advances in Machine Learning*, pages 367–381, 2009.

[84] M. Ponsen, G. Gerritsen, and G. Chaslot. Integrating opponent models with monte-carlo tree search in poker. In *Proc. Conf. Assoc. Adv. Artif. Intell.: Inter. Decis. Theory Game Theory Workshop, Atlanta, Georgia*, pages 37–42, 2010.

[85] D. Silver and J. Veness. Monte-carlo planning in large pomdps. *Advances in Neural Information Processing Systems (NIPS)*, 46, 2010.

[86] R. Gaudel, M. Sebag, et al. Feature selection as a one-player game. In *International Conference on Machine Learning*, pages 359–366, 2010.

[87] K. Kira and L.A. Rendell. The feature selection problem: Traditional methods and a new algorithm. In *Proceedings of the National Conference on Artificial Intelligence*, pages 129–129. John Wiley & Sons Ltd, 1992.

[88] T.J. Walsh, S. Goschin, and M.L. Littman. Integrating sample-based planning and model-based reinforcement learning. In *Proceedings of AAAI*, 2010.

[89] H. Nakhost and M. Müller. Monte-carlo exploration for deterministic planning. In *IJCAI*, pages 1766–1771, 2009.

[90] S. Pinson, F. Balbo, and T. Cazenave. Monte-carlo bus regulation. In *IEEE Conf. on Intelligent Transportation Systems, ITSC'09*, 2009.

[91] A. Wren and J.M. Rousseau. Bus driver scheduling-an overview. *Computer-aided transit scheduling*, pages 173–187, 1995.

[92] G. Chaslot, S. De Jong, J.T. Saito, and J. Uiterwijk. Monte-carlo tree search in production management problems. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, pages 91–98, 2006.

[93] A. Rimmel, F. Teytaud, and T. Cazenave. Optimization of the nested monte-carlo algorithm on the traveling salesman problem with time windows. *Applications of Evolutionary Computation*, pages 501–510, 2011.

[94] E.L. Lawler, J.K. Lenstra, A.H.G.R. Kan, and D.B. Shmoys. *The traveling salesman problem: a guided tour of combinatorial optimization*, volume 3. Wiley New York, 1985.

[95] A. Sabharwal, H. Samulowitz, and C. Reddy. Guiding combinatorial optimization with uct. *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*, pages 356–361, 2012.

[96] G. Nemhauser and D. Bienstock. *Integer Programming and Combinatorial Optimization: 10th International IPCO Conference, New York, NY, USA, June 7-11, 2004, Proceedings*, volume 10. Springer, 2004.

[97] H. J. van den Herik, H. H. L. M. Donkers, and P. H. M. Spronck. Opponent Modelling and Commercial Games. In G. Kendall and S. Lucas, editors, *Proceedings of IEEE 2005 Symposium on Computational Intelligence and Games CIG'05*, pages 15–25, 2005.

[98] M. Campbell, A.J. Hoane, and F. Hsu. Deep blue. *Artificial intelligence*, 134(1):57–83, 2002.

[99] D. Carmel and S. Markovitch. Learning models of intelligent agents. In *Proceedings of the National Conference on Artificial Intelligence*, pages 62–67, 1996.

[100] D. Carmel, S. Markovitch, et al. *Learning models of opponent's strategy in game playing*. Technion-Israel Institute of Technology, Center for Intelligent Systems, 1993.

[101] M. Pazzani and D. Billsus. Learning and revising user profiles: The identification of interesting web sites. *Machine learning*, 27(3):313–331, 1997.

[102] D. Sklansky. *The theory of poker*. Two Plus Two Pub, 1999.

[103] M.F. Goodchild. Data models and data quality: problems and prospects. *Visualization in geographical information systems*, pages 141–149, 1993.

[104] J.R. Anderson. *Machine learning: An artificial intelligence approach*, volume 2. Morgan Kaufmann, 1986.

[105] Luís Filipe Teófilo, Nuno Passos, Luís Paulo Reis, and Henrique Lopes Cardoso. Adapting strategies to opponent models in incomplete information games: a reinforcement learning approach for poker. In *Autonomous and Intelligent Systems*, pages 220–227. Springer, 2012.

[106] BD Davidson and H. Hirsh. Predicting sequences of user actions. predicting the future: Ai approaches to time-series problems. Technical report, Technical Report WS-98-07, 1998.

[107] John A Hartigan. *Clustering algorithms*. John Wiley & Sons, Inc., 1975.

[108] P. Domingos and M. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine learning*, 29(2):103–130, 1997.

[109] RJS Baker and PL Cowling. Bayesian opponent modeling in a simple poker environment. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 125–131. IEEE, 2007.

[110] F. Southey, M.P. Bowling, B. Larson, C. Piccione, N. Burch, D. Billings, and C. Rayner. Bayes' bluff: Opponent modelling in poker. *arXiv preprint arXiv:1207.1411*, 2012.

[111] J.M. Vidal and E.H. Durfee. Agents learning about agents: A framework and analysis. In *Working Notes of the AAAI-97 workshop on Multiagent Learning*, volume 7176, 1997.

[112] E. Oliveira, J.M. Fonseca, and N.R. Jennings. Learning to be competitive in the market. In *Proceedings of the AAAI Workshop on Negotiation: Settling Conflicts and Identifying Opportunities*, pages 30–37, 1999.

[113] Luís Filipe Teófilo and Luís Paulo Reis. Identifying player's strategies in no limit texas hold'em poker through the analysis of individual moves. *arXiv preprint arXiv:1301.5943*, 2013.

[114] Luís Filipe Teófilo, Rosaldo Rossetti, Luís Paulo Reis, Henrique Lopes Cardoso, and Pedro Alves Nogueira. Simulation and performance assessment of poker agents. In *Multi-Agent-Based Simulation XIII*, pages 69–84. Springer, 2013.

[115] D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of poker. *Artificial Intelligence*, 134(1):201–240, 2002.

[116] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA, 1995.

[117] A. Del Giudice, P. Gmytrasiewicz, and J. Bryan. Towards strategic kriegspiel play with opponent modeling. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 1265–1266. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[118] M. Richards and E. Amir. Opponent modeling in scrabble. In *Proceedings of the 20th international joint conference on Artifical intelligence*, pages 1482–1487. Morgan Kaufmann Publishers Inc., 2007.

[119] F. Schadd, S. Bakkes, and P. Spronck. Opponent modeling in real-time strategy games. In *8th International Conference on Intelligent Games and Simulation (GAME-ON 2007)*, pages 61–68, 2007.

[120] D. Carmel and S. Markovitch. Opponent modeling in multi-agent systems. *Adaption and Learning in Multi-Agent Systems*, pages 40–52, 1996.

[121] Charles M Grinstead and J Laurie Snell. *Introduction to probability*. American Mathematical Soc., 1998.

[122] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International joint Conference on artificial intelligence*, volume 14, pages 1137–1145. Lawrence Erlbaum Associates Ltd, 1995.