

TECHNICAL UNIVERSITY OF CRETE, GREECE
SCHOOL OF ELECTRONIC AND COMPUTER ENGINEERING

Algorithm Modeling for Hardware Implementation of a Blokus Duo Player



Sofia Maria Nikolakaki

Thesis Committee

Professor Apostolos Dollas (ECE)

Professor Minos Garofalakis (ECE)

Associate Professor Ioannis Papaefstathiou (ECE)

Chania, February 2014

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Μοντελοποίηση Αλγορίθμων για
Υλοποίηση σε Υλικό για το Παιχνίδι
Blokus Duo



Σοφία Μαρία Νικολακάκη

Εξεταστική Επιτροπή

Καθ. Απόστολος Δόλλας (ΗΜΜΥ)

Καθ. Μίνως Γαροφαλάκης (ΗΜΜΥ)

Αναπλ. Καθ. Ιωάννης Παπευσταθίου (ΗΜΜΥ)

Χανιά, Φεβρουάριος 2014

Abstract

Artificial Intelligence has a profound impact on a wide range of scientific fields and has been well applied especially in game playing. Until recently, the dominance of the Minimax algorithm in two player zero-sum games was indisputable, since it guaranteed an optimal solution as long as the search reached a certain tree depth. However, it required configuration of a good heuristic function and sometimes a prohibitive amount of execution time. This led to the design of a new and quite recent algorithm, the Monte Carlo Tree Search (MCTS) algorithm. MCTS appeared to be promising from the very beginning since it performed better than state-of-the-art algorithms in the currently most challenging two player zero-sum game, Go. Over the last seven years, many have tried to comprehend and evaluate the MCTS algorithm performance by applying it on different games and by gathering experimental results. Others, have used a variety of heuristics and techniques to improve the algorithm's efficiency, thus creating different MCTS variations. Among these, the most popular one is the Upper Confidence Bound for Trees algorithm (UCT).

The motivation behind the present work was initiated by the potential for comparing the performance of MCTS with other algorithms, modeling it for hardware implementation and improving its efficiency in terms of winning percentage. Therefore, we applied the algorithm on the game of Blokus Duo, a relatively new game, open for research since it meets the requirements of MCTS and appears to be demanding. More specifically, we present four competitive Blokus Duo players and show that the ones, based on Monte Carlo simulations outperform the Minimax-based one. To the best of our knowledge, this is the first work that compares for the same game, software-based Minimax, Monte Carlo and MCTS players. For each of these players we suggest opportunities for hardware implementation and discuss potential bottlenecks. Furthermore, we apply certain heuristics on our MCTS-based player to understand how they affect the efficiency of the algorithm specifically for the game of Blokus Duo.

Περίληψη

Ο τομέας της Τεχνητής Νοημοσύνης έχει προκαλέσει αισθητό αντίκτυπο σε ένα ευρύ φάσμα επιστημονικών πεδίων και εφαρμόζεται ιδιαίτερα σε παιχνίδια. Μέχρι πρόσφατα, η κυριαρχία του αλγορίθμου Minimax ήταν αδιαμφισβήτη σε zero-sum παιχνίδια με δύο παίκτες, καθώς εγγυόταν βέλτιστη λύση αρκεί να έφτανε η αναζήτηση σε συγκεκριμένο βάθος του δέντρου. Ωστόσο, προϋπόθετε την ύπαρξη μίας καλής ευριστικής συνάρτησης και ορισμένες φορές απαγορευτικό χρόνο εκτέλεσης. Αυτό οδήγησε στο σχεδιασμό ενός καινούριου και αρκετά πρόσφατου αλγορίθμου, του Monte Carlo Tree Search (MCTS) αλγορίθμου. Ο MCTS φάνηκε από την αρχή πολλά υποσχόμενος καθώς είχε καλύτερη απόδοση από state-of-the-art αλγορίθμους στο τρέχον πιο δύσκολο zero-sum παιχνίδι δύο παιχτών, το Go. Τα τελευταία επτά χρόνια πολλοί προσπάθησαν να κατανοήσουν και να αξιολογήσουν την απόδοση του MCTS, με την εφαρμογή του σε διαφορετικά παιχνίδια και την συλλογή πειραματικών αποτελεσμάτων. Άλλοι, έχουν χρησιμοποιήσει διαφορετικές ευριστικές μεθόδους και τεχνικές, για να βελτιώσουν την επίδοση του αλγορίθμου, δημιουργώντας έτσι και νέες παραλλαγές του. Ανάμεσα σε αυτές, η πιο γνωστή είναι ο αλγόριθμος Upper Confidence Bound for Trees (UCT).

Το κίνητρο για αυτή τη δουλειά προκλήθηκε από τη δυνατότητα σύγκρισης της επίδοσης του MCTS με άλλους αλγορίθμους, μοντελοποίησης του για υλοποίηση σε υλικό και βελτίωσης του ποσοστού νίκης του. Γι' αυτό το λόγο, εφαρμόσαμε τον αλγόριθμο στο παιχνίδι Blokus Duo, ένα σχετικά καινούριο παιχνίδι και ανοιχτό για έρευνα, καθώς ανταποκρίνεται στις απαιτήσεις του MCTS και φαίνεται να είναι αρκετά δύσκολο. Ειδικότερα, παρουσιάζουμε τέσσερις ανταγωνιστικούς Blokus Duo παίκτες και δείχνουμε ότι εκείνοι που βασίζονται σε Monte Carlo προσομοιώσεις αποδίδουν καλύτερα σε σχέση με εκείνον που βασίζεται στον Minimax αλγόριθμο. Σύμφωνα με τα όσα γνωρίζουμε, η συγκεκριμένη δουλειά είναι η πρώτη που συγκρίνει για το ίδιο παιχνίδι παίκτες υλοποιημένους σε λογισμικό, που βασίζονται στους αλγορίθμους Minimax, Monte Carlo και MCTS. Για κάθε έναν από αυτούς τους παίκτες υποδεικνύουμε δυνατότητες για την υλοποίησή τους σε υλικό και αναφέρουμε πιθανά bottlenecks. Επιπλέον, εφαρμόζουμε συγκεκριμένες ευριστικές μεθόδους στον παίκτη που βασίζεται στον MCTS αλγόριθμο, ώστε να καταλάβουμε πώς επηρεάζουν την απόδοση του αλγορίθμου, ειδικά για το παιχνίδι του Blokus Duo.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor, Prof. Apostolos Dollas, for his trust, enthusiasm, continuous guidance from the very beginning and for our fruitful discussions concerning my future steps. I would also like to thank Prof. Minos Garofalakis and Prof. Ioannis Papaefstathiou for accepting to be in my committee.

I am also deeply grateful to Prof. Stavros Christodoulakis for inspiring me throughout the first years of my studies and for providing me the right incentives to excel.

Furthermore, I would like to thank Pavlos Malakonakis for his assistance whenever needed and for our cooperation, as well as Nikolaos Kofinas and Ioakeim Perros for not only being good friends to me, but for also offering me fine suggestions regarding my thesis.

My parents, Ioannis and Souzana, as well as my beloved younger brother, Emmanouil for always being my role models in matters of morals and principles, for loving me and for inspiring me to follow their steps.

I am thankful to my dear Ioannis Demertzis for encouraging and supporting me unconditionally, in the last few years.

Last but definitely not least, my two best friends Theoni Magounaki and Dimitra Paliatsa for believing in me, for being beside me no matter what and for crafting with me memories that I will always cherish. I would also like to thank all my friends for all the moments, surprises and experiences that we shared together.

Contents

1	Introduction	1
1.1	Thesis Contribution	2
1.2	Thesis Outline	3
2	Background	5
2.1	The Game of Blokus Duo	5
2.1.1	History	5
2.1.2	Rules	6
2.1.3	Strategy Tips	7
2.1.4	Existing Programs	11
2.2	Decision theory	11
2.2.1	Markov decision processes	12
2.2.2	Applications	14
2.3	Game theory	15
2.3.1	Games	15
2.3.2	Game tree	16
2.3.3	Combinatorial games	16
2.4	Minimax with alpha-beta pruning	17
2.4.1	Minimax with alpha-beta pruning Algorithm	17
2.4.2	Complexity	18
2.5	Monte Carlo Methods	19
2.5.1	Monte Carlo simulations	20
2.5.2	Uniform sampling in Monte Carlo	21
2.6	Bandit-Based Methods	21
2.6.1	Multi-armed bandit problems	21

CONTENTS

2.6.2	Regret	22
2.6.3	UCB1	23
2.7	Monte Carlo Tree Search	24
2.7.1	MCTS Development	24
2.7.2	MCTS Characteristics	25
2.7.2.1	Benefits	25
2.7.2.2	Drawbacks	26
2.7.3	MCTS Algorithm	26
2.7.3.1	Selection	28
2.7.3.2	Expansion	28
2.7.3.3	Simulation	29
2.7.3.4	Backpropagation	29
2.7.3.5	Final move selection	29
2.7.4	Upper Confidence Bounds for Trees - UCT	30
2.7.5	MCTS Enhancements	31
2.7.5.1	Selection phase	31
2.7.5.2	Simulation phase	33
3	Related Work	35
3.1	Blokus Duo	35
3.1.1	Blokus Duo MCTS approach	35
3.1.2	Blokus Duo Minimax based agents	36
3.2	Open source Go MCTS implementations	39
3.2.1	Fuego 1.1 Version	39
3.2.2	Pachi 10.0 Version	40
4	Implementation	43
4.1	Blokus Duo components	43
4.1.1	Tiles	44
4.1.2	Game Board	44
4.1.3	Software Implementation	44
4.1.4	Hardware Implementation	45
4.2	Minimax with alpha-beta pruning player	45
4.2.1	Software Implementation	45
4.2.1.1	Minimax Structures	46

4.2.1.2	Minimax Algorithm	46
4.2.2	Modeling for Hardware Implementation	49
4.2.2.1	Code Profiling	49
4.2.2.2	Memory Requirements	50
4.2.2.3	Potential Parallelism	50
4.2.2.4	Potential Bottlenecks	51
4.2.3	Hardware Implementation	51
4.3	MCTS player	53
4.3.1	Software Implementation	53
4.3.1.1	MCTS structures	53
4.3.1.2	MCTS Algorithm	55
4.3.2	Modeling for Hardware Implementation	58
4.3.2.1	Code Profiling	58
4.3.2.2	Memory Requirements	58
4.3.2.3	Potential Parallelism	59
4.3.2.4	Potential Bottlenecks	60
4.4	Monte Carlo player	61
4.4.1	Software Implementation	61
4.4.1.1	Monte Carlo structures	61
4.4.1.2	Monte Carlo Algorithm	62
4.4.2	Modeling for Hardware Implementation	63
5	Optimizations on MCTS	65
5.1	Selection Policies	65
5.1.1	UCB1 Adjustment	65
5.1.2	UCB1-Tuned	66
5.2	Selection Phase	67
5.2.1	First Play Urgency (FPU)	67
5.2.2	Progressive Bias	69
5.3	Simulation Phase	70
5.3.1	Evaluation Function	72
5.3.2	Score Bonus	73
5.3.3	Best Combination of Optimizations	73

CONTENTS

6	Comparison of players	77
6.1	Comparison of UCT with Monte Carlo	77
6.2	Comparison of UCT with Minimax	79
6.3	Comparison of Monte Carlo with Minimax	80
6.4	Comparison of Enhanced MCTS with UCT	82
6.5	Comparison of Enhanced MCTS with Monte Carlo	83
6.6	Comparison of Enhanced MCTS with Minimax	85
7	Conclusion and Future Work	87
7.1	Conclusion	87
7.2	Future Work	88
	References	91

List of Figures

2.1	Tiles of the Blokus Duo game. Each tile comprises from one to five units.	8
2.2	Board of the Blokus Duo game. Square 1 indicates the starting point of player 1 and square 2 indicates the starting point of player 2.	9
2.3	The purple tiles are connected with a corner-to-corner contact.	10
2.4	The left move is allowed, since same colored tiles are connected with a corner-to-corner contact but do not have any edge-to-edge contact. The right move is prohibited, since the purple tiles have an edge-to-edge contact.	11
2.5	An example snapshot of the game.	12
2.6	An example of a completed game. The orange player placed all of his tiles, hence he gets a bonus of 15 points. The purple player did not place 2 three-unit tiles and a four-unit tile. Therefore, his score is $2*(-3)+(-4) = -10$. The orange player has the highest score and is the winner of this game. .	13
2.7	Important areas on the Blokus Duo gameboard	14
2.8	Four phases of the MCTS algorithm.	30
4.1	Minimax core module.	52
4.2	Minimax controller's FSM	53
5.1	Winning percentage for different Cp values	66
5.2	Comparison between UCB1 and UCB1Tuned policies. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.	67
5.3	Comparison between Heuristic, 10000 and 0.1 FPU values. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.	69

LIST OF FIGURES

- 5.4 Comparison between the UCT algorithm enhanced with the Progressive Bias technique that uses the evaluation function and the normal UCT algorithm. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player. 71
- 5.5 Comparison between the UCT algorithm enhanced with the Progressive Bias technique that counts the units of the tile and the normal UCT algorithm. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player. . . . 71
- 5.6 Comparison between the UCT algorithm enhanced with the Evaluation Function technique that uses the a heuristic function and the normal UCT algorithm. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player. 72
- 5.7 Comparison between the UCT algorithm enhanced with the Evaluation Function technique that counts the units of the tile and the normal UCT algorithm. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player. 73
- 5.8 Comparison between the UCT algorithm enhanced with the Score Bonus technique that uses the first family of scores and the normal UCT algorithm. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player. . . . 74
- 5.9 Comparison between the UCT algorithm enhanced with the Score Bonus technique that uses the second family of scores and the normal UCT algorithm. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player. . . . 74
- 6.1 Comparison between Monte Carlo Tree Search and Monte Carlo players when UCT plays first. Blue indicates UCT and gray indicates MC. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player. 78
- 6.2 Comparison between Monte Carlo Tree Search and Monte Carlo players when MC plays first. Blue indicates UCT and gray indicates MC. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player. 79

LIST OF FIGURES

6.3	Comparison between Monte Carlo Tree Search and Minimax players when UCT plays first. Blue indicates UCT and gray indicates Minimax. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.	81
6.4	Comparison between Monte Carlo Tree Search and Minimax players when Minimax plays first. Blue indicates UCT and gray indicates Minimax. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.	81
6.5	Comparison between Monte Carlo and Minimax players when MC plays first. Blue indicates Monte Carlo and gray indicates Minimax. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.	82
6.6	Comparison between Monte Carlo and Minimax players when Minimax plays first. Blue indicates Monte Carlo and gray indicates Minimax. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.	83
6.7	Comparison between Enhanced MCTS and UCT players when Enhanced MCTS plays first. Blue indicates Enhanced MCTS and gray indicates UCT. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.	84
6.8	Comparison between Enhanced MCTS and UCT players when UCT plays first. Blue indicates Enhanced MCTS and gray indicates UCT. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.	84
6.9	Comparison between Enhanced MCTS and Monte Carlo players when Enhanced MCTS plays first. Blue indicates Enhanced MCTS and gray indicates Monte Carlo. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.	86
6.10	Comparison between Enhanced MCTS and Monte Carlo players when Monte Carlo plays first. Blue indicates Enhanced MCTS and gray indicates Monte Carlo. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.	86

LIST OF FIGURES

Chapter 1

Introduction

Artificial Intelligence (AI) is a term frequently encountered in every day life, thus proving its significant applicability on a wide range of fields. Among these fields, some of the more popular ones include playing games competitively and decision theory problems. Until recently in order to find a solution to these types of problems, the primarily suggested techniques were knowledge-based approaches and the Minimax algorithm with a-b pruning. However, both solutions perform poorly when decision making problems have either one or more of the following properties: a high branching factor, a deep tree or a moderate heuristic function to evaluate non-terminal nodes. An example of such a problem that comprises these properties is the game of Go, where the game has approximately 200 moves, there are about 250 legal moves per round and there is no knowledge of a reliable heuristic function for non-terminal states. It is a game that attracts the interest of many computer scientists since humans are much better players than the best known Go programs. This led to the development of the MCTS algorithm that seems to be more appropriate for such challenges. MCTS is an anytime and aheuristic algorithm that works based on statistics and usually the longer the execution time lasts, the better the performance. There are cases reported, such as in the game of Go, where MCTS based agents are the strongest computer players. Hence, it is evident that MCTS is promising and capable of succeeding on daunting problems where other techniques fail. Its efficiency can be attributed to the fact that it does not evaluate intermediate game positions, as Minimax does, but conducts a broader and more robust search. The most prominent algorithm in the MCTS family appears to be the Upper Confidence Bounds

1. INTRODUCTION

for Trees algorithm (UCT), due to the fact that it is simple to implement and guarantees to be within a constant factor of the best possible bound on the growth of regret. In general, MCTS approaches are applied on combinatorial games, single player games, non-deterministic games as well as non-game applications, such as combinatorial optimization and scheduling problems. In order to be able to study and evaluate the MCTS algorithm we decided to apply it on a game that is characterized by properties appropriate for the particular algorithm. Therefore we concluded on the game of Blokus Duo, a perfect information two player zero-sum game with a finite number of moves and no chance elements a.k.a a combinatorial game. The specific game drew our interest because it was the challenge of the [ICFPT 2013 design conference](#), it is relatively new with few published heuristic functions but, most importantly, the first levels of the tree created have an exponentially increasing number of children nodes and the number of candidates is sometimes over 1000, properties that indicate its demanding nature. Also it appears to be quite similar to Go since each player focuses on gaining the largest owned area in order to win a game, but has a much smaller game size of maximum 42 moves that renders it appropriate for the MCTS method. Due to the fact that MCTS was developed recently, scientists are still trying to comprehend basic factors of the algorithm and how they affect its overall performance. Therefore MCTS is applied on a variety of problems and in each one modifications are made in order to understand the impact of different parameters.

1.1 Thesis Contribution

The [ICFPT 2013 conference](#) proposed the game of Blokus Duo as the challenge for its design competition, thus showing that it is at least an intriguing game. While preparing for this competition we studied algorithms that could create an efficient Blokus Duo player and concluded that Minimax, Monte Carlo and MCTS seemed appropriate ones. However, due to the competition's time and memory restrictions we decided to implement a basic Minimax hardware-based Blokus Duo player and did not get involved with the other two algorithms. Within this work, we decided to implement in software all three algorithms and study their performance for the specific game, as well as compare their efficiency in terms of time and memory resources. To the best of our knowledge no previous work has conducted such a comparison, given that the MCTS algorithm was

published quite recently. Furthermore, we introduce an enhanced Blokus Duo player based on the Upper Confidence Bounds for Trees algorithm (UCT) of the Monte Carlo Tree Search family. Each step of the development process, as well as the incentive behind of every modification is described in section 5. According to the Monte Carlo Tree Search Methods 2012 Survey [1], research directions incline to apply different techniques in conjunction with the general-purpose UCT algorithm to improve its performance. The discrete stages of the UCT algorithm allow it to be hybridised with a wide range of other approaches, especially heuristic evaluations of intermediate states and knowledge based approaches. We applied suggested techniques to improve our player and study their impact on the execution time and winning percentage factors. Moreover, for each of the Blokus Duo players we provide opportunities, suggestions and expected bottlenecks for hardware implementations. These are primarily defined by the nature of the algorithm, such as whether it is recursive or not, memory requirements etc. They also depend on the elements of the game, such as the rules, the components and others.

1.2 Thesis Outline

Chapter 2 introduces us to the Monte Carlo Tree Search approaches by presenting thoroughly the transition from the Monte Carlo simulations to the UCT algorithm. Furthermore, the rules of the Blokus Duo game are described in this section, as well as critical strategies followed by professional players. These strategies provided us with useful information, necessary for the heuristics that would be implemented. We also present a brief view of the scientific field we are studying emphasizing on techniques that appeared to be useful during the development of our enhanced agent. Chapter 3 sums up significant attempts made in the field to improve the basic UCT algorithm for various games, as reported in the literature. This overview indicates techniques proven to be successful and highlights approaches that have not been yet taken into account. We also refer to previous works concerning the Blokus Duo game, some of which were recently published in the ICFPT 2013 conference. In Chapter 4 we present how the Blokus Duo players were implemented and mention all data structures that were used, as well as the execution flow of the algorithm. We also analyze hardware opportunities and bottlenecks that these algorithms offer. In Chapter 5 we describe all heuristics that were applied to the general-purpose UCT algorithm to improve its performance and discuss whether

1. INTRODUCTION

they were eventually effective. In Chapter 6 we conduct a comparison between all four implemented Blokus Duo players and comment on the results. Finally, in Chapter 7 a conclusion about the presented work is provided, followed by future work directions that are worth considering.

Chapter 2

Background

2.1 The Game of Blokus Duo

Blokus Duo (a.k.a Travel Blokus) is an expansion of the game of Blokus. It is a zero-sum, perfect information, 2-player game. Given that Blokus Duo is relatively new there has not been much research conducted on it, which makes it challenging and intriguing. In the following subsections we first present the history of the Blokus Duo game and then we explain the rules. Next, we mention some of the properties that characterize the game and finally we describe existing programs that implement competitive Blokus Duo players.

2.1.1 History

As stated above, Blokus Duo is an expansion of the main game of Blokus. The latter is a strategy board game of two or four players, invented by Bernard Travitian. Blokus was first released in 2000 by the French firm Sekkoïa but in 2009 the rights of the game were passed to the famous toy manufacturing enterprise [Mattel](#). Blokus Duo itself, was first introduced in 2005, implying that the game is relatively new. However it captured the interest of the research community in 2013, when it became the challenge of the design competition of the [ICFPT 2013 Design Competition](#). To the best of our knowledge, Blokus Duo tournaments are currently held in Asian countries and are less prevalent in the western community.

2. BACKGROUND

2.1.2 Rules

In order to explain how the Blokus Duo game is played we adapted the [rules](#) published by Mattel.

Components

- 42 game pieces (tiles) in total - Two 21-piece sets (usually one of orange and one of purple color). Each set contains 21 pieces of different shape shown in [Figure 2.1](#). A tile comprises unit squares i.e. the little squares that compose a tile. There is 1x1-unit tile, 1x2-unit tile, 2x3-unit tiles, 5x4-unit tiles and 12x5-unit tiles.
- A game board with 14x14 grid size, i.e. 196 squares.

Goal

Each player aims to place as many of his 21 pieces as possible on the board.

How to play

- 1 Each player is assigned a color (orange or purple) and gets the respective tiles. Both players are allowed to flip or rotate any tile before placing it on the board.
- 2 Whoever decides to play first is deemed as Player 1 and the other player is considered as Player 2. In the beginning of the game, Player 1 places any tile he wants in any way at a specific starting point on the board with coordinates (5,5). Then, Player 2 does the same thing but his starting point has coordinates (10,10) or (a,a) of the example board in [Figure 2.2](#).
- 3 The game continues as each player places one piece at a time on the board. The players play in an alternate manner and should follow the following restrictions:
 - The to be placed tile must have at least one corner-to-corner contact with a tile of the same color, as shown in [Figure 2.3](#)
 - The to be placed tile must not have edge-to-edge contact with any tile of the same color, as shown in [Figure 2.4](#)

- Different colored tiles can touch in any manner
 - The position of a placed tile cannot be changed until the end of the game.
- 4 In case a player does not have any legal moves to make or cannot find one, that player should pass. From that point to the end of the game the player who passed his turn cannot lay any other piece on the board.
- 5 The game ends when any of the following conditions are met:
- One of the two players has placed all his tiles on the board
 - Both players have passed their turn
- 6 Once the game has ended, scores are calculated and the player with the highest score wins.

An example of a game is shown in Figure [2.5](#)

Score

Each player counts the total number of unit squares of their remaining tiles that were not placed on the board. Each unit square counts as '-1' and therefore adding unit squares gives a negative total. The player who gets the highest score wins. Specific bonuses are given in the following situations:

- There is a +15 bonus added to any player that places all of his tiles on the board
- Additionally to the previous +15 bonus there is also a +5 one if the last tile placed on the board is the 1 unit square piece

An example of a completed game and how the score is counted is shown in Figure [2.6](#)

2.1.3 Strategy Tips

As all games, Blokus Duo also has some fine strategic points that may determine the outcome of the game. We present some of these, to provide a better understanding of the game.

2. BACKGROUND



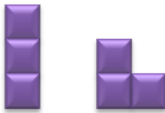
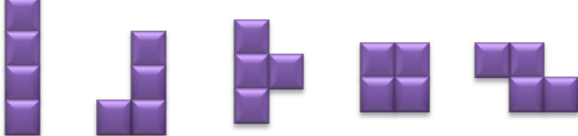
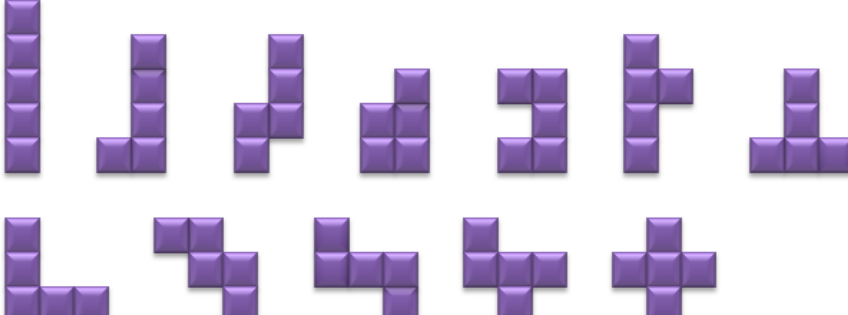
	1 x 1-unit tile
	1 x 2-unit tile
	2 x 3-unit tiles
	5 x 4-unit tiles
	12 x 5-unit tiles

Figure 2.1: Tiles of the Blokus Duo game. Each tile comprises from one to five units.

- Given that the score is computed based on the non-placed unit squares, each player should aim to minimize this number. Therefore the bigger the tile, the bigger the urge to place it in the first rounds of the game since later on, it may not fit on the board. One of the common mistakes that new players make is playing a 4-piece when a 5-piece can accomplish exactly the same thing. Note that most of the 4-pieces are contained within quite a few of the 5-pieces and so the latter could be placed when possible, since they are worth more points
- Each player should try and move towards the center of the board from the beginning of the game. The center is of great strategic importance because it is both, a safety net and a control territory. The former, because it provides multidirectional ways

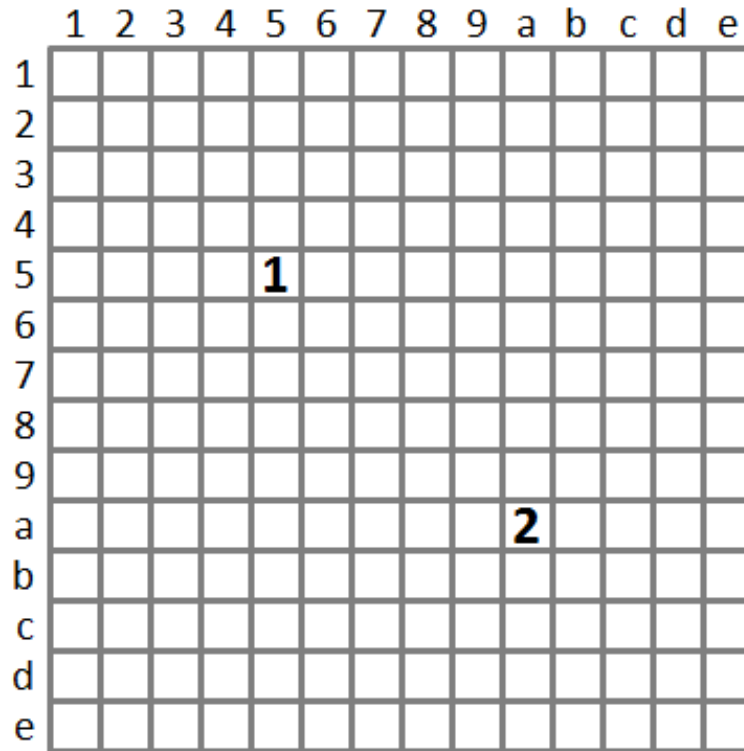


Figure 2.2: Board of the Blokus Duo game. Square 1 indicates the starting point of player 1 and square 2 indicates the starting point of player 2.

out in case a player is blocked and the latter for basically the same reason but now instead of ways out, the center provides paths that lead to areas that can be dominated.

- While playing, a player must think forward before deciding to fight or to flight. More specifically, in a situation where the opponent has claimed a big area somewhere, one might think of making a move to fight back by reducing the number of the opponent's corners or by trying to lessen his space. However this is a risk move because without realizing it the player might let his opponent invade his small space while his attack may evidently have no impact. Another approach would require giving up some space but making it hard for the other player to access the area you already control. Therefore, before rushing into making any reckless moves, one should try and figure the opponent's possible plans and then act appropriately.

2. BACKGROUND

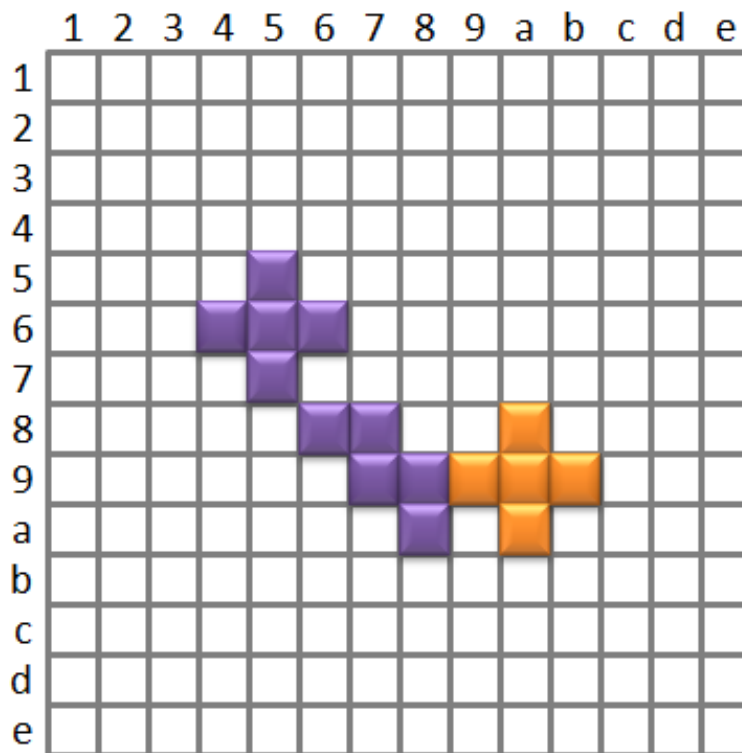


Figure 2.3: The purple tiles are connected with a corner-to-corner contact.

- The basic principle of Blokus Duo is to cover as much space as possible. Both players struggle throughout the game to prevail on the space of the board. Whenever a player controls an area on the board it is more likely that he will dominate it. Hence, it is important to know which areas are the bigger ones and aim to control these first. All areas of the board are shown in Figure 2.7.
- Detect areas that are open and have usable space, while making moves that will either lead towards the direction of these areas or that will yield good plays in that space later on in the game. For example, one could create available corners in these areas, in order to link other pieces easily and attempt to control that space. In general, due to the corner-to-corner attachment rule, creating corners implies more available positions to place a tile. However, the same applies for the opponent too and therefore, a player must focus upon creating corners for himself, while blocking the opponent's corners.

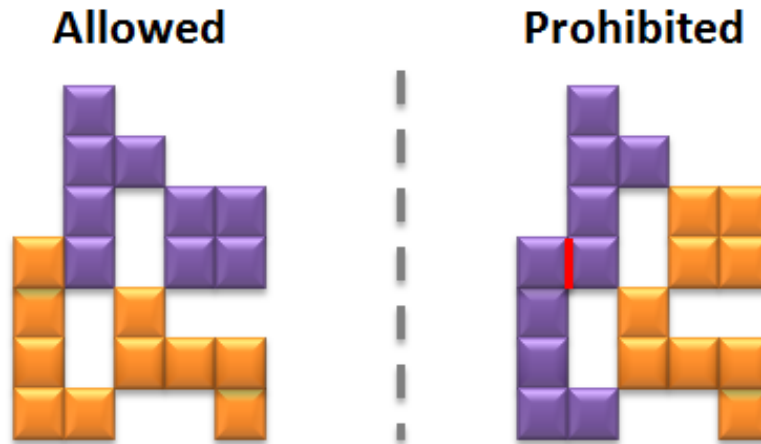


Figure 2.4: The left move is allowed, since same colored tiles are connected with a corner-to-corner contact but do not have any edge-to-edge contact. The right move is prohibited, since the purple tiles have an edge-to-edge contact.

- Some tiles must be placed carefully in the early stages of the game. More specifically, some tiles are sneaky ones as they do not only provide corners but they are also flexible to fit in small holes. If played appropriately, they can assist other tiles to be placed towards the end of the game, when each player tries to have as few tiles as possible left.

2.1.4 Existing Programs

Due to the fact that Blokus Duo is a quite recent game compared to other popular AI games, such as Go and Chess, there are only few reliable Blokus Duo computer programs. Among these we consider only three reliable, [Pentobi](#), [metablok](#) and [block'em](#), all licensed by GNU. Although Pentobi surpasses by far all other attempts, it does not come with a documentation while the others mention that they are Minimax based agents.

2.2 Decision theory

The basis of **decision theory** arises from the combination of the principles of probability theory and utility theory. Probability theory can be defined as how knowledge affects an

2. BACKGROUND

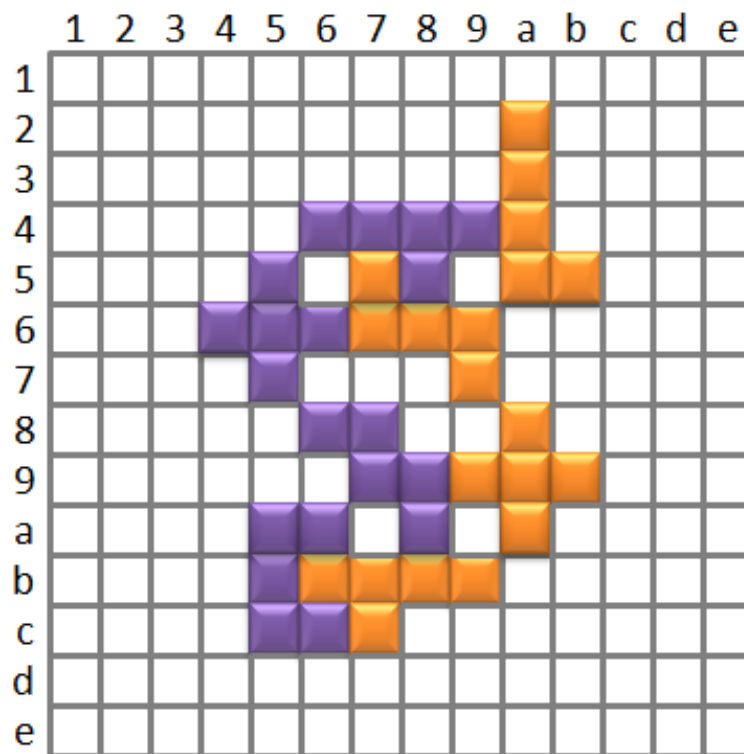


Figure 2.5: An example snapshot of the game.

agent's belief, while utility theory presents actions that an agent prefers to do. Decision theory uses both, to form an explicit and thorough framework for decision making in an environment of uncertainty [2, p. 9]. The main concept of decision theory is that an agent is rational if and only he selects among all actions, the one that will yield the highest expected utility, averaged over all the possible outcomes of the action [2, p. 465] In the following subsections we first define the term *Markov decision process* that studies sequential decision problems and then we present key applications in which decision theory is used.

2.2.1 Markov decision processes

In the case of a fully observable environment, **Markov decision process (MDP)** is a stochastic process that uses the Markovian transition model and additive rewards to model the dynamics of the environment under different actions.

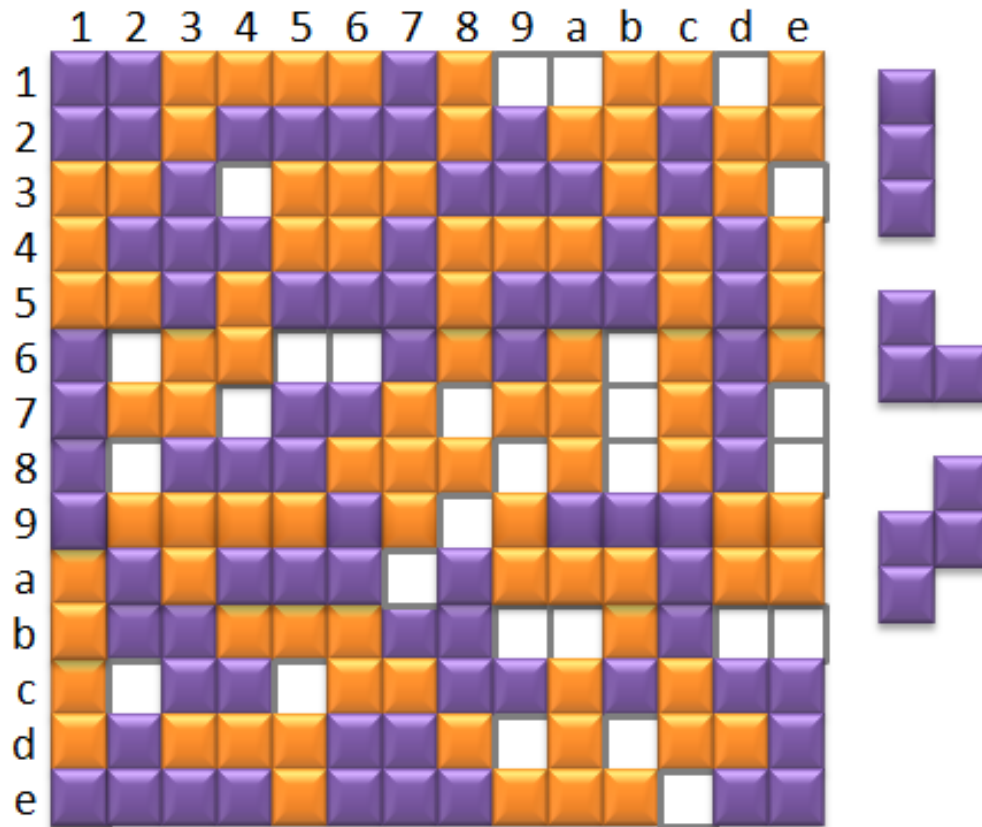


Figure 2.6: An example of a completed game. The orange player placed all of his tiles, hence he gets a bonus of 15 points. The purple player did not place 2 three-unit tiles and a four-unit tile. Therefore, his score is $2 * (-3) + (-4) = -10$. The orange player has the highest score and is the winner of this game.

Formally MDP is a 4-tuple that comprises the following four components:

- S : A set of states
- A : A set of actions
- $T(s, a, s')$: A transition model that determines the probability of reaching state s' if action a is applied to state s .
- $R(s)$: A reward function

2. BACKGROUND

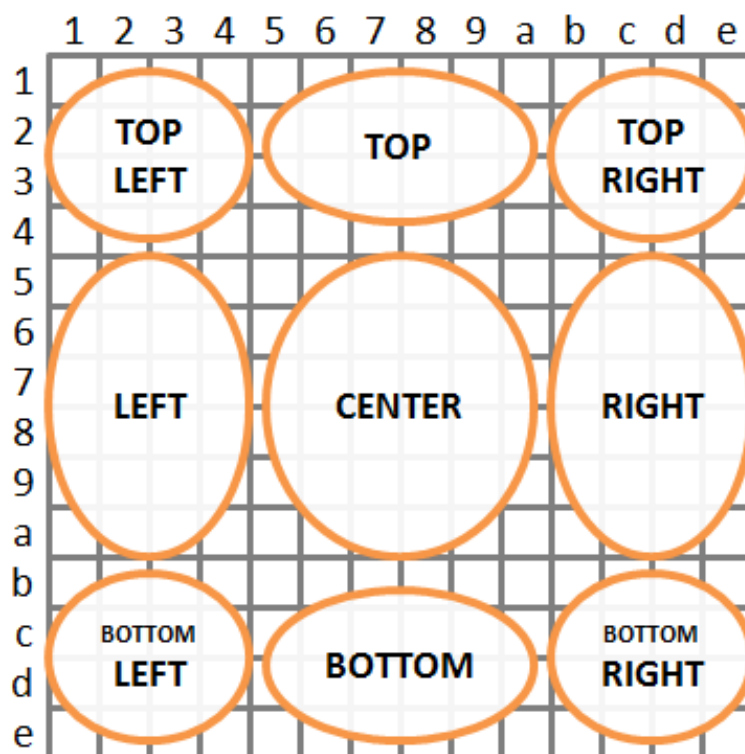


Figure 2.7: Important areas on the Blokus Duo gameboard

The decision-making process is depicted by sequences of state-action pairs. Each step requires the transition to a next game state. This state is determined by a probability distribution which depends on the current state s and the selected action a . In many cases, an agent might end up in positions that deviate from the initial goal. However he must be able to adapt to the new environment and make appropriate decisions given the new situation. Therefore a solution must specify what an agent should do for any state that is reached and this is defined as a policy. We usually denote policy by π and $\pi(s)$ is the action recommended by the policy π for the state s . We aim to converge to an optimal policy, i.e. the policy that yields the highest expected utility [Norvig 17 pg 615].

2.2.2 Applications

The name alone of decision theory implies its application on various decision problems. More specifically, it helps reach rational decisions in significant domains, where deter-

mining actions is crucial and the stakes are high. Examples of such fields are business, government, law, military strategy, medical diagnosis and public health, engineering design and resource management [2, p. 604]

2.3 Game theory

Game theory studies interactions that occur among rational agents in situations of competition, in order to achieve the best outcome in a game. Given that a game is entirely defined by decisions that lead to actions, one realizes that game theory is an extend of decision theory in the domain of games. Games are in general, an intriguing class of decision problems, as they have practical significance and their solutions can be easily extended to other problems too. In the following subsections we first present what is a game in the sense of game theory, we mention the basic elements of a game tree and finally, we define the term *combinatorial games* that consists of a class of games frequently used in game theory.

2.3.1 Games

Games have always been played among people, but their theoretic approach is a matter of the last century. A precise definition of a **game** is given by Salen and Zimmerman in [3]. *A game is a system in which players engage in an artificial conflict, defined by rules, that results in a quantifiable outcome.* The formulation of the above definition was completed after studying essential elements of many game definitions, given in prior studies. According to [4], significant components of a game are the following:

- *Players*
- *Actions*
- *Payoffs*
- *Information*

By putting together these elements we get the rules of a game that aim to describe any game situation. A game player is an individual that throughout the game, performs actions that will eventually maximize his rewards. Therefore he devises plans, known

2. BACKGROUND

as strategies, that point out which actions are appropriate to make, given any current useful information provided by the game. The interaction between different strategies determines the final outcomes and is the equilibrium of the game. An action or a move by a player is a choice he must make and in most parts of the game there is a set of actions he can perform. A player's payoff is a value that depends on the strategies followed throughout the game and is calculated based on a utility function. This value is usually arbitrary and can either be positive, indicating the number of accumulated points, or negative implying momentary loss. However, the reward in final game states is typically depicted by the values +1,0,-1, showing win, draw or loss respectively. Finally, the term *game scenario* is frequently encountered to describe all steps that occur in a game starting from the beginning of the game or from a current position until the very end of it.

2.3.2 Game tree

A **game tree** (or, game in extensive form) is a directed graph whose nodes are game states and the oriented edges represent possible moves that can be performed from a given node-state. The root of the tree depicts the initial or current position of the game, while the leaves of the tree indicate terminal states of the game. Game trees are important to AI, since they constitute a formal description of how a non-cooperative game can be played, thus allowing search for different outcomes and scenarios to be performed on the game.

2.3.3 Combinatorial games

What distinguishes combinatorial games from classic games is that they are bounded by a specific set of conditions:

- *2-player*: A game played by two players.
- *Zero-sum*: A player's gain or loss is exactly balanced by the losses or gains of the rest. If the total gains of the participants are added up, and the total losses are subtracted, these will sum to zero.
- *Perfect Information*: Players are fully aware of the board condition and the set of available moves

- *Deterministic*: There are no chance elements in the game
- *Finite*: The game ends in a finite number of moves no matter how it is played

Well known combinatorial games are Chess, Go, Tic-Tac-Toe or Nim. Note, that solitaire puzzles can also be considered as combinatorial games, assuming that the participating two players are the puzzle designer and the puzzle solver.

2.4 Minimax with alpha-beta pruning

In actual games it is common to evaluate a player's performance in the terminal states of a game. That's why many algorithms use game trees to depict possible complete scenarios and outputs of the game. Minimax with alpha-beta pruning is until recently, the most applied algorithm on combinatorial games that involves a game tree.

2.4.1 Minimax with alpha-beta pruning Algorithm

Minimax with alpha-beta pruning is an algorithm that takes into account every possible move that the opponent might make and performs a recursive depth-first search exploration with "back-ups". The goal of Minimax is to estimate the best move in the actual game round and thus the information from the game tree is used appropriately. In order to evaluate game states, Minimax with alpha-beta uses the Minimax value which is defined as follows:

$$\begin{aligned} \text{Minimax-value}(n) = & \\ & \text{UTILITY}(n) \text{ If } n \text{ is a terminal node} \\ & \max_{s \in \text{successors}(n)} \text{Minimax-value}(s) \text{ If } n \text{ is a MAX node} \\ & \min_{s \in \text{successors}(n)} \text{Minimax-value}(s) \text{ If } n \text{ is a MIN node} \end{aligned}$$

MAX selects the move that maximizes the Minimax value, while MIN selects the move that minimizes the Minimax value. The utility value calculated in terminal states represents a player's strategy, since it sums up all the factors that the player considers possible to maximize the likelihood of winning. Furthermore, the utility value is propagated from each leaf towards its predecessors, selecting either the minimum or the maximum value at each level. The first, is propagated whenever the opponent selects a move, since he desires

2. BACKGROUND

to minimize our score, while the latter is the reverse case, since we desire to maximize our score and play the best move.

Note, that the number of game states to evaluate is exponential in the number of moves. In order to avoid evaluating game states that will evidently not affect the overall Minimax-value, the alpha-beta pruning method that practices a commonly used branch and bound technique is applied. Usually, the pruning that occurs does not only remove leaf nodes but entire subtrees too, thus making alpha-beta quite effective. Its basic principle is simple: Let us consider that from a current state the player has only one valid move to perform depicted by node n somewhere in the tree. If the player could select a better move m either at the parent node of n or at any point further up, then n will never be reached in the actual play. Hence, once we have reached that conclusion about n we can prune it. The alpha parameter is the value of the highest-value choice we have found so far at any choice point along the path of MAX, while the beta parameter is the value of the lowest-value choice we have found so far at any choice point along the path of MIN. Consequently, MAX nodes with a value higher than beta are pruned since MIN would never let us reach them and MIN nodes with a value lower than alpha are also pruned since MAX would never let us reach them [2, p. 169]. The respective pseudocode is presented in 1.

2.4.2 Complexity

- **Time Complexity:** $O(b^{\frac{d}{2}})$, where b is the average or constant branching factor and d the search depth of plies. In the case of a pessimal move ordering the maximum number of leaf node positions evaluated is $O(b * b * b..b) = O(b^d)$ which is the same as simple Minimax search. However, if the best moves are always searched first the time complexity reduces to $O(b^{\frac{d}{2}})$ allowing the search to go twice as deep with the same amount of computation.
- **Space Complexity:** $O(b * d)$, where b is the average or constant branching factor and d the search depth of plies.

Algorithm 1 Minimax with alpha-beta pruning

```

1: function MINIMAX(node, depth, min, max)
2:   if depth == 0 or Leaf(node) then return Evaluate(node)
3:   if node is MAX then
4:      $u \leftarrow \min$ 
5:     for all children of node do
6:        $value \leftarrow \text{Minimax}(child, depth - 1, u, max)$ 
7:       if  $value > u$  then
8:          $u \leftarrow value$ 
9:       if  $u > max$  then return  $max$ 
       return  $u$ 
10:  if node is MIN then
11:     $u \leftarrow max$ 
12:    for all children of node do
13:       $value \leftarrow \text{Minimax}(child, depth - 1, min, u)$ 
14:      if  $value < u$  then
15:         $u \leftarrow value$ 
16:      if  $u < min$  then return  $min$ 
       return  $u$ 

```

2.5 Monte Carlo Methods

In general, **Monte Carlo** (MC) approaches constitute a broad class of algorithms that repeatedly perform random statistical sampling experiments to provide approximate solutions to a variety of problems. The accuracy of the solution depends on the number of experiments that are conducted. MC methods have a significant impact on the field of AI for games and this, will be our main concern in this section. However, it is also applied in numerous other fields, such as physical sciences, engineering, applied statistics, finance and business.

In order to comprehend the basics of MC we formulate it with the use of mathematical terms: Consider a random variable X with a probability density function $f_X(x)$ which is greater than zero on a set of values X and a function g of X . Assuming that X is continuous, we get an n -sample of X , such as $(x^{(1)}, \dots, x^{(n)})$, and we calculate the mean

2. BACKGROUND

of $g(x)$ over the sample. Then we get the Monte Carlo value:

$$g_n(x) = \frac{1}{n} \sum_{i=1}^n g(x^{(i)}) \quad (1)$$

2.5.1 Monte Carlo simulations

In combinatorial game scenarios where each player follows his own strategy the final result may vary, implying uncertainty. The key feature of **Monte Carlo simulations** is that it models complete game scenarios and estimates how likely a resulting outcome is. For example, assuming that a player wants to perform a specific action, then by performing Monte Carlo simulations from that move and onwards in the game he can evaluate how likely this move will yield a win, a loss or a draw. A Monte Carlo simulation performs complete game scenarios by selecting a new random move in each game state until it has reached a terminal state. The final result (a win, a loss or a draw), as well as the initial random move that was made are stored and the process is executed repeatedly, as many times as possible, given that in real games there are usually time or resource limitations. Typically, hundreds or thousands of Monte Carlo simulations are performed in a single game round, each time using different randomly selected values throughout the game. By the end of the procedure we get a big number of recorded result-initial move pairs that are used to calculate the probability of reaching various outcomes.

Now that we have described how a Monte Carlo simulation method works, we adopt the notion of Gelly and Silver [5] and rewrite equation (1) in terms of a combinatorial game:

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{j=1}^{N(s)} \mathbb{I}_j(s, a) z_j \quad (2)$$

where, $Q(s, a)$ is the Monte Carlo value, $N(s, a)$ indicates how many times action a was chosen from state s , $N(s)$ shows the total number of times that different play-outs included state s , $\mathbb{I}_j(s, a)$ has either value 1 or 0 depending on whether move a was performed from state s in the j -th simulation or not respectively and finally z_j is the result of the j -th play-out that began from state s .

2.5.2 Uniform sampling in Monte Carlo

Some Monte Carlo methods perform uniform sampling to select actions. Such approaches have proven to be in some cases quite competent, as in the case of Sheppard in [6], who reached the top rankings of the world with such approaches. However in cases where uniform sampling is not that promising, heuristic biasing can potentially improve the credibility of the method. More specifically, the selection of a move is biased based on previous results and therefore the expected value of an outcome computed by the algorithm differs from the true expected value.

2.6 Bandit-Based Methods

In this section we formalize the properties that describe bandit-based problems and we provide definitions necessary for the comprehension of a bandit algorithm. Then we define the term *regret*, an essential parameter of bandit-based problems that each player aims to minimize. Finally, we focus on the Upper Confidence Bounds class of approaches and more specifically on the UCB1 approach.

2.6.1 Multi-armed bandit problems

The fact that in a game the distribution of the resulting outcomes is originally unknown for each action, led to the exploration-exploitation dilemma. More specifically, it is common for some to require a balance between exploitation-exploration, while others might want to regulate this trade-off under some considerations. Exploitation concerns moves that appear to be promising and exploration takes into regard those, that may seem suboptimal. The role of the **mult-armed bandit problem** is to model this exploitation exploration dilemma. The term *multi-armed bandit* derives from the analogy of the problem with a slot machine with multiple arms. In this analogy a player chooses a specific one from the finite number of the machine's arms. The reward that he gets is a sample from a certain probability distribution related to the selected arm. The ultimate goal of the player is to maximize the total reward through repetitive plays. This cumulative reward is equal to:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \dots = \sum_{k=1}^{\infty} r_{t+k} \quad (3)$$

2. BACKGROUND

A formulation of the multi-armed bandit problem follows:

Formulation of the multi-armed bandit problem: We consider a fixed number of arms N . The reward received when choosing arm i is a sample from a distribution P_i . Both this distribution and the expectation of arm μ_i are unknown to the player. Successive plays of bandit i yields rewards which are identically and independently distributed (iid).

Note, that the finite number of arms allows the conduction of numerous experiments. In some cases all available arms are explored multiple times, thereby approximating more precise averages μ_i and estimating the sample distribution of each arm. Hence, the player can decide to exploit bandits that currently appear promising due to their high averages. Also a policy may be chosen to indicate which arm to play at each time step t . This policy is usually formed based on observed past rewards.

2.6.2 Regret

One of the most fundamental performance measure in multi-armed bandit problems is regret, since it is a measure of success in the exploration-exploitation dilemma. **Regret** is defined as the expected difference between the best possible cumulative reward and the sum of rewards actually gained at time step t :

$$R_N = \sum_{i=1}^M (\mu^* - \mu_i) \mathbb{E}[T_i(n)] \quad (4)$$

where, n indicates the number of games, $\mu^* = \max\{\mu_i : 1 \leq i \leq M\}$ i.e. the best possible expected reward, μ_i denotes the actual expected reward for arm i and finally $\mathbb{E}[T_i(n)]$ declares how many times arm i is expected to be selected in the first n games. Choosing the bandit with the currently highest expected reward may not result in the highest overall reward because another more optimal bandit might not have been detected yet. Nevertheless, playing a new machine is not necessarily a better solution since its reward remains unknown and it can increase the regret.

Although regret is more than helpful when we are trying to achieve a good exploitation-exploration balance, we might eventually aim to bound it to a constant after a certain number of iterations or a specific amount of execution time. Therefore, lots of studies have tried to examine the rate of bounds on the expected regret in time and conceive their dependency on the number of arms N of the bandit problem.

2.6.3 UCB1

Lai, Robbins et al. proved in [7] that no policy can achieve regret that grows slower than $O(\ln n)$ for a vast class of reward distributions. Hence, a policy is taken into account to solve the exploration- exploitation dilemma, if the growth of regret is within the constant factor of the $O(\ln n)$ rate.

Auer et al. proposed in [8] the use of certain approaches simple to implement, that achieve logarithmic regret. One of them is the **UCB1** policy, originally suggested by Agrawal in [9]. UCB1 decreases the number of random simulations for apparent suboptimal moves, to exploit those that appear to be better. The mathematical formulation of this policy is:

$$UCB1(j) = \bar{X}_j + C * \sqrt{\frac{2 * \ln n}{n_j}} \quad (5)$$

where, \bar{X}_j denotes the average reward for move j i.e. its winning percentage, n indicates the total number of executed plays and n_j shows how many times bandit j has been played. Constant C is called the exploration constant and has to be experimentally tuned to increase or decrease the ration between exploration and exploitation. According to the literature, the C coefficient may be initially set to $C = 1$ and either tuned towards allowing further exploration ($C > 1$) or towards allowing more exploitation ($C < 1$). Nevertheless, it should be appropriately adjusted, as it highly depends on the domain, the computational resources and the MCTS implementation.

Policy: UCB1

Initialization: Play each machine

Loop:

- Play machine j that maximises: $\bar{X}_j + C * \sqrt{\frac{2 * \ln n}{n_j}}$
- Get reward r_j
- $t = t + 1 \dots$

, where $j \in 1.., N$ number of machines

Factor \bar{X}_j urges the exploitation of actions that yield higher rewards, while factor $\sqrt{\frac{2 * \ln n}{n_j}}$ ensures that less visited game states will not be completely neglected.

2.7 Monte Carlo Tree Search

The term **Monte Carlo Tree Search** (MCTS) was first introduced in 2006 by Coulom in [10]. He proposed an innovative approach that integrated Monte Carlo evaluations with tree search algorithms. The same year Kocsis & Szepesvári incorporated in [11] the Upper Confidence Bounds into the MCTS algorithm, creating the UCT variant. Since 2006, there has been a burst in MCTS-based games and applications, especially in cases where the Minimax algorithm performs poorly. In this section first we explain in detail the MCTS algorithm. Then, we describe the main properties of the MCTS algorithm and also provide a brief background of some significant MCTS variations that are taken into account in this work.

2.7.1 MCTS Development

The inspiration for the development of the MCTS algorithm originates in the Monte Carlo methods. The term *Monte Carlo*, was conceived in 1946 by John von Neumann and Stanislaw Ulam. Initially these methods were applied in physical sciences but later on proved to be useful in other sciences too, such as engineering, applied statistics and artificial intelligence (AI) for games. Specifically in the field of AI, which is of our interest, Monte Carlo methods have been used extensively in various games. Further information about these methods have already been presented in 2.5. The weakness of the Monte Carlo methods is that due to the random sampling selection of the actions there are no game-theoretic guarantees that the algorithm will eventually converge to the optimal move. The transition from the Monte Carlo methods to the MCTS algorithm was first presented by Coulom in 2006 [10], where Monte Carlo evaluations were integrated with tree search. Briefly, the proposed algorithm gradually builds an asymmetric tree and repeatedly executes random simulations from a current game state to a terminal one. A current game state is calculated in every iteration and equals the one that has higher probability of being the best move. The same year another work, this time by Kocsis & Szepesvári in [12] boosted even more the interest of the research community for this novel algorithm. Kocsis & Szepesvári proposed adding child selectivity policies to the MCTS to reduce the error probability, when the algorithm is terminated prematurely. This probability error depends on the trade-off between exploitation and exploration that was described in detail in 2.6.1. The key to ensure balance between these two essential

factors is the use of multi-armed bandit policies also described in 2.6.1. It appeared that the UCB1 selection policy 2.6.3, presented by Auer et al. in [8] was the most promising among these, in terms of simplicity and efficiency. At the same time, the UCT variant of the MCTS family of approaches was frequently applied by Go players and results showed that it was highly effective in cases where other state-of-the-art AI approaches had failed.

2.7.2 MCTS Characteristics

This section presents the benefits and the drawbacks of the MCTS algorithm. The former, along with the algorithm's good performance, indicate its significance in solving challenging games and problems. On the other hand, the latter highlights which parts of the algorithm need to be enhanced for further improvement.

2.7.2.1 Benefits

The MCTS algorithm is characterized by notable benefits, that distinguish it from other approaches in the area of AI. These are the following:

- *Aheuristic*: The MCTS can perform well, even if there is no knowledge of the domain it is applied on. This is the key characteristic that boosted the algorithm's popularity, especially in games where a sufficiently good heuristic function has not been determined yet, such as in the game of Go.
- *Asymmetric*: The selection policy used in the family of MCTS algorithms, leads to an asymmetric tree growth, since MCTS aims to visit more frequently visited nodes that appear to be promising. More specifically, the construction of the tree is skewed towards regions of nodes, that are more probable to yield the best possible expected rewards. Note, that the shape of the tree can lead to a better understanding of the game.
- *Anytime*: The anytime characteristic implies that whenever the MCTS algorithm is stopped, it returns the currently best estimate. This is due to the fact that in the end of each MCTS iteration all nodes that participated are updated with the new values. This is a considerable aspect of the algorithm, especially in cases where a specific amount of execution time is given and the best answer should be received.

2. BACKGROUND

2.7.2.2 Drawbacks

As all algorithms, MCTS also has disadvantages that are taken into account in many studies. These are the following:

- *Playing Strength*: Although the MCTS algorithm is suitable for games with a vast move space, still in certain situations it fails to find even simple solutions in permissible time. In some cases the algorithm halts prematurely, limiting the exploration of the search tree and therefore neglecting certain good moves.
- *Speed*: Even in games with medium complexity, finding a good solution may require hundreds or millions of simulations and then the search performed in the search tree is not sufficient and the results are not reliable.

2.7.3 MCTS Algorithm

MCTS is a best-first search algorithm that creates and gradually expands a game tree, based on the results of random simulations. Once there are no resources left or enough samples have been gathered, the algorithm determines which is the optimal move based on values that have been computed for each game state. The pseudo-code for MCTS is presented in 2 based on [13].

In MCTS, game states are depicted by nodes. Each one of the nodes usually holds two pieces of information:

- *Value*: Indicates the winning percentage of the node.
- *Counter*: Shows the number of times the specific node has participated in a playout.

Usually, in the beginning of the MCTS algorithm the search tree contains only the root. The algorithm's structure comprises four basic steps, which are repeated until a terminal condition is met:

- *Select*: At this point, the tree is traversed from the root to a node that has not been entirely expanded yet.
- *Expand*: A new child is added to the selected node, therefore expanding the tree.

- *Simulate*: Starting from the expanded node, a self-play is conducted simulating game scenarios, until a terminal game state is reached.
- *Backpropagate*: The result received from the simulation is propagated from the expanded node backwards, to its ancestors, updating simultaneously their overall reward.

The above four steps are explained in detail, in [2.7.3.1](#) [2.7.3.2](#) [2.7.3.3](#) [2.7.3.5](#) respectively. Once the execution of the algorithm is terminated, MCTS returns the best node. The respective pseudo-code is presented in [2](#) and is based on [\[13\]](#).

Algorithm 2 Monte Carlo Tree Search pseudo-code

Input: *root_node*

Output: *optimal_move*

```

1: while (WithinComputationalBudget) do
2:   current_node  $\leftarrow$  root_node
3:   % Selection Phase
4:   while (current_node  $\in$  SearchTree) do
5:     last_node  $\leftarrow$  current_node
6:     current_node  $\leftarrow$  Select(current_node)
7:   % Expand Phase
8:   last_node  $\leftarrow$  Expand(last_node)
9:   % Simulation Phase
10:  result  $\leftarrow$  Simulation(last_node)
11:  % Backpropagation Phase
12:  while (current_node  $\in$  SearchTree) do
13:    current_node.Backpropagate(reward)
14:    current_node.visit_number  $\leftarrow$  current_node.visit_number + 1
15:    current_node  $\leftarrow$  current_node.parent
16:  optimal_move =  $\operatorname{argmax}_{N \in N_c(\text{root\_node})} (N.\text{best\_move})$ 
17: return optimal_move

```

2. BACKGROUND

2.7.3.1 Selection

All MCTS approaches begin with the **selection** phase. The term *selection* refers to the fact that we choose to follow and eventually expand a specific path of the tree. In order to determine this path, a child selection policy is employed starting from the root and is repeatedly applied until a leaf node is reached. Note, that leaf nodes either depict a terminal game state or a game state that has not been expanded yet. The simplest MCTS approaches use a greedy child selection policy during this first step, i.e. moves that have highest rewards are selected. However, the child selection policy is responsible for regulating the exploration-exploitation trade-off that was described in 2.6.1. Balancing these factors is crucial, since the value of a node alone does not indicate anything. For example, assume that a node is not extensively explored, due to poor sampling, but appears to have a high average reward score. This value is probably inaccurate compared to other ones that may have been derived from numerous playouts. Furthermore, the selection problem is similar to the multi-armed based problem described in 2.6.1, in a way that it chooses a next move that will lead to an unpredictable outcome. However, in the MCTS algorithm several such choices should be made, since starting from the root we first need to select a node from depth 1, then a node from depth 2, a node from depth 3 and so on. Another reason for which the child selection policy should be chosen wisely, is because it should be able to distinguish the good first moves. In many games, including Blokus Duo the game is significantly determined by the first moves. However, typical selection policies require that all children of a node are visited at least once before any of them is explored. Therefore, the required time to do so may be completely impractical, particularly when the game state space is vast at the first levels of the tree. An example of the selection phase is shown in Figure 2.8.

2.7.3.2 Expansion

In the **expansion** phase, one or more children nodes are added to the selected node, therefore expanding the tree. The simplest implementations add one child per simulation. Usually the number of children added at each expansion step varies, since it depends on the nature of the game and the computational budget. We assume that the selected node does not depict a terminal game state. Otherwise, the expansion step of the algorithm is

ignored since only updates will be required. An example of the expansion phase is shown in Figure 2.8.

2.7.3.3 Simulation

In the **simulation** phase a complete game scenario takes place to evaluate a certain move. It starts from the selected node of 2.7.3.1 and performs a self-play until a terminal game condition is reached. A similar procedure is followed by Monte Carlo based approaches where each move in the simulated game is selected by random sampling. Random sampling is the default policy of MCTS's simulation phase too, as it is simple to implement, domain independent and has low time complexity compared to domain based policies. Also, it is guaranteed that due to the nature of random sampling the whole game tree will eventually be covered. Still such simulations are rarely applied since they do not represent actual rational game players, therefore leading to inaccurate results and redundant computational cost. An example of the simulation phase is shown in 2.8.

2.7.3.4 Backpropagation

The end of a simulation triggers the final step of every MCTS algorithm, the **backpropagation** phase. Clearly the results that were produced in previous stages need to be recorded. This happens in a "backwards" manner starting from the expanded node (from where the simulation began) towards the root of the tree. Each node of the selected path increments its visit counter and updates the average estimated reward based on the simulation outcome. Note that this reward may either be the actual final score of the game (final score model), or an indication of whether the player won or lost, +1 and -1 respectively (win-or-lose model). An example of the backpropagation phase is shown in 2.8.

2.7.3.5 Final move selection

According to Chaslot [13], once the execution of the MCTS algorithm is completed it places in the actual game the move that is considered as the best. A best move may be determined according to one of the following ways:

- 1 *Max child*: The max child is the child that has the highest value.

2. BACKGROUND

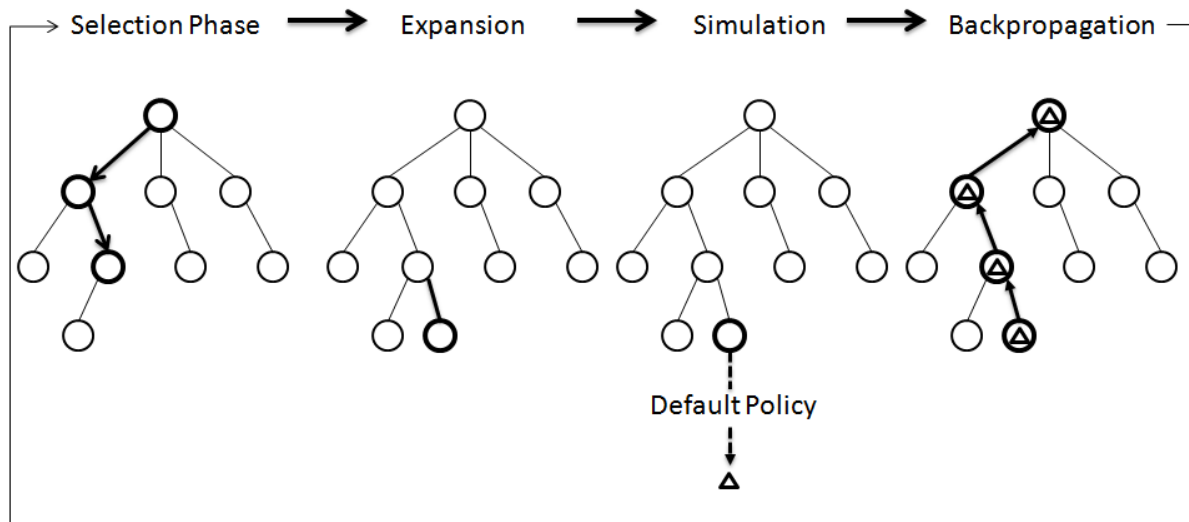


Figure 2.8: Four phases of the MCTS algorithm.

- 2 *Robust child*: The robust child is the child with the highest visit count.
- 3 *Robust-max child*: The robust-max child is the child with both the highest visit count and the highest value. If there is no robust-max child at the moment, more simulations are played until a robust-max child is obtained [10]
- 4 *Secure child*. The secure child is the child that maximizes a lower confidence bound.

However, when only a short thinking time per move was used (e.g., below 1 second), choosing the max child turned out to be significantly weaker than the rest of the selection approaches.

2.7.4 Upper Confidence Bounds for Trees - UCT

The Upper Confidence Bounds for Trees algorithm, known as **UCT** is a Monte Carlo planning algorithm first introduced by Kocsis & Szepesvári in [11]. It is currently the most common MCTS approach and is particularly known for its implementation on computer Go agents. This success is due to the fact that it was the first approach that improved the greedy child selection policy in MCTS 2.7.3.1, by using the UCB1 algorithm 2.6.3 to choose actions. The main difference between the greedy policy and the UCB1 approach is that the latter introduced an exploration factor that has higher value

in less visited nodes, whereas the former focuses only on exploiting seemingly interesting moves. Therefore, it ensures focusing on good moves without neglecting others that could later on be substantial.

Although the UCB1 score, shown in equation 5 is used in most implementations, as it is simple to calculate and guarantees to be within a constant factor of the best possible bound on the growth of regret, still it is not the best child selection strategy, as seen in the work of Tesauro et al. in [14]

2.7.5 MCTS Enhancements

2.7.5.1 Selection phase

In this section we describe enhancements targeted for the selection phase of the MCTS algorithm. These enhancements involve the bandit-based policy that is selected, the search that is conducted on the UCT tree, as well as suitable pruning that focuses the search on critical nodes.

UCB1-Tuned

In addition to UCB1, Auer et al. introduced **UCB1-Tuned** in [8] and, Gelly and Wang were the first to apply this policy to the UCT algorithm in [15]. The main difference between the two policies is that UCB1-Tuned takes empirical variance into account when calculating the upper confidence bound, to tune the UCB1 bound more finely. More specifically they used:

$$V_j(s, n) = \left(\frac{1}{s} \sum_{i=1}^s X_{j,i}^2\right) - \bar{X}_{j,s}^2 + \sqrt{\frac{2 * \ln n}{s}} \quad (6)$$

This is an estimate upper bound for the variance of the node j , where s shows the visit count of node j , and n denotes the visit count of the parent node of j . The term $\left(\frac{1}{s} \sum_{i=1}^s X_{j,i}^2\right) - \bar{X}_{j,s}^2$ computes the variance of node j . The final UCB1-Tuned bandit-based policy selects the child that maximizes the following value:

$$\bar{X}_j + \sqrt{\frac{2 * \ln n}{n_j} \min\left\{\frac{1}{4}, V_j(n_j, n)\right\}} \quad (7)$$

2. BACKGROUND

The results of Auer et al. showed that UCB1-Tuned outperformed UCB1 on all experiments but they could not provide theoretical guarantees for the regret of UCB1-Tuned, as they do for UCB1.

Other notable bandit based strategies include *MOSS* [16], *Bayesian UCT* [14], *Hierarchical Optimistic Optimization* [17] and *EXP3* [18].

First Play Urgency (FPU)

The notion of **First-Play Urgency** (FPU) for an MCTS algorithm was first introduced by Gelly and Wang in [15]. The basic MCTS algorithm suggests that all children of a parent node are visited at least once before any of them is expanded. However, in problems where the branching factor is big and the execution time is small some nodes may be never visited, let alone be exploited. *First-Play Urgency* suggests that a certain value should be assigned to each unvisited node. The idea is to select unvisited nodes according to this value and then once they have been explored at least once, exchange this value with the selection bandit-based policy. For their Go player MoGo, Gelly and Wang set the FPU value equal to 1000 to ensure exploration of each move at least once, before encouraging any exploitation of an already visited move. However, First-Urgency Play can be used to allow exploitation of promising nodes even from the early stages of the MCTS algorithm.

Progressive Bias

This technique introduces domain knowledge to lead the search towards potential moves. **Progressive bias** was used by Chaslot et al. to improve the performance of their Go player MANGO [19]. More specifically, they do not use the UCB1 policy but they introduce an additional term:

$$f(n_i) = \frac{H_i}{n_i + 1} \quad (8)$$

where, H_i is a heuristic value for the node i and n_i denotes the number of times this node has been visited. During the selection phase, they choose children that maximize the following value:

$$\bar{X}_j + C * \sqrt{\frac{2 * \ln n}{n_j}} + f(n_i) \quad (9)$$

In cases where a node has few statistical information, inserting heuristics may be crucial to quickly find valuable moves. However, the progressive bias term influences the selection policy, only when a node has been visited few times, since as the number of n_i visits increases, the $f(n_i)$ value decreases and the strategy converges to the a selection strategy. Some approaches suggest that if H_i is slow to compute, then progressive bias should not be inserted to the selection policy until a node has been visited a fixed number of times. Therefore the number of simulations and the speed of the MCTS algorithm are not reduced substantially.

2.7.5.2 Simulation phase

We mentioned in 2.7.3.3 that during the simulation phase of the MCTS algorithm, moves are selected randomly from a set of available actions, without incorporating any domain knowledge. However, these simulations rarely resemble actual game scenarios, thus reducing a player's performance. This led to a class of approaches that aim to enhance the simulation policy.

Evaluation Function:

One approach suggests the use of an evaluation function to choose a move, instead of selecting randomly. However, as in Minimax, the evaluation function crucially affects the outcome of the game, since it biases the player's game strategy towards either good or bad moves. Winands and Björnsson describe in [20], that the most effective strategy for designing an evaluation function is to avoid making bad moves in the beginning of the game, but playing greedily towards the end.

Score Bonus:

We mentioned in 2.3.1 that the reward value of a final game state is typically +1,0 or -1, depicting win, draw or loss respectively. It is typical for a UCT implementation to represent the outcome of the simulation phase with the use of these values. However, these values simply indicate whether there was a win, a loss or a draw, but do not reveal anything about the score difference that occurred. For example, moves that led to a strong or a weak win are not distinguished and are assigned with the same value. A way to deal with this issue, is by backpropagating different values in the interval [-1; 1]

2. BACKGROUND

depending on the score difference, but there is no reported case, where this technique actually improved a player's strength.

Chapter 3

Related Work

3.1 Blokus Duo

In this section we focus on works that have created Blokus Duo players. First, we present the only approach found, that uses the MCTS approach and then we describe four other Minimax-based implementations.

3.1.1 Blokus Duo MCTS approach

Shibahara and Kotani in [21] proposed an MCTS player for Blokus Duo. As it is mentioned in 2.7.3.1, MCTS approaches select moves according to a highest mean value, that in most cases is the UCB1 policy in 5. Shibahara and Kotani used the winning percentage value, \bar{X}_i to choose moves. In general, the winning percentage may either be calculated based on the final score model or the win-or-lose model described in 2.7.3.5, although in most cases the former has proved to be inferior to the latter. However, they conducted a comparison between the two models in order to study their behavior in the game of Blokus Duo and for this purpose they implemented a UCT (uses win-or-lose model) and a UCB (uses final score model) player. The following observations were made:

- 1 For a small number of simulations, UCB did not perform any different than UCT.
- 2 Unlike UCB, UCT can hardly exploit advantages of the final score model.
- 3 The credibility of the final score value depends on the number of simulations performed. The bigger the number of simulations the more accurate the final score.

3. RELATED WORK

Based on the above conclusions Shibahara and Kotani decided to integrate final score into the UCT algorithm somewhat efficiently. More specifically, they thought of combining the two scores, by using a sigmoid function. The formula of the proposed function is the following:

$$f(x) = \frac{1}{1 + \exp^{-kx}} \quad (6)$$

where, x denotes the final score and k is a constant. Whenever k is increased, it gets closer to the win-or-lose value and vice versa. Based on observation 3 of 3.1.1 the authors decided to adjust the value of k according to the phase of the Blokus Duo game, since according to the phase a different number of simulations may be executed. The experimental results showed that with the use of the sigmoid function, the UCT player could achieve at most 54% winning average and that the final score proved to be effective only in the final stages of the game. Apart from the experimental results, they also asked 5 Blokus Duo beginners what they thought of the sigmoid function-based player and what of the win-or-lose-based player. The majority decided that the latter player was stronger, but the former played in a more humanly manner and was more amusing.

Since both, the MCTS algorithm and the game of Blokus Duo are relatively new, there were no other works that combined the two. However, due to the fact that the game is more popular in Asian countries there may be MCTS-based approaches described in a language we are not familiar with.

3.1.2 Blokus Duo Minimax based agents

Of the three players described in this section the first two were introduced in the recent [FPT 2013 Design Competition](#). Other players were also presented in the same conference, but we did not have access to their implementation details.

The first player proposed by Jiu Cheng Cai et al. in [22] is a hardware-based approach. It implements the basic Minimax algorithm with alpha-beta pruning, with the use of the LegUp open-source HLS framework. The innovation provided by this work concerns the heuristic applied by Minimax. More specifically, four criteria are taken into account:

- *Number of squares placed on the board*
- *Number of corners adjacent to an opponent's tile*

- *Influence area*: It calculates available empty space around the player's tiles where new tiles can be possibly placed.
- *Weighted reachability*: To calculate this parameter they executed a breadth-first search (BFS) on the empty squares in the game board. A search is conducted for each empty square that is diagonally adjacent to a tile. However the search is conducted in a horizontal and vertical manner, not a diagonal one. During this procedure, each square is assigned a cost that decreases proportionally to its distance from the initial square.

Experimental results showed that of all the heuristic parameters, the most significant ones were first the weighted reachability factor and then the influence area.

The second player proposed by Erik Altman et al. in [23] is also a hardware-based approach that implements the Minimax algorithm, but there is no mention of any pruning. The authors aimed to test whether a high-level language can provide results whose quality could match a hardware designed with standard tools. From the aspect of AI and the Minimax algorithm they do not describe the evaluation function in an accurate manner. However, the authors present some of the heuristics they tested during their experiments, which are the following:

- *"Decreasing width"*: The Minimax search seeds the tree with a certain number of initial moves. For each game state a specific number of children is added and this number decreases while the depth of the tree increases. Once a few moves have been searched ahead, the width parameter descends to 1. At subsequent depths only the best move of each position is evaluated.
- *Square influence*: Similarly to parameter 3 of 3.1.2 the evaluation function estimates in a finite neighborhood of squares, directions towards which the player could possibly place tiles later on in the game.

Sha Huang proposed in [24] another Blokus Duo agent. Only, this time it is implemented in software. More specifically, he uses the Minimax algorithm with alpha-beta pruning, in conjunction with iterative deepening depth-first search to find the optimal move. Again, what differentiates this work from others are the parameters used in the evaluation function. The author implemented two groups of evaluation functions and

3. RELATED WORK

evaluated their performance. The 1st group comprises three evaluation functions each one of which is applied in a different stage of the game:

First 3 moves

$$value = a - b \quad (7)$$

where, a equals the number of squares between the opponent's starting point and the piece before placing it and b equals the number of squares between the opponent's starting point and the piece after placing it.

Moves 4 to 16

$$value = 3 * c + 5 * d + 2 * e \quad (8)$$

where, c denotes the number of unit squares of the piece to be placed, d shows the player's incremental number of corners after placing the piece and e , the opponent's decreasing number of corners after placing the piece

Moves 17 until the end

$$value = 3 * c + 3 * d + 4 * e \quad (9)$$

where, c , d and e are the same as the previous equation.

The 2nd group consists of only one evaluation function, that was inspired by the game of Go. More specifically, it introduces the valid grids factor that computes squares where the player can place a new piece:

$$value = 10 * a - 5 * b + 10 * c + d \quad (10)$$

where, a equals the number of squares between the opponent's starting point and the piece before placing it and b equals the number of squares between the opponent's starting point and the piece after placing it, c indicates the opponent's decreasing number of corners after placing the piece and d is the difference of valid grids between the two players. Experimental results showed that the 2nd group, that is actually a single evaluation function outperformed the 1st group in all cases.

3.2 Open source Go MCTS implementations

The MCTS algorithm was first applied on Go computer players (Gelly et al. in [25] with their program MOGO), where it performed exceptionally well compared to other algorithms. Since then, many have created UCT-based Go players and others have dedicated their time into enhancing their agents, by applying optimizations or different techniques. Therefore, we considered it useful to study such approaches that have been evaluated over and over again, in order to comprehend better how certain MCTS variations work and identify which of these are proven to be the best. Both of the UCT-based Go implementations described in this section, are open source codes that come with respective documentation.

3.2.1 Fuego 1.1 Version

The Go computer player, named [Fuego](#) was the first to beat a top human professional at a 9x9 Go. Before searching to find an optimal move for the current board status, Fuego will try and lookup the move in the Go book. In case that does not generate a move, a search is conducted. Note, that Fuego can perform a UCT search, One-ply Monte Carlo search or no search at all and select a move according to a policy. However, we will focus on the UCT approach since it is the main purpose of this thesis. The Top level search architecture comprises the following stages:

- 1 The UCT tree is initialized.
- 2 The UCT search is performed by repeatedly playing games, while gradually the UCT tree is build. When the tree cannot be further expanded the search is completed.
- 3 In the end, game states with low count are pruned and the best move sequence is found.

We emphasize on steps 2 and 3 of [3.2.1](#), since step 1 is quite trivial.

UCT search

The UCT search is conducted as follows. The `PlayGame()` function is called in a recurrent manner until the UCT tree cannot be further expanded. The subfunctions that

3. RELATED WORK

interest us the most during the `PlayGame()` are: 1)`PlayInGame()`, 2)`PlayoutGame()`, 3)`UpdateValues()`

- **PlayInGame()**: is responsible for expanding nodes in the UCT tree. It stops expanding nodes when a proven win/loss occurs. In other words, `PlayInGame()` executes the selection 2.7.3.1 and expansion 2.7.3.2 steps of the MCTS algorithm. First, it generates all legal moves and creates their children. Then it selects one child and executes this move in the UCT tree. The child selection is performed according to the UCT Bound Formula:

$$UCTBound = \bar{X}_j + C * \sqrt{\frac{\log n}{T_j(n)}} = Estimated\ move\ value + UCTbias \quad (11)$$

For the calculation of the Estimated move value (EMV) or \bar{X}_j they use a combination of Move and RAVE values, while for the calculation of the UCTbias they compute:

$$UCTbias = C * \sqrt{\frac{2 * \log\ visits}{1 + plays}} \quad (12)$$

where, constant C equals 0.7, *visits* denotes the number of times the node was visited and *plays* the number of times the move participated in a playout

- **PlayoutGame()**: The `PlayoutGame()` function is performed out of the UCT tree and corresponds to the simulation phase of the MCTS algorithm 2.7.3.3. The playout phase repeatedly produces moves until a null move is generated i.e. has reached a terminal game state. The move generation is not done in a random manner but according to a series of specifications that the placed tile must meet. This is the playout policy.
- **UpdateValues()**: During this phase all appropriate nodes of the tree should be updated to the new statistics and RAVE values.

The above were derived from the study of Grace I. Lin in [26]

3.2.2 Pachi 10.0 Version

In his Master's Thesis [27] Petr Baudiš proposed **Pachi**, a Go computer player that currently competes in competitions and ranks among the best players. They have implemented an MCTS-based approach and enhanced it with a variety of published and

3.2 Open source Go MCTS implementations

original methods. Also, they aim to resolve some of the issues that arise from information sharing techniques. In this section we will describe some of the techniques employed by Pachi, focusing on those that are not intended only for the game of Go. More specifically, these methods are: 1)Simulation policy, 2)Prior Values, 3)RAVE, 4) Information Sharing, 5)Situational Information Sharing, 6)Horizon effect:

- **Simulation policy:** The typical simulation policy in MCTS suggests to randomly select moves [2.7.3.3](#). Pachi uses a biased simulation policy π_s , until a terminal state is found. This policy comprises several Go heuristics, applied in a fixed order. The first one matching is the one eventually applied. During the simulation, they do not intend to minimize only their prediction error, but the opponent's too to achieve combined error minimization.
- **Prior Values:** Prior Values are used to avoid the overhead of expanding all unvisited nodes at least once, since many of them are bad moves. In order to first explore moves that are probably good, a constant UCB value is assigned to all newly created nodes, such that if a move appears promising it will be further explored before others. The assigned constant value is computed, based on a set of heuristics that perform a static evaluation of the move. Furthermore, they use the progressive bias technique to apply heuristics that determine the number and the results of virtual simulations, eventually combined with the real ones.
- **RAVE:** During the MCTS execution, certain statistics and values are propagated to specific nodes. RAVE extends this information to a bigger class of nodes. However, it does not propagate only simulation results, but AMAF expectations too i.e. expectations that concern the node under study which has at any point contributed to the result of any random simulation
- **Information Sharing:** To achieve information sharing they applied the following methods:
 - Situational Information Sharing: In case most of the simulations performed for a move converge towards a conclusion, the score threshold can be adjusted accordingly.

3. RELATED WORK

- Horizon effect: There are certain situations, such as when the result of a move is not currently clear, where AMAF results are biased and may result in the selection of a wrong move. They suggest enhancing the heuristics to overcome such adverse cases.
- Local value: They assign a domain dependent value to a local position to estimate its local efficacy.
- **In-Tree Parallelization:** They have implemented lockless in-tree parallelization, where searches and simulations occur simultaneously by multiple threads.
- **Criticality:** The use of criticality allows to focus on powerful areas of the board. Criticality accumulates data that indicate dominance in final positions and correlates them with the probability of winning.

One of the most interesting features of Pachi is that it does not take into consideration the UCB child selection policy, but only uses RAVE values. They also do not perform any pruning on the tree

Chapter 4

Implementation

The whole project was implemented with C programming language, as no object-oriented overhead was required. Also, it is within the scope of this thesis to model algorithms for hardware implementation, hence we needed our program to have an abstraction level close to the case of embedded hardware. Each section of this chapter comprises two subsections, one that details how the software implementation was carried out and one that studies software in terms of hardware. Note, that we have already implemented the Blokus Duo components [4.1](#) and the Minimax with alpha-beta pruning player [4.2](#) in hardware. Although it is not in the scope of this thesis to describe in detail how the hardware was implemented, basic implementation issues are mentioned to predict how the rest of the players could perform in hardware. In the following subsections we first present how the Blokus Duo game components were depicted. Next, we describe the implementations and the potential hardware-based implementations of our three basic players, Minimax with alpha-beta pruning, Monte Carlo and MCTS.

4.1 Blokus Duo components

The two components of the Blokus Duo game are each player's tiles and the game board, that were described in [2.1.2](#). However, during implementation we considered efficient ways for their representation, in terms of execution time and memory allocation.

4. IMPLEMENTATION

4.1.1 Tiles

In the beginning of the game, each player has at his disposal 21 different tiles that can be flipped and rotated in any manner. Therefore the player does not consider 21 different tiles, but 168 (21 pieces x 4 rotations x 2 flips). However among these 168 tiles, some are symmetrical with respect to the x or y axis or sometimes both and thus there can be in total eight, four, two or one different representations of a specific tile. By eliminating all duplicates we no longer consider 168 distinct tiles, but 92, thus avoiding examining the same tiles multiple times. A Blokus Duo tile consists of at the most 5 unit squares, thus we needed a 5x5 array to be able to depict each one of them.

4.1.2 Game Board

In the Blokus Duo game the board is a 14x14 grid. However, due to the fact that each tile is represented as an array, we considered the board as a 16x16 grid, since in certain cases a valid tile position might be found in the bounds of the board. We observed that we needed a board of size 16x16 to "place" tiles and to find valid positions on the edges, but during execution we evaluated the 14x14 squares of the actual board i.e. we did not increase the number of possible valid positions.

4.1.3 Software Implementation

All tiles are included in an array of integers, with dimensions 21x5x5 i.e. `int pieces[21][5][5]`, that contains and depicts all basic 21 tiles. To be able to perform rotations we created another array of integers, with dimensions 8x5x5, i.e. `int rotate[8][5][5]`. Finally, in order to evaluate only distinct tiles and eliminate duplicates we created a third array of integers, with dimensions 92x2 i.e. `int pieces_dupl2[92][2]`, where `int pieces_dupl2[92][1]` indicates the tile's actual number (from 0 to 20) in the availability register and `int pieces_dupl2[92][0]` a possible rotation (from 0 to 7). For example `pieces_dupl2[0][1]` equals 0 corresponds to the first of the 92 distinct tiles that is located in position 0 in *pieces* array, while `pieces_dupl2[0][0]` equals 0 corresponds to the first of the 92 distinct tiles that has rotation 0 in the *rotate* array. Furthermore, we implemented an array of integers, with dimensions 2x21 i.e. `int available[2][21]`, to store for each player which tiles have been used and which are still available. In the this section, we refer to the actual global

available array either as available, or as availability register. Finally, we created an array of integers, with dimensions 16x16 i.e. `int board[16][16]` to represent the board for the reasons that were described in 4.1.2.

4.1.4 Hardware Implementation

Similarly to the software, in the hardware implementation each tile is represented by a 7x7x2 array. This representation offered a single cycle determination about whether a tile fit somewhere on the board. The FPGA's internal Block RAM-BRAM (90x98) was used for the storage of the tiles with their distinct rotations (without duplicates). The distinct rotations of a tile were stored together and in a specific sequential order to ease the implementation. Furthermore, a lookup table was used, so as to reduce memory accesses and swiftly index specific tiles. In order to remember which tiles had been used and which were still available, a 21-bit availability register was implemented for each player. Finally, the board was depicted by a 16x16x2 array. More specifically, the first two dimensions were set in such way, so as to represent each cell of the actual board while the last dimension indicated i) if the board cell is empty (00), ii) if the board cell is occupied by one of the players' tiles (01) or (10), iii) if the cell can be occupied by a tile due to a corner-to-corner contact (11).

4.2 Minimax with alpha-beta pruning player

We implemented the basic Minimax algorithm with alpha-beta pruning and used a simple heuristic, since it was not in the scope of this thesis to find a good Minimax heuristic function for the Blokus Duo game. The same heuristic function with different weights was applied in both, the software and the hardware.

4.2.1 Software Implementation

The creation of the Minimax with alpha-beta pruning player was based on algorithm 1. First, we present the structures that were used in our implementation and then some basic elements of the Minimax algorithm, but now specifically for the Blokus Duo game.

4. IMPLEMENTATION

4.2.1.1 Minimax Structures

For the implementation of the Minimax-based player we used two structures. The first contains all information necessary to describe a move, while the second is a list of moves. More specifically:

```
/* Keeps all information necessary to describe a move */
```

```
typedef struct {  
    int x;           //coordinate x of the board  
    int y;           //coordinate y of the board  
    int piece;       //number of tile (0-20)  
    int rotate;     //number of rotation (0-7)  
} move;
```

```
/* This struct is a list of moves, necessary to keep the children of a node. */
```

```
typedef struct moves {  
    move oneMove;    //current move of the list  
    moves *next;     //next move  
} moves;
```

4.2.1.2 Minimax Algorithm

In this section we present the basic functions of our Minimax approach, along with a brief description for each and then, we describe the execution flow of the algorithm.

Basic Functions:

- **move doMinimax()**: Starts the execution of the Minimax algorithm and returns the estimated best move of the player. It considers as root of the Minimax tree the global current state of the game. Therefore, it does not require inputs.
- **float getMax(int depth, float alpha, float beta, int piece)** : This function is executed whenever it meets a MAX node in the Minimax tree and requires as inputs:

4.2 Minimax with alpha-beta pruning player

- *depth*: Indicates the depth of the children of the MAX node in the tree.
- *alpha* and *beta*: These values are required for the alpha-beta pruning procedure.
- *piece*: The piece or tile that is in a node of the Minimax tree. This variable is used in the evaluation function.

The `getMax()` function first checks whether we have reached a leaf node or a certain depth of the tree. If this condition is met, it returns the value calculated by the evaluation function for the given node. Otherwise, it finds all valid moves that follow the MAX node. These valid moves are the MIN children nodes of the MAX node and are kept in a list of type *moves* mentioned in 4.2.1.1. Note, that the `getMax()` function considers one MIN child node at a time. In case there are no children left in the list to evaluate, the `getMax()` function returns the MAX value of its children. Furthermore, whenever the alpha-beta pruning condition is met in any MIN child node, the function returns, thereby terminating the evaluation of the rest of the node's children. For example, let us assume that a MIN node has found that the value 5 is the minimum among its MAX children nodes until then and that now it is evaluating the next MAX child node. In case the first value propagated from the leaves to the MIN child node under study is 8, there is no point in evaluating the rest of the MAX child node's subtrees, since even if a bigger value is found to replace 8, the parent MIN node would not consider it, as it has already found a MAX child node with value 5. Therefore, this MAX child node prunes the rest of its subtrees without affecting the overall Minimax value.

- **float getMin(int depth, float alpha, float beta, int piece)** : This function is similar to the `getMax()` function 4.2.1.2. The main difference is that now the MIN value is found and returned, as this function is executed whenever a MIN node is met. Also, it prunes subtrees and nodes in a different manner that will be clarified with the following example. Let us assume that a MAX node has found that the value 8 is the maximum among its MIN children nodes until then and that now it is evaluating the next MIN child node. In case the first value propagated from the leaves to the MIN child node under study is 6, there is no point in evaluating the rest of the MIN child node's subtrees, since even if a smaller value is found to

4. IMPLEMENTATION

replace 6 the parent MAX node would not consider it, as it has already found a MIN child node with value 8. Therefore, this MIN child node prunes the rest of its subtrees without affecting the overall Minimax value.

- **float GreedyEvaluate(int piece):** The GreedyEvaluate() function is executed whenever the algorithm reaches a leaf node or a certain predefined depth of the Minimax tree. It requires as inputs all parameters that are in the evaluation function. In our case we only use properties of the piece (tile) that is to be placed, as it is not in the scope of this thesis to create an optimal Minimax-based player. More specifically the evaluation function is the following:

$$value = 0.5 * a - 0.5 * b + 0.7 * c \quad (13)$$

where, a is the incremental number of the Minimax player's corners, b denotes the reductive number of the opponent's corners and c shows the number of unit squares of the piece.

- **moves returnAllValidMoves():** This function finds and returns a list of type *moves* objects mentioned in 4.2.1.1. These are all valid moves that can be performed given a current game state.
- **int check_move(int player, int turn, move m, int gameboard[16][16], int availability_reg[2][21]) :** Given a game state and a specific action to be performed, this function evaluates whether the action can be made or not. More specifically, for input coordinates, tile and rotation, *check_move()* examines the corresponding area on the board and determines the validity of the move. In order to do so, it needs to check a 5x5 area, since the size of the tile is 5x5.

The execution of the Minimax algorithm in our approach begins with the *doMinimax()* function. Since the root node of the Minimax tree is a MAX node, *doMinimax()* calls *getMax(depth,alpha,beta, piece)*, where *depth* has a predefined value, *alpha* has value -1000 , *beta* has value 1000 and *piece* has value 1 , although it could be anything. Given that *depth* is not equal to 0 , the following actions happen in *getMax()*:

1. *returnAllValidMoves()* function creates and returns a list of the root's MIN children nodes that represent valid moves in the game.

4.2 Minimax with alpha-beta pruning player

2. For the first MIN child node of the list, *getMax()* calls the *getMin(depth-1,alpha,beta,piece)* function, where *depth-1* is because the MIN child node is one level below the root, *alpha* and *beta* still have values -1000 and 1000 respectively and *piece* equals the piece value of the move represented by the first MIN child node.

At this point, given that the depth of the min node is not equal to 0, *getMin()* performs the same actions as *getMax()*, only now *returnAllValidMoves()* returns a list with the MAX children nodes of the first MIN node and for the first MAX node in the list *getMin()* calls *getMax(depth-1,alpha,beta,piece)*. This recursion continues until we have either reached a leaf node or *depth* equals 0. Then, regardless of whether we are in a MAX or MIN node, *GreedyEvaluate(piece)* computes the evaluation function for the specific *piece* and MIN or MAX node returns this value to its parent. Depending on whether the parent node is a MAX or MIN node, the *alpha* or *beta* value is updated respectively. The parent node will then consider the rest of its children nodes and will keep the maximum value among these if it is a MAX node, or the minimum value if it is a MIN node. Once, all children have been evaluated the parent node returns the maintained MAX or MIN value to his parent node and this process continues, until all children of the root node (all valid subsequent moves) have been evaluated. Finally, the best of these is returned to the *doMinimax()* function. Note, that if an alpha-beta pruning condition is met in a node, then this node returns its current value without searching the rest of its children. We do not provide a pseudo-code of the algorithm, because it is based on the one presented in 1.

4.2.2 Modeling for Hardware Implementation

In this section we present critical points of the software code that indicate hardware opportunities. In order to do so, we performed profiling of the code using the Linux GNU GCC Profiling Tool (gprof), we computed the size of the structs used in the software code, we detected potential parallelism, as well as potential bottlenecks.

4.2.2.1 Code Profiling

The results of the code profiling showed that greater time consumption occurred in the *check_move()* function. We expected such a result, due to the fact that the complexity of

4. IMPLEMENTATION

check_move is $O(n^2)$, as an 5x5 area of the board is evaluated. Also, knowing whether a move is valid is essential, since most of the rest of the functions depend on it. Therefore *check_move()* is called many times in different parts of the algorithm. The next most time consuming process is executed in the *returnAllValidMoves()* function. This result can be also interpreted, as we know that in order to find all valid moves, we need to consider all remaining tiles, with their respective rotations for each board position. Given that in the worst case the tiles are 21, the rotations are 8 and the board size is 14x14, it is clear that 21x8x14x14 iterations are needed to find all available moves, a number that affects significantly the overall execution time of the algorithm. Furthermore, *GreedyEvaluate()* is the next most time consuming process, as besides knowing the units of the tile to be placed, it requires knowing the available corners of both players before and after placing the tile. However, counting a player's available corners on the board is an expensive process, since a corner might be anywhere on the 14x14 squares of the board.

4.2.2.2 Memory Requirements

In order to determine the memory requirements of the Minimax algorithm, we present the size of the two structures that are involved in the specific algorithm that were mentioned in 4.2.1.1. The first is the structure *move* that consists of four integer values and therefore has a size of 16 bytes. The second is the linked list *moves*. Each element in the list contains an object of type *move* and a pointer that shows to the next item of the list, hence each element of the list has a size of 24 bytes. This list contains all valid actions that can be performed for a certain game state and therefore may utmost contain 21x8x14x14 moves, for the reasons mentioned in 4.2.2.1. However, this is an unrealistic case, as on average there are 10 available corners for each player (10 squares on the board) and definitely less than 21x8 tiles to be placed, since this number also contains duplicates as mentioned in 4.1.1. Therefore, we can estimate that the size of a list of moves will have approximately a size of $(24\text{bytes}) \times (10\text{corners}) \times (21\text{tiles}) \times (4\text{rotations}) = 20160$ bytes.

4.2.2.3 Potential Parallelism

We mentioned that the most time consuming process of the Minimax algorithm is checking whether a tile fits in a specific position of the board or not. We also explained that this procedure requires to check a 5x5 area on the board, hence 25 iterations are needed.

However, each square of the 5x5 area on the board can be evaluated independently from the rest, since we do not need a previous value to determine its state. Therefore, we can determine a valid position in a single clock cycle, thus reducing significantly the time complexity of the *check_move function*. Furthermore, computing the number of corners for each player can be also parallelized and determined in a single clock cycle, to reduce the 14x14 sequential checks. However, this would imply a slow clock and instead of a single clock cycle, perhaps 6 or 7 would be recommended.

4.2.2.4 Potential Bottlenecks

Finding a move that is valid seems to be a potential bottleneck for a hardware implementation, since it arbitrarily delays the execution of the algorithm. More specifically, checking whether a tile fits in a square of the board requires a single clock cycle. However, we might evaluate several board positions before finding a valid move and therefore this procedure needs an arbitrary number of clock cycles. Note, that in an extreme case where 10 tiles are left and none of them fits in the board, we will need $(10\text{tiles}) \times (8\text{rotations}) \times 14 \times 14 = 15680$ clock cycles to realize it. Also, Minimax is a recursive algorithm and this fact inevitably limits the amount of parallelism that can occur.

4.2.3 Hardware Implementation

In the hardware implementation Minimax was performed in the Minimax Core Module, shown in Figure 4.1. The Minimax Core Module comprises of multiple subsystems. The first stage of the algorithm is implemented in the MetaData Module. This module generates new tiles when requested, after the board is updated with an opponent move. In the beginning, a first possible move is produced according to the first available tile that is found in the tile availability vector. Next, the control, shown in Figure 4.2, activates the Move Module in order to find for a given tile, the first valid move on the board. This move indicates that a new game state should be defined and pushed into the stack for future use. A game state will contain all essential information that not only describe the state but also provide information about the sequence of the moves that were made. Therefore a game state consists of the previous state of the board before placing the new move, the tile and rotation that are used in the new move, the coordinates of the board where the placement will occur and finally two values used for the alpha-beta pruning.

4. IMPLEMENTATION

Simultaneously the board is updated according to the new game state. This process continues iteratively until a terminal game state is met. Then the Evaluation Module calculates the corresponding evaluation score. The main factors that are taken under consideration for the calculation of the position score are: i) the incremental number of corners of the player after placing the tile, ii) the reductive number of corners of the opponent after placing the tile and iii) the new score after placing tile.

Then using the stack of game states, the respective alpha or beta value or none is updated depending on whether we are in a Max or Min node. Whenever all the outputs of a game state have been considered the corresponding state is popped from the stack and the evaluation score, as well as the state are propagated to the Best Module. This stage will be repeated until the current game state contains at least one move in the move set and the pruned condition of the Minimax algorithm is not met. Additionally, in case the state popped from the stack is the root of the search tree, the move that generated the current sub-tree will be kept along with the evaluation score. As a result, after the search process has been completed the game state at the root of the search tree will contain the best move found and will be returned as a response to the opponent. In order to calculate all possible outputs, both players need to be considered. Whenever it is the opponent's turn to play, the board is flipped and the respective tile availability vector is taken into consideration.

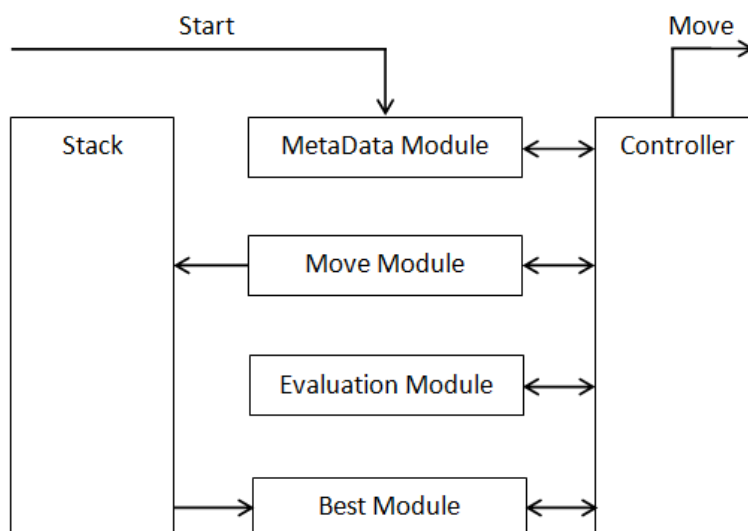


Figure 4.1: Minimax core module.

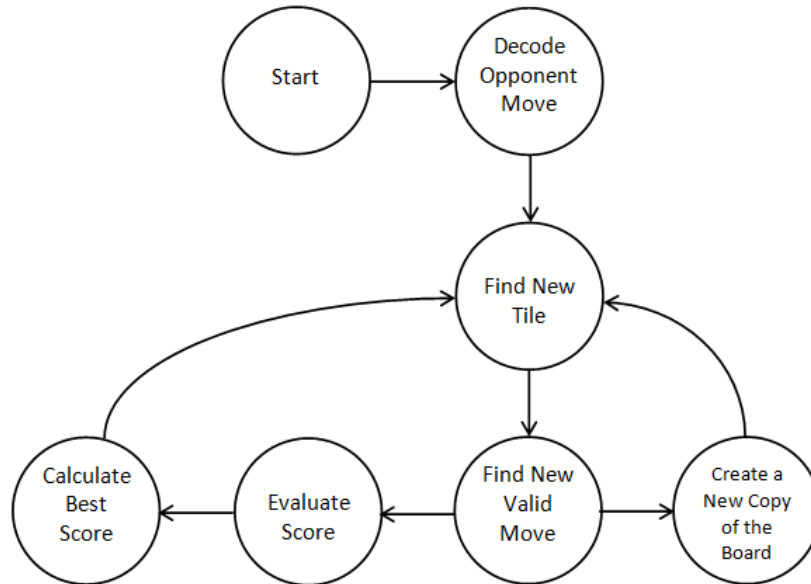


Figure 4.2: Minimax controller's FSM

4.3 MCTS player

In this section we describe our basic MCTS-based player that uses the UCB1 selection policy (UCT algorithm). We present the software implementation and discuss possible opportunities for hardware implementation.

4.3.1 Software Implementation

The creation of the MCTS player was based on algorithm 2. First, we present the structures that were used in our implementation and then some basic elements of the MCTS algorithm, specifically for the Blokus Duo game.

4.3.1.1 MCTS structures

For the implementation of the MCTS-based player we used five structures. More specifically:

4. IMPLEMENTATION

/ Keeps all information necessary to describe a move */*

```
typedef struct {  
    int x;           //coordinate x of the board  
    int y;           //coordinate y of the board  
    int piece;       //number of tile (0-20)  
    int rotate;      //number of rotation (0-7)  
} move;
```

/ Keeps all information necessary to describe the statistics of a move i.e. how many times it participated in a playout and its overall value */*

```
struct move_stats {  
    int playouts;    //number of playouts in which a node has participated  
    float value;     //the current overall value of a node  
};
```

/ Keeps all information necessary for a node in the UCT tree. We need to maintain information about the node's statistics, its position in the tree in relation to the other nodes, the actual move that the node represents, whether has been expanded at least once and its depth in the tree */*

```
struct tree_node {  
    struct move_stats data;    //statistics of the tree node  
    struct tree_node *parent, *sibling, *children;    //position of the node in the tree  
    move coord;    //the move that is represented by the tree node  
    bool is_expanded;    //shows whether the node has been expanded once or not  
    unsigned short depth;    //the node's depth in the tree  
};
```

/ Keeps all information necessary for the UCT tree used in the MCTS algorithm. We need to remember the board and availability register that are updated according to the selected path, as well as the root of the tree that maintains the rest of the tree structure*/*

```
struct tree {  
    int availability_reg[2][21];    //the UCT tree availability register
```

```

int gameboard[16][16];    //the UCT tree game board
struct tree_node *root;  //the root of the UCT tree that keeps the tree structure
int max_depth;           //the max depth to which the UCT tree has reached
};

```

```

/* The selected path is created during the selection phase and it inserts the nodes with
the highest UCB1 value. All moves inserted in this list are considered as executed by the
UCT tree. */

```

```

struct selected_path {
    int player;    //player that performed this move of the selected path
    int turn;      //the UCT tree game board
    struct tree_node* current;    //node of the game tree inserted in the selected path
    struct selected_path *next;   //next node of the selected path
};

```

4.3.1.2 MCTS Algorithm

In this section we present the basic functions of our MCTS approach, along with a brief description for each and then we describe the execution flow of the algorithm.

Basic Functions:

- **void tree_init(int player, int turn):** This is the first function called during the MCTS algorithm. It creates the UCT tree based on the actual game status. More specifically, the UCT *gameboard* and *availability_reg* are initialized with the current values of board and available array of the actual game. The root of the tree is also determined along with its children. More specifically, given the current actual game state, all possible moves that can be performed in the next turn are found and inserted in a list. The moves in this list are the root node's children. Apart from its children, the root node does not contain any other useful information. Once the *tree_init()* function has been executed the global UCT tree has been created and the first level of the tree contains all valid moves that can be performed in the next round.

4. IMPLEMENTATION

- **struct selected_path *play_in_tree(int player, int turn)**: This function performs the selection and the expansion phase of the MCTS algorithm 2.7.3.1 and 2.7.3.2 respectively. Therefore it returns a selected path that begins from the root of the UCT tree and reaches a leaf node and expands a leaf node of the UCT tree. In order to determine the selected path, *play_in_tree()* function traverses the UCT tree in a certain manner. More specifically, the starting point of the traversal is the root which is the first node inserted in the selected path. The next node inserted is the child of the root that has the highest UCB1 value. In the same manner by considering only children of the nodes already in the selected path, each child with the highest UCB1 value is added to the selected path until we have reached a leaf node. As for the expansion phase, if we have visited at least once all of a node's children, then the child node with the best UCB1 value is expanded. In any other case, we should first evaluate once all of the children nodes before expanding either one of them. Eventually the number of the nodes in the selected path will equal the depth of the UCT tree.
- **struct tree_node *select_child(struct tree_node *node, int player)**: This function receives as input a father node and returns the child node with the highest UCB1 value. However, this requires that all children nodes have been visited at least once. If this does not apply, then the first unvisited child node found is returned to be evaluated. In general, this function is used to add nodes to the *selected_path*.
- **float calculate_UCB1(struct move_stats data, int father_playouts)**: This function has as inputs the move statistics of a node and its parent's number of playouts. According to these values it calculates the UCB1 value 5 and returns this value. We decided to set the C constant value equal to 0.7 after experimental tuning.
- **int expand_node_array(int player, int turn, int gameboard[16][16], int availability_reg[2][21])**: Given a specific game state, the *expand_node_array()* function returns the number of all valid moves that can be performed in the next turn. The inputs of the function constitute and describe the game state i.e. whose turn it is, in which turn is the game, what is the board's and the availability

register's state. Apart from the number of valid moves, this function also finds which are these moves and inserts each one of them in a global array, the *available_moves* array that contains elements of type *move*.

- **int play_out_tree(struct selected_path* selected):** This function performs the simulation phase of the MCTS algorithm and its execution is quite simple. As long as the game is not over (we have not met a terminal condition) play random moves i.e. simulate a game scenario by choosing randomly a move in each turn. Once the simulated game is over the function returns 1 if our player won or -1 if our player lost.
- **move random_simulation(int player, int turn, int (*tboard)[16], int (*tavailable)[21])** This function simply returns a random valid move for a given game state.
- **void update_uctvalues(struct selected_path* selected,int value)** This function performs the backpropagation phase of the MCTS algorithm [2.7.3.5](#). More specifically, depending on the outcome of the simulation this function updates the statistics of all UCT tree nodes that participated in the specific payout.

The execution of the MCTS algorithm in our approach begins with the *tree_init()* function when the UCT tree is initialized appropriately. Then, the *play_in_tree()* function executes the selection phase and returns the selected path of the UCT tree that includes the children that were found to have the highest UCB1 value. This function also performs the expansion phase of the MCTS algorithm and either expands the UCT tree if all children of a node have been visited, or evaluates a child node that has not been yet searched. Once the *play_in_tree()* has returned, the *play_out_tree()* function is executed and starting from the expanded or the unvisited node it performs a game simulation by executing random moves. When a terminal game condition is met it returns the outcome of the simulation declaring either a win (returns value 1) or a loss (returns value -1). Finally, the *update_uctvalues()* function is executed, in order to update the number of playouts and the value of all the UCT tree nodes that participated in the playout. The above procedure is performed repeatedly until either we have reached a specific time limit, or there are no memory resources left due to the size of the UCT tree.

4. IMPLEMENTATION

4.3.2 Modeling for Hardware Implementation

In this section we present critical points of the software code that indicate hardware opportunities. In order to do so, we performed profiling of the code using the Linux GNU GCC Profiling Tool (gprof), we computed the size of the structures used in the software code, we detected potential parallelism, as well as potential bottlenecks.

4.3.2.1 Code Profiling

The results of the code profiling showed that MCTS and Minimax share common time consuming functions. Once more, *check_move()* requires the most execution time, followed by *expand_node_array()* that is similar to *returnAllValidMoves()* of the Minimax algorithm. The complexity of these functions has been analyzed in 4.2.2.1. Two other procedures that appear to be time consuming are placing and removing a move. More specifically, placing a move requires updating a 5x5 area of the board and the same applies for removing a move. Therefore 25 iterations are needed and given that moves are placed and removed many times during the execution of the algorithm, these procedures occupy a significant percentage of the execution time.

4.3.2.2 Memory Requirements

In order to determine the memory requirements of the MCTS algorithm, we present the size of the five structures that are involved in the specific algorithm that were mentioned in 4.3.1.1. Structure *move* is the same as Minimax and requires 16 bytes. MCTS also requires keeping statistics of a move, that occurs in structure *move_stats*. These statistics are the number of playouts of a node and its current value. Therefore, structure *move_stats* has a size of 8 bytes. Furthermore, we have mentioned many times structure *tree_node*, since it keeps all information necessary to describe and evaluate a node at the end of the execution of the MCTS algorithm. This structure contains objects of type *move_stats*, *move*, as well as a boolean and an unsigned short value. It also contains pointers that show towards his father node, his siblings and his children, to know the node's position in the UCT tree. Therefore, a *tree_node* object initially has a size of 56 bytes, but its size increases significantly when siblings and children are added. Another important structure of the MCTS algorithm, is the *tree*. This structure contains the current board, the availability register, the maximum depth of the tree and an object

of type *tree_node* that is the root that holds the whole UCT tree. It is clear, that this structure requires the most memory which may eventually be prohibitive. Note, that the size of the tree increases exponentially at each level for the Blokus Duo game. The fifth and final MCTS structure is the *selected_path*. This is a list that contains two integer values, a *tree_node* object and a pointer towards the next item of the list. Therefore, its size varies is determined by the size of the *tree_node* object.

4.3.2.3 Potential Parallelism

As in 4.2.2.3, we can check whether a move is valid in a single clock cycle. The same applies for placing and removing a move from the board. More specifically, these procedures need to update a 5x5 area of the board which is performed sequentially in the software implementation. In hardware this update can occur simultaneously and in a single clock cycle, since the values to be updated are independent. At this point we should mention an essential aspect of the MCTS algorithm. Given a specific time limit, MCTS performs less simulations than the Monte Carlo algorithm, which is understandable, since MCTS also requires performing actions on the UCT tree, besides executing simulations. As we will see in our experimental results, the reduced number of simulations might lead to inaccurate statistical knowledge for a move i.e. bad performance. Some approaches proposed parallelizing the MCTS procedure, in order to address this issue. More specifically, three parallelisation methods have been proposed: leaf parallelisation, root parallelisation and tree parallelisation. We will briefly describe each method and decide whether their implementation in hardware would be efficient.

- *Leaf Parallelisation*: Leaf Parallelisation is a simple idea that was proposed by Chaslot et al. in [28]. The idea of their approach is quite simple and suggests that when we reach a leaf node in the UCT tree, we do not only execute one simulation for this leaf but multiple ones, to gain more accurate statistics. The issue with this approach is that the outcomes of the different simulations will be ready in different time instances. Therefore, the execution will have to wait for the outcome of the last simulation in order to update the nodes with the new values. Experiments showed that running 4 simulated games in parallel lasted 1.15 times longer than running one simulated game and therefore they concluded that plain leaf parallelisation is an inefficient way of parallelising MCTS.

4. IMPLEMENTATION

- *Root Parallelization*: Root parallelisation was also proposed by Chaslot et al. in [28]. Now, instead of parallelising simulations of a leaf node, the whole MCTS search is parallelised. More specifically, multiple UCT tree instances are created and a thread is used per tree. The MCTS algorithm is executed independently for each tree and there is no information sharing. Once a time limit is exceeded, the best move is decided after all UCT tree clones have been merged and all scores have been added. Unlike leaf parallelisation, root parallelisation was quite effective especially because little information sharing between the trees is required and local optimals are avoided. However, root parallelisation needs memory for multiple UCT trees, that may not be available.
- *Tree Parallelisation*: Tree parallelisation was also suggested by Chaslot et al. in [28]. This final approach allows running simultaneous games on the same UCT tree and proved to be the most efficient approach among the three. Note, that each thread can modify information of a node but mutexes are required to do so effectively.

Unlike Minimax, MCTS is not a recursive algorithm and parallelisation is suggested to increase the efficiency of the algorithm. Specifically for a hardware implementation, the simulation phase can be pipelined and multiple simulations can be run concurrently. Getting more accurate results from the simulation phase will yield better solutions.

4.3.2.4 Potential Bottlenecks

Finding all valid moves is the main bottleneck of MCTS, as in 4.2.2.4. Note, that this issue can affect dramatically the simultaneous execution of different simulations since a single simulation can delay the whole procedure. Another bottleneck is the size of the MCTS tree. As the tree expands, its size increases exponentially and the memory of an FPGA is not sufficient. Therefore, we recommend implementing the UCT tree and executing all UCT tree actions in the software side and performing simultaneous simulations in the hardware side.

4.4 Monte Carlo player

4.4.1 Software Implementation

The implementation of the Monte Carlo player was based on the theory of Monte Carlo simulations presented in 2.5.1. First, we present the structures that were used and then some basic elements of the MCTS algorithm, but now specifically for the Blokus Duo game.

4.4.1.1 Monte Carlo structures

We mentioned that during the simulation phase of the MCTS algorithm, a simulation of a game scenario is executed by selecting random moves from a set of available actions. A Monte Carlo player practically performs multiple times this phase of the MCTS algorithm, sums the outcomes of each candidate move and finally selects the one with the highest score. Due to the fact that our Monte Carlo-based player is similar to the simulation phase of our MCTS-based player, the structures used in the first case are quite similar to the ones used in the latter case. More specifically:

```
    /* Stores all moves that are candidates for selection in this round of the game. */  
    struct node candidates[1000]  
  
    /* Keeps all information necessary to describe a move */  
    typedef struct {  
        int x;          //coordinate x of the board  
        int y;          //coordinate y of the board  
        int piece;     //number of tile (0-20)  
        int rotate;    //number of rotation (0-7)  
    } move;  
  
    /* Defines a node that includes the coordinations of a move and the current score of  
    that move in order to be able to identify at the end the move with the highest score*/
```

4. IMPLEMENTATION

```
struct node {
    move coord;    //the move that is represented by the tree node
    int score;     //the total score for the specific move
};

/* Keeps the sequence of moves performed during the simulation. These moves are
performed on our global board, therefore we need the whole sequence in order to undo
these moves once the simulation has terminated. */
struct tree_node {
    move current;           //the move that is represented by the tree node
    struct undo_sequence *next //sequence of moves we need to remove from the board
};
```

4.4.1.2 Monte Carlo Algorithm

In this section we present the basic functions of our Monte Carlo approach, along with a brief description for each and then we describe the execution flow of the algorithm.

Basic Functions:

- **move playMonteCarlo(int player, int turn):** This function triggers the execution of the Monte Carlo algorithm.
- **int expand_node_arrayMonte(int player, int turn, int gameboard[16][16], int availability_reg[2][21]):** Given the current state of the game, this function finds all candidate moves that can be performed next and stores them in the global array, `struct node candidates[1000]`.
- **int montecarloSimulation(move coord,int player,int turn):** This function receives as input a game state and runs iteratively until a terminal game condition is met, therefore simulating a game scenario. The move sequence is created randomly and is kept in a list. Once the simulation is terminated, the moves of the list are "unperformed" in order to get the initial condition of the game state and the outcome of the simulation is returned.

The execution of the Monte Carlo algorithm in our approach begins with the *playMonteCarlo()* function. It initially calls *expand_node_arrayMonte()* in order to find and store all candidate moves that are to be evaluated. Then, a random generator selects a move from the set of the candidate actions. The selected move, along with the current game state are given to the *montecarloSimulation()* function so that the simulation for the specific move can begin. The simulation is performed according to a random policy i.e. moves are selected randomly. Once a move is selected, it is executed and stored in a list that contains the sequence up to that time. The simulation continues until there is no move left to play. Then, we evaluate whether the outcome is a win, loss or draw and undo the moves that are stored in the sequence list to return to the initial game state. The *montecarloSimulation()* function returns the outcome of the simulation to *playMonteCarlo()*, that finds the location of the initially selected candidate move in the `struct node candidates[1000]` array and updates its score. We repeat the above procedure until a predefined time limit is exceeded. Then, the candidate move with the highest score is found and returned as the Monte Carlo solution.

4.4.2 Modeling for Hardware Implementation

Due to the fact, that the simulation phase of MCTS performs Monte Carlo simulations, an implementation of these two algorithms in hardware would have many common points. In fact, differences only exist in actions performed on the UCT tree of the MCTS algorithm. Therefore, in terms of memory Monte Carlo only requires maintaining structures that keep all information necessary to describe a move and a list that contains a sequence of actions performed during the simulation, in case we will need to remove these actions. As for potential parallelism and potential bottlenecks, they are the same with those provoked by the simulation phase of MCTS. Many simulations can be executed in parallel to get accurate results, but finding a valid move may delay by arbitrary clock cycles the procedure. We do not further analyze modeling Monte Carlo for hardware implementation, in order to not repeat those that were mentioned in section 4.3.2.

4. IMPLEMENTATION

Chapter 5

Optimizations on MCTS

We have already mentioned that the performance of our MCTS algorithm can be improved with the use of several techniques. In this chapter we study the adjustment of the UCB1 selection policy for the Blokus Duo game and we test the performance of another selection policy too. Furthermore, enhancements in the selection phase are strongly recommended to create a competitive MCTS-based player. Therefore, we implemented and examined the First Play Urgency and the Progressive Bias enhancement techniques. A better simulation method based on an evaluation function is tested too. The last optimization we tried to integrate into our enhanced player gave a score bonus in the backpropagation method, but the performance was not increased.

5.1 Selection Policies

The selection policies used by our MCTS-based players use the outcomes of simulated games up to the current point to find the best move. The selection policies accept values in the interval of $[0; 1]$. However, in Blokus Duo, draws may also occur and therefore the game values may be -1 for a lost game, 0 for a draw and 1 for a won game.

5.1.1 UCB1 Adjustment

The UCB1 policy, mentioned in 5 includes the constant $C_p > 0$ that has an arbitrary value. The constant C_p controls the exploration-exploitation trade-off of the search by giving more or less weight to the upper confidence bound. It is strongly recommended to

5. OPTIMIZATIONS ON MCTS

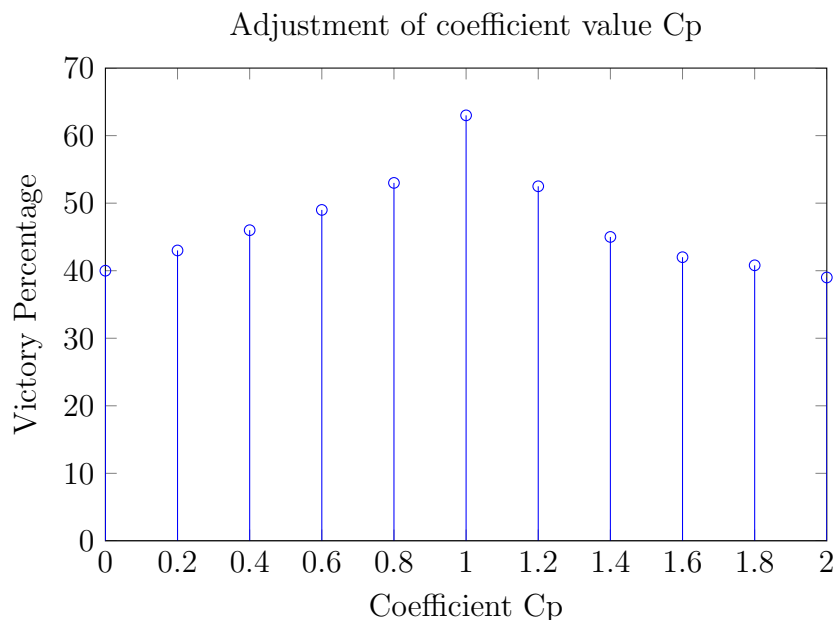


Figure 5.1: Winning percentage for different C_p values

adjust this value since it highly depends on the domain of the problem, the computational resources and the MCTS implementation. Therefore, we tested different C_p values to find the most appropriate one for the Blokus Duo game. We conducted several experiments by setting C_p equal to 0, 0.2, 0.4, 0.6, 0.8, 1, 1.2, 1.4, 1.6, 1.8, 2 and executed the algorithm for 5 seconds. The results are shown in table 5.1 where the first value of each cell denotes the percentage of those won of all games and the second one shows the mean value of all the scores. The most appropriate value appeared to be 1 that balances perfectly the exploitation and exploration factors.

5.1.2 UCB1-Tuned

Gelly and Wang stated that UCB1-Tuned policy yielded better results than UCB1 although they could not provide theoretical guarantees for the regret of UCB1-Tuned, as they did for UCB1. Furthermore, the fact that the UCB1-Tuned formula does not require any values to be adjusted, makes it easy to implement and test. We evaluated the UCB1-Tuned policy's performance against an MCTS player that uses the UCB1 policy. The algorithm was executed for 1, 2 and 5 seconds and for each experimental result 40

Selection Policies	1 sec	2 sec	5 sec
UCB1Tuned	70%	77.5%	82.5%
	26.75	26.55	25.64
UCB1	27.5%	17.5%	17.5%
	31.95	32.05	32.45

Figure 5.2: Comparison between UCB1 and UCB1Tuned policies. The first value in each cell denotes the player’s percentage of won games and the second the mean value of all the scores of the player.

games were played. Of the two values in each cell the first denotes the percentage of won games under all games and the second the mean value of all the scores. The results are shown in 5.2. We observe that UCB1-Tuned policy outperforms the UCB1 policy, even when a small number of simulations is executed.

5.2 Selection Phase

5.2.1 First Play Urgency (FPU)

The First Play Urgency technique is used by many professional MCTS-based players because it allows exploiting nodes from an early stage of the MCTS algorithm. We created players with different FPU values to evaluate which one improves the performance. These values are:

- **10000**: A high FPU value encourages exploration and all nodes will be visited at least once before letting expansion to occur to a sibling node. Setting FPU equal to 10000 is equivalent to the default selection policy of the MCTS algorithm.
- **0.1**: A low FPU value encourages exploitation. However, if the nodes are explored in a pessimistic manner and a small number of simulations is executed we may get stuck to a local optimum.
- **Heuristic**: We considered a heuristic approach for the FPU values. More specifically, units of a tile can guide the search towards the most urgent moves, since big tiles are encouraged to be placed first. We know that the UCB1 policy initially

5. OPTIMIZATIONS ON MCTS

returns values in the interval $[-0.5;0.5]$ and therefore we multiplied the number of units of a tile with the number 0.1 in order to have comparable values. For example, if the FPU values had a higher value, then when comparing the UCB1 value of the visited nodes with the FPU values of the unvisited nodes, we would select those with the highest value i.e. the unvisited ones, thus encouraging exploration.

We evaluated the three different FPU values and the results are shown in 5.3. The algorithm was executed for 1, 2 and 5 seconds and for each experimental result 40 games were played. We observed the following:

1. The comparison between the Heuristic FPU value and the 10000 FPU value showed that the first outperforms the latter, especially when the execution time is small. Given that the heuristic FPU value guides the search, it allows exploring promising moves first right from the very beginning of the execution. However, as time increases more exploration occurs, more statistics are gathered and we notice that the performance of the Heuristic FPU values decreases, but not significantly.
2. The comparison between the Heuristic FPU value and the 0.1 FPU value also showed that the heuristic approach outperformed exploitation. As mentioned above, when few information about a move is accumulated it is easy for the algorithm to get stuck to a local optimum.
3. The comparison of the performance of the 0.1 and 10000 values was interesting. In the first rounds of a Blokus Duo game, there are many first good moves to be placed (there are 12×5 unit tiles to be placed and only one starting point for each player). Therefore, if the algorithm detects one or more of these moves and lets exploitation occur, the results will be better than exploring all moves. However, as the execution time increases more moves are explored. The results prove that for a small execution time encouraging exploitation is better than exploration for the Blokus Duo game, but as time increases exploration could yield more accurate results.

Player1 vs Player 2	1 sec	2 sec	5 sec
Heuristic vs 10000	92.5% - 0.7% 17.32 - 38.37	77.5% - 15% 15.92 - 38.25	72.5% - 27.5% 16.42 - 35.68
10000 vs Heuristic	10% - 87.5% 36.62 - 20.15	10% - 85% 32.67 - 19.4	20% - 77.5% 30.76 - 19.85
Heuristic vs 0.1	87.5% - 12.5% 17.47 - 36.65	72.5% - 25% 19.85 - 35.05	72.5% - 27.5% 21.36 - 31.67
0.1 vs Heuristic	17.5% - 82.5% 31.15 - 22.15	30% - 65% 30.47 - 23.02	27.5% - 70% 32.24 - 20.45
10000 vs 0.1	32.5% - 55% 26.1 - 25.6	30% - 48% 24.9 - 24.34	37.5% - 52.5% 22.34 - 24.96
0.1 vs 10000	57.5% - 37.5% 25.07 - 28.52	52.5% - 40% 25.47 - 27.72	47.5% - 50% 25.34 - 24.52

Figure 5.3: Comparison between Heuristic, 10000 and 0.1 FPU values. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

5.2.2 Progressive Bias

Progressive Bias is used to lead the search towards promising moves when few statistics have been gathered. In order to do so, it requires precise and effective domain knowledge that may not be always available. Also, computing the heuristic values needed in the Progressive Bias approach should occupy a negligible percentage of the MCTS execution time. We decided to test two heuristic approaches which are the following:

- **Progressive Bias Function:** In this approach we used a function to evaluate nodes located in the first level of the tree and a simple value for the rest. This function was formed according to the round of the game since different stages imply a different approach. More specifically:
 - **Turn ≤ 16 :** In the first rounds of the game we need our player to play defensive and create corners to be flexible on the board. Also, the biggest tiles should be placed first in the beginning and in the middle of the game, so we concluded with the following function: $0.5 * a + 1 * b + 0.25 * c$, where

5. OPTIMIZATIONS ON MCTS

a denotes the number of units of the tile, b is the increasing number of our player's corners, c shows the decreasing number of the opponent's corners.

- **Turn > 16:** Towards the end of the game we need to get rid of big tiles, while preventing the opponent to do the same thing. Therefore, we concluded with the following function: $1 * a + 0.25 * b + 1 * c$, where a denotes the number of units of the tile, b is the increasing number of our player's corners, c shows the decreasing number of the opponent's corners.

- **Progressive Bias Units:** In this approach we simply compute the units of the tile to be placed and set this number as the Progressive Bias value. Therefore, we do not lose any time in heuristics while guiding the search towards big tiles, that is essential in the Blokus Duo game.

We evaluated the two different Progressive Bias approaches against a normal UCT player and the results are shown in tables 5.4 and 5.5. The algorithm was executed for 1, 2 and 5 seconds and for each experimental result 40 games were played. Of the two values in each cell the first denotes the percentage of won games under all games and the second the mean value of all the scores. We observed the following:

1. Although computing the values of the Progressive Bias function occupied a noticeable percentage of the MCTS execution time, the enhanced player performed slightly better than the normal one. This indicates that eventhough the heuristic used is probably not the most efficient one, it still increases the performance of the player.
2. Simply considering the units of a tile is not sufficient since there is no information about the state of the game board.

5.3 Simulation Phase

The default MCTS player executes simulations that select and play moves randomly. This approach explores many actions and evaluates a variety of game scenarios, but does not focus on the ones that will yield the best results. Therefore, we decided to apply an evaluation function to select moves to bias the search towards moves that seem more promising.

Progressive Bias	1 sec	2 sec	5 sec
Progressive Bias Function	50% 28.92	52.5% 30.2	55% 28.67
UCT	50% 28.45	37.5% 26.92	40% 30.05

Figure 5.4: Comparison between the UCT algorithm enhanced with the Progressive Bias technique that uses the evaluation function and the normal UCT algorithm. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

Progressive Bias	1 sec	2 sec	5 sec
Progressive Bias Units	35% 33.52	37.5% 33.95	40% 30.75
UCT	57.5% 26.75	60% 29.42	47.5% 29.57

Figure 5.5: Comparison between the UCT algorithm enhanced with the Progressive Bias technique that counts the units of the tile and the normal UCT algorithm. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

5. OPTIMIZATIONS ON MCTS

Evaluation Function	1 sec	2 sec	5 sec
Evaluation Function Heuristic	95%	97.5%	92.5%
	22.15	19.72	22.42
UCT	0.25%	0.25%	0.5%
	51.35	53.35	54.4

Figure 5.6: Comparison between the UCT algorithm enhanced with the Evaluation Function technique that uses the a heuristic function and the normal UCT algorithm. The first value in each cell denotes the player’s percentage of won games and the second the mean value of all the scores of the player.

5.3.1 Evaluation Function

Besides the Progressive Bias technique, we can also use domain knowledge for the simulations executed in the simulation phase of the algorithm to avoid unrealistic and inefficient game scenarios. More specifically, during the simulation we do not select moves randomly, but play moves that maximize some value. These values are the same as in Progressive Bias 5.2.2 since we decided to use simple heuristics that would not occupy a significant amount of the execution time. We evaluated the two different simulation approaches against a normal UCT player and the results are shown in tables 5.6 and 5.7. The algorithm was executed for 1, 2 and 5 seconds and for each experimental result 40 games were played. Of the two values in each cell the first denotes the percentage of won games under all games and the second the mean value of all the scores. We observed the following:

1. Selecting moves according to heuristics is definitely better than allowing simulations to choose random ones from the set of available actions. Due to the fact that the number of executed simulations is usually limited, it is important to know that they use some bias to consider promising moves and perform rational games. It is clear, that using heuristics during the simulation policy increases significantly the performance of the MCTS-based player.
2. In order to get accurate results from the simulations, it is essential to execute as many as possible. Therefore, little time should be spend to compute heuristics and more should be devoted to gather statistics.

Evaluation Function	1 sec	2 sec	5 sec
Evaluation Function Units	97.5%	95%	95%
	21.7	20.15	18.82
UCT	0.25%	0.5%	0.25%
	53.17	54.62	52.07

Figure 5.7: Comparison between the UCT algorithm enhanced with the Evaluation Function technique that counts the units of the tile and the normal UCT algorithm. The first value in each cell denotes the player’s percentage of won games and the second the mean value of all the scores of the player.

5.3.2 Score Bonus

The result of a Blokus Duo game can indicate a strong or weak win, as well as a strong or weak loss. In order to evaluate the significance of this piece of information we decided to implement the Score Bonus technique using two families of scores:

- **First family:**
- **Second family:**

We evaluated the two different score bonus approaches against a normal UCT player and the results are shown in tables 5.8 and 5.9. The algorithm was executed for 1, 2 and 5 seconds and for each experimental result 40 games were played. Of the two values in each cell the first denotes the percentage of won games under all games and the second the mean value of all the scores. We observed that both approaches created players inferior to the normal UCT player. This is probably due to the fact that eventhough the eventual score does indicate the power of a player, it is a result of a sequence of moves and not of the first move played. Therefore, it tells us little information about whether the first move alone was a good one.

5.3.3 Best Combination of Optimizations

In the sections above, the optimized parameters are only investigated for their own. The combination of two or more optimizations can lead to even better results, although this is not necessary since the results of a technique may negatively affect the performance

5. OPTIMIZATIONS ON MCTS

Score Bonus	1 sec	2 sec	5 sec
Score Bonus First Family	42.5%	25%	27.5%
	25.1	26.77	28.02
UCT	55%	65%	62.5%
	32.35	28.52	31.92

Figure 5.8: Comparison between the UCT algorithm enhanced with the Score Bonus technique that uses the first family of scores and the normal UCT algorithm. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

Score Bonus	1 sec	2 sec	5 sec
Score Bonus Second Family	42.5%	37.5%	35%
	25.02	26.12	26.22
UCT	57.5%	55%	52.5%
	28.02	31.12	29.3

Figure 5.9: Comparison between the UCT algorithm enhanced with the Score Bonus technique that uses the second family of scores and the normal UCT algorithm. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

of another technique. In case, we were interested in creating a highly optimized player different combinations should have been searched by many trials in different experiments. However, since this is not the scope of this thesis we decided to apply the UCB1-Tuned selection policy that has been successful for many players, the FPU technique that uses the units of the tiles as FPU values, the Progressive Bias heuristic method that computes the evaluation function and the simulation policy that only considers the units of the tiles, in order to not occupy time from the execution of the MCTS algorithm.

5. OPTIMIZATIONS ON MCTS

Chapter 6

Comparison of players

This chapter presents the results of the comparison studies of the implemented Blokus Duo players.

6.1 Comparison of UCT with Monte Carlo

This section compares the UCT-based player with the Monte Carlo-based player. In table 6.1 we see the results when the UCT player is player 1 and, in table 6.2 we see the results when the Monte Carlo player is player 1. We evaluate both cases to assure that no player has an advantage that may depend on his turn in the game. In order to determine the overall performance of each player, nine experiments with different time limitations were conducted and for each, 40 games were played. Of the two values listed in each cell the first denotes the percentage of won games under all games and the second the mean value of all the scores. We observed the following:

1. When the time given to the two players is the same, but small i.e. 1 second, the Monte Carlo-based player beats the UCT-based player. This is understandable since Monte Carlo performs more simulations than UCT in the same time period and this difference greatly affects the player's performance. UCT has not had time to gather enough data to compute accurate statistics leading to the selection of bad moves. However, when both have 5 or 10 seconds to find a move, the performance of the UCT-based player increases since the execution of more simulations, combined with the acquired statistical knowledge allows finding better solutions. Also, the

6. COMPARISON OF PLAYERS

<div style="display: inline-block; transform: rotate(-45deg); transform-origin: center;"> UCT \ MC </div>	1 sec	5 sec	10 sec
1 sec	35% - 62.5% 33.02 - 30.62	40% - 60% 32.8 - 29.2	22.5% - 77.5% 36.07 - 27.75
5 sec	57.5% - 37.5% 32.15 - 33.32	42.5% - 47.5% 30.07 - 29.55	42.5% - 52.5% 31.35 - 29.24
10 sec	62.5% - 32.5% 30.27 - 32.3	65% - 30% 30.4 - 30.72	45% - 47.5% 28.01 - 30.63

Figure 6.1: Comparison between Monte Carlo Tree Search and Monte Carlo players when UCT plays first. Blue indicates UCT and gray indicates MC. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

UCT player is able to play almost as good as the Monte Carlo player and there is small statistical difference in the values of won games.

2. It is clear that if we give more execution time to any one, of the two players, that player has higher winning percentage and a smaller average of total score points. But although both perform better in such cases, the Monte Carlo-based player yields better statistical results. Note, that in the first table 6.1, when 10 seconds are given to the UCT-based player and 1 second is given to the Monte Carlo-based player, the UCT player has 62.5% winning percentage. However, if the Monte Carlo-based player is given 10 seconds the UCT-based player is given 1 second, Monte Carlo has 77.5% winning percentage. One possible interpretation could be that, the UCT player requires more than 10 seconds to be able to converge to promising moves, while, the Monte Carlo player has executed enough simulations to find sufficiently good moves.
3. It seems that the one who plays first, has a small advantage.

6.2 Comparison of UCT with Minimax

MC \ UCT	1 sec	5 sec	10 sec
1 sec	67.5% - 30% 28.5 - 34.92	32.5% - 57.5% 30.2 - 29.42	32.5% - 60% 31.45 - 28.67
5 sec	67.5% - 32.5% 26.77 - 31.9	47.5% - 45% 31.40 - 35.52	45% - 55% 31.07 - 31.4
10 sec	80% - 20% 27.35 - 38.42	65% - 25% 28.02 - 31.27	60% - 37.5% 28.25 - 30.13

Figure 6.2: Comparison between Monte Carlo Tree Search and Monte Carlo players when MC plays first. Blue indicates UCT and gray indicates MC. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

6.2 Comparison of UCT with Minimax

This section compares the UCT-based player with the Minimax-based player. In table 6.3 we see the results when the UCT player is player 1 and, in table 6.4 we see the results when the Minimax player is player 1. We evaluate both cases to be sure that no player has an advantage that may depend on his turn in the game. Note, that the sequence according to which Minimax investigates moves, determines its overall performance. If moves are searched from the best to the worst one, the alpha-beta pruning will let the algorithm evaluate only useful moves and a good solution will be found early. In the opposite case the Minimax algorithm will need to search almost the whole tree to find a good move. Therefore, we let the Minimax-based player to first examine moves that include big tiles, as in any other case we would expect him to perform poorly. However, since Minimax benefits from this order of search and to be fair, we also let UCT search moves that include big tiles first. In order to determine the overall performance of each player, nine experiments with different time limitations were conducted and 40 games were played for each result. Of the two values listed in each cell the first denotes the percentage of won games under all games and the second the mean value of all the scores. We observed the following:

1. Although we let Minimax search moves in an optimistic manner, it is not enough for

6. COMPARISON OF PLAYERS

him to play at least competitive. In all cases, the UCT-based player presents better results, even when he has 1 second of thinking time and Minimax has 10 seconds of thinking time. In order to interpret the supremacy of the UCT algorithm we need to recall the complexity of the Blokus Duo game and how Minimax works. More specifically, a Blokus Duo game instance has an average of 10 available corners for each player and about 40 tiles to consider. Therefore, 400 available moves need to be evaluated for a single move. Given the complexity of the tree and that Minimax should find a solution in a reasonable period of time, he is not able to search beyond the fourth level of the Minimax tree and therefore cannot play well. The existence of such large trees led to the creation of the UCT algorithm in the first place.

2. Note, that when Minimax is given 5 or 10 seconds to find a good answer and UCT is given only 1 second, the winning percentage, as well as the average score of the first are improved significantly compared to when he is given only 1 second. This is due to the fact that 1 second allows the algorithm to search until depth 3, which is clearly not sufficient. However, when 5 and 10 seconds are available to the Minimax player, the algorithm can reach depth 4 and a better answer is achieved.
3. The improvement in the performance of the Minimax algorithm, when he is given 5 or 10 seconds and UCT is given only 1 second is small, since in both cases Minimax has still only reached depth 4 and much more time is required for him to move to depth 5.

6.3 Comparison of Monte Carlo with Minimax

This section compares the Monte Carlo-based player with the Minimax-based player. In table 6.5 we see the results, when the Monte Carlo player is player 1 and in table 6.6 we see the results when the Minimax player is player 1. We evaluate both cases to be sure that no player has an advantage that may depend on his turn in the game. For the same reasons mentioned in the comparison of UCT and Minimax, the Minimax-based player first examines moves that include big tiles. In order to determine the overall performance of each player nine experiments with different time limitations were conducted and for each 40 games were played. Of the two values listed in each cell the first denotes the

6.3 Comparison of Monte Carlo with Minimax

Minimax \ UCT	1 sec	5 sec	10 sec
UCT	90% - 10% 25.32 - 33.75	77.5% - 17.5% 25.75 - 30.9	67.5% - 30% 25.92 - 29.12
1 sec	95% - 0.5% 23.75 - 35.42	92.5% - 0.25% 26.1 - 35.27	92.5% - 0.5% 26.83 - 33.67
5 sec	97.5% - 0.25% 23.72 - 40.22	95% - 0.25% 23.16 - 38.86	90% - 0.7% 25.37 - 35.37
10 sec			

Figure 6.3: Comparison between Monte Carlo Tree Search and Minimax players when UCT plays first. Blue indicates UCT and gray indicates Minimax. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

UCT \ Minimax	1 sec	5 sec	10 sec
Minimax	12.5% - 85% 31.45 - 22.21	0.75% - 87.5% 33.6 - 20.24	0.75% - 92.5% 34.86 - 21.35
1 sec	20% - 77.5% 30.56 - 22.76	12.5% - 82.5% 32.56 - 21.88	12.5% - 85% 32.75 - 20.48
5 sec	37.5% - 62.5% 28.64 - 24.86	25% - 72.5% 29.18 - 24.65	20% - 77.5% 29.69 - 22.26
10 sec			

Figure 6.4: Comparison between Monte Carlo Tree Search and Minimax players when Minimax plays first. Blue indicates UCT and gray indicates Minimax. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

6. COMPARISON OF PLAYERS

<div style="display: inline-block; transform: rotate(-45deg);"> Minimax MC </div>	1 sec	5 sec	10 sec
1 sec	77.5% - 20% 25.32 - 33.75	80% - 20% 28.1 - 31.2	72.5% - 27.5% 30.02 - 33.72
5 sec	85% - 10% 27.67 - 34.27	87.5% - 0.25% 27.87 - 33.5	70% - 20% 29.25 - 32.22
10 sec	95% - 0.5% 26.22 - 35.62	92.5% - 0.25% 29.15 - 34.2	90% - 0.75% 27.35 - 34.22

Figure 6.5: Comparison between Monte Carlo and Minimax players when MC plays first. Blue indicates Monte Carlo and gray indicates Minimax. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

percentage of won games under all games and the second the mean value of all the scores. Due to the fact that the observations are similar to the ones in section 6.2 we will mention them briefly. It is clear that Monte Carlo outperforms Minimax in all cases for the same reasons that UCT was better than Minimax. An interesting fact, is that playing with Minimax, Monte Carlo presents better statistics than those presented by UCT. This is attributed to the fact that UCT executes less simulations for the same time period.

6.4 Comparison of Enhanced MCTS with UCT

This section compares the Enhanced MCTS-based player with the normal UCT-based player. In table 6.7 we see the results, when the Enhanced MCTS player is player 1 and in table 6.8 we see the results when the normal UCT player is player 1. We evaluate both cases to be sure that no player has an advantage that may depend on his turn in the game. In order to determine the overall performance of each player nine experiments with different time limitations were conducted and for each 40 games were played. Of the two values listed in each cell the first denotes the percentage of won games under all games and the second the mean value of all the scores. We observed the following:

1. Note, that the Enhanced MCTS-based player presents a much higher winning percentage and a much smaller mean score than the UCT-based player. Such a result

6.5 Comparison of Enhanced MCTS with Monte Carlo

<div style="display: inline-block; transform: rotate(-45deg); transform-origin: center;"> Minimax \ MC </div>	1 sec	5 sec	10 sec
1 sec	15% - 72.5% 32.67 - 29.37	0.5% - 95% 33.4 - 27.12	0.25% - 95% 35.17 - 27.55
5 sec	27.5% - 67.5% 32.27 - 30.45	0.25% - 92.5% 34.67 - 28.22	0.25% - 95% 34.07 - 29.12
10 sec	35% - 65% 32.6 - 31.22	10% - 90% 34.37 - 29.92	0.25% - 97.5% 35.27 - 27.85

Figure 6.6: Comparison between Monte Carlo and Minimax players when Minimax plays first. Blue indicates Monte Carlo and gray indicates Minimax. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

was expected, since the enhanced player uses a more efficient selection policy and guides all explorations towards promising moves. On the other hand, UCT uses the standard UCB1 policy and selects random moves. Therefore, the performance of the UCT-based player depends on whether some of the random simulations will yield accurate results. This is clear in cases when UCT has 10 seconds to return an answer, since then more simulations are executed and his winning percentage is increased. However, he still remains inferior to the Enhanced MCTS-based player.

- Again, we see that the player 1 has a slight advantage compared to player 2. Note, that the statistics of the Enhanced MCTS-based player are slightly decreased when he plays second.

6.5 Comparison of Enhanced MCTS with Monte Carlo

This section compares the Enhanced MCTS-based player with the Monte Carlo player. In table 6.9 we see the results, when the Enhanced MCTS player is player 1 and in table 6.10 we see the results when the Monte Carlo player is player 1. We evaluate both cases to be sure that no player has an advantage that may depend on his turn in the game. In order to determine the overall performance of each player nine experiments

6. COMPARISON OF PLAYERS

UCT \ Enhanced MCTS	1 sec	5 sec	10 sec
1 sec	85% - 15% 32.37 - 40.97	82.5% - 15% 32.1 - 42.55	85% - 15% 30.8 - 45.52
5 sec	80% - 17.5% 30.87 - 41.17	82.5% - 12.5% 32.15 - 42.37	77.5% - 22.5% 32.4 - 42.67
10 sec	92.5% - 0.5% 30.7 - 43.55	92.5% - 0.75% 29.12 - 45.12	77.5% - 20% 29.42 - 56.25

Figure 6.7: Comparison between Enhanced MCTS and UCT players when Enhanced MCTS plays first. Blue indicates Enhanced MCTS and gray indicates UCT. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

UCT \ Enhanced MCTS	1 sec	5 sec	10 sec
1 sec	30% - 70% 32.97 - 30.27	15% - 80% 40.55 - 27.77	0.75% - 87.5% 38.75 - 27.02
5 sec	22.5% - 77.5% 38.3 - 27.32	17.5% - 75% 39.9 - 26.77	0.75% - 90% 40.75 - 27.05
10 sec	27.5% - 65% 35.57 - 27.95	22.5% - 70% 36.25 - 27.37	30% - 70% 37.1 - 28.01

Figure 6.8: Comparison between Enhanced MCTS and UCT players when UCT plays first. Blue indicates Enhanced MCTS and gray indicates UCT. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

6.6 Comparison of Enhanced MCTS with Minimax

with different time limitations were conducted and for each 40 games were played. Of the two values listed in each cell the first denotes the percentage of won games under all games and the second the mean value of all the scores. We observed the following:

1. The comparison of the two players showed that the Enhanced MCTS-based player did not perform better than the simple UCT-based player against Monte Carlo. Note, that the optimizations applied to enhance the MCTS algorithm were all evaluated against a basic UCT-based player. We can assume that the specific set of heuristic methods was sufficient to outperform the player against whom they were tested, but that does not imply that they are always equally effective. In order to reach the conclusion that a combination of techniques will always yield to a good result no matter who is the opponent, we need to test these techniques against many different opponents and accumulate numerous experimental results.
2. The fact that the Enhanced MCTS-based player is as competitive as the simple UCT-based player against Monte Carlo, does not imply that the two MCTS-based players are equally efficient. An interesting observation, is that the Enhanced MCTS -based player has in most cases smaller average score compared to the simple UCT-based player. In other words, although their overall performance seems to be the same, the Enhanced MCTS-based player places tiles more efficiently and achieves a better score.

6.6 Comparison of Enhanced MCTS with Minimax

Experiments were conducted to compare the performance of the Enhanced MCTS-based player with the Minimax-based player performance. However, due to the fact that there is no random factor in any of the two algorithms for the same period of time, they return the same best move. In all cases, the Enhanced MCTS beat Minimax, but for each combination of time limits the score was either similar or the same. There is no need to present detailed experimental results, since the results are almost always the same for the same combination of time limits.

6. COMPARISON OF PLAYERS

Enhanced MCTS \ MC	1 sec	5 sec	10 sec
1 sec	60% - 37.5% 26.95 - 32.55	40% - 57.5% 31.12 - 29.1	25% - 75% 33.27 - 27.55
5 sec	57.5% - 42.5% 27.25 - 31.32	50% - 47.5% 30.62 - 31.15	32.5% - 57.5% 30.42 - 29.82
10 sec	65% - 35% 27.42 - 30.4	47.5% - 52.5% 30.2 - 30.07	35% - 62.5% 33.65 - 29.35

Figure 6.9: Comparison between Enhanced MCTS and Monte Carlo players when Enhanced MCTS plays first. Blue indicates Enhanced MCTS and gray indicates Monte Carlo. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

MC \ Enhanced MCTS	1 sec	5 sec	10 sec
1 sec	57.5% - 42.5% 33.2 - 36.2	55% - 42.5% 32.85 - 36.02	45% - 52.5% 34.62 - 35.12
5 sec	72.5% - 27.5% 29.82 - 38.62	55% - 42.5% 31.4 - 35.75	55% - 47.5% 32.42 - 35.83
10 sec	75% - 20% 31.07 - 38.57	67.5% - 30% 29.67 - 38.2	62.5% - 35% 29.36 - 37.74

Figure 6.10: Comparison between Enhanced MCTS and Monte Carlo players when Monte Carlo plays first. Blue indicates Enhanced MCTS and gray indicates Monte Carlo. The first value in each cell denotes the player's percentage of won games and the second the mean value of all the scores of the player.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis we created four Blokus Duo players. The first was based on the well applied Minimax algorithm, the following two used the recently proposed MCTS algorithm and the fourth executed Monte Carlo simulations to find a solution. Comparison of the performance of all four of our players showed that for the same execution time, the Monte Carlo-based player performed slightly better than the MCTS-based players, due to the fact that in the first case, more simulations were executed, leading to more accurate results. Minimax always performed poorly, since the complexity of the Blokus Duo game did not allow searching deeper than depth 4 of the game tree. We also tested a group of optimization techniques to enhance the MCTS-based player. However, they only increased the player's performance when the opponent used the simple MCTS approach, but the same did not apply when playing against Monte Carlo. We did notice though, that the average score of all games was improved indicating that better moves were made by the enhanced MCTS player. Last but not least we modeled the software-based players for hardware implementation. The recursive nature of the Minimax algorithm, encourages sequential search and renders him less suitable for hardware implementation. On the other hand, the MCTS and Monte Carlo algorithms offer parallelism, since in both cases many simulations can be executed concurrently. But the memory requirements necessary for maintaining the UCT tree of the MCTS algorithm imply that it would be more efficient to maintain the tree, along with the actions performed on it on the software side and execute simulations on the hardware side.

7.2 Future Work

The time for completing a diploma thesis is always too short for implementing all ideas that arise during the work. At the end, three of them are outlined as outlook for future work. Blokus Duo is a relatively new game and few heuristics have been suggested that improve a player's performance. Among these, almost all of them are designed specifically for Minimax-based approaches and cannot be applied to the MCTS algorithm, since they would add a prohibitive execution time overhead. However, Minimax performs poorly in a complex game, such as Blokus Duo and therefore we are interested in finding an effective group of optimizations appropriate for the MCTS algorithm.

Furthermore, we intend to create a hardware MCTS-based player to evaluate its performance against the hardware Minimax-based player. In order to do so, we plan on designing an efficient way to find valid moves, that will not require an arbitrary number of clock cycles, in order to benefit the most from parallelism offered by the MCTS algorithm.

Finally, the Blokus Duo players described in this thesis were implemented to eventually compare the performance of the Minimax, MCTS and Monte Carlo algorithms. However, we have acquired valuable knowledge about critical points and tacticts of the game and we would like to create a Blokus Duo player that is competitive against any optimized Blokus Duo agent.

References

- [1] Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on* **4**(1) (2012) 1–43 [3](#)
- [2] Russell, S.J., Norvig, P., Canny, J.F., Malik, J.M., Edwards, D.D.: *Artificial intelligence: a modern approach*. Volume 74. Prentice hall Englewood Cliffs (1995) [12](#), [15](#), [18](#)
- [3] Salen, K.: *Rules of play: Game design fundamentals*. The MIT Press (2004) [15](#)
- [4] Rasmusen, E., Blackwell, B.: *Games and information*. Cambridge, MA (1994) [15](#)
- [5] Gelly, S., Silver, D.: Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence* **175**(11) (2011) 1856–1875 [20](#)
- [6] Schaeffer, J., Van den Herik, H.J.: *Games, computers, and artificial intelligence*. *Artificial Intelligence* **134**(1) (2002) 1–7 [21](#)
- [7] Lai, T.L., Robbins, H.: Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics* **6**(1) (1985) 4–22 [23](#)
- [8] Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. *Machine learning* **47**(2-3) (2002) 235–256 [23](#), [25](#), [31](#)
- [9] Agrawal, R.: Sample mean based index policies with $o(\log n)$ regret for the multi-armed bandit problem. *Advances in Applied Probability* (1995) 1054–1078 [23](#)

REFERENCES

- [10] Coulom, R.: Efficient selectivity and backup operators in monte-carlo tree search. In: Computers and games. Springer (2007) 72–83 [24](#), [30](#)
- [11] Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Machine Learning: ECML 2006. Springer (2006) 282–293 [24](#), [30](#)
- [12] Kocsis, L., Szepesvári, C., Willemson, J.: Improved monte-carlo search. Univ. Tartu, Estonia, Tech. Rep **1** (2006) [24](#)
- [13] Chaslot, G.: Monte-carlo tree search. PhD thesis, PhD thesis, Maastricht University (2010) [26](#), [27](#), [29](#)
- [14] Tesauro, G., Rajan, V., Segal, R.: Bayesian inference in monte-carlo tree search. arXiv preprint arXiv:1203.3519 (2012) [31](#), [32](#)
- [15] Gelly, S., Wang, Y.: Exploration exploitation in go: Uct for monte-carlo go. (2006) [31](#), [32](#)
- [16] Audibert, J.Y., Bubeck, S.: Minimax policies for adversarial and stochastic bandits. (2009) [32](#)
- [17] Bubeck, S., Munos, R., Stoltz, G., Szepesvári, C.: X-armed bandits. arXiv preprint arXiv:1001.4475 (2010) [32](#)
- [18] Auer, P., Cesa-Bianchi, N., Freund, Y., Schapire, R.E.: Gambling in a rigged casino: The adversarial multi-armed bandit problem. In: Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on, IEEE (1995) 322–331 [32](#)
- [19] Chaslot, G.M.J., Winands, M.H., HERIK, H.J.V.D., Uiterwijk, J.W., Bouzy, B.: Progressive strategies for monte-carlo tree search. New Mathematics and Natural Computation **4**(03) (2008) 343–357 [32](#)
- [20] Winands, M.H., Björnsson, Y.: Evaluation function based monte-carlo loa. In: Advances in Computer Games. Springer (2010) 33–44 [33](#)
- [21] Shibahara, K., Kotani, Y.: Combining final score with winning percentage by sigmoid function in monte-carlo simulations. In: Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On, IEEE (2008) 183–190 [35](#)

- [22] Cai, J.C., Lian, R., Wang, M., Canis, A., Choi, J., Fort, B., Hart, E., Miao, E., Zhang, Y., Calagar, N., et al.: From c to blokus duo with legup high-level synthesis [36](#)
- [23] Altman, E., Auerbach, J.S., Bacon, D.F., Baldini, I., Cheng, P., Fink, S.J., Rabbah, R.M.: The liquid metal blokus duo design. In: Field-Programmable Technology (FPT), 2013 International Conference on, IEEE (2013) 490–493 [37](#)
- [24] Huang, S.: Entwicklung einer künstlichen intelligenz für das strategische 2-personen-spiel blokus. (2012) [37](#)
- [25] Gelly, S., Wang, Y., Munos, R., Teytaud, O., et al.: Modification of uct with patterns in monte-carlo go. (2006) [39](#)
- [26] Lin, G.I.: Fuego go: The missing manual. (2009) [40](#)
- [27] Baudi, P.: Mcts with information sharing [40](#)
- [28] Chaslot, G.M.B., Winands, M.H., van Den Herik, H.J.: Parallel monte-carlo tree search. In: Computers and Games. Springer (2008) 60–71 [59](#), [60](#)