POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Design of Artificial Intelligence for Mancala Games

Relatore: Prof. Pier Luca Lanzi

Tesi di Laurea di:
Gabriele Rovaris, Matr. 823550

Anno Accademico 2015-2016

Abstract

Mancala games are an important family of board games with a millennial history and a compelling gameplay. The goal of this thesis is the design of artificial intelligence algorithms for mancala games and the evaluation of their performance. In order to do so, we selected five of the main mancala games, namely Awari, Oware, Vai Lung Thlan, Ohvalhu and Kalah. Then, we designed and developed several artificial intelligence mancala players: a greedy algorithm, that makes the locally optimal choice; three different versions of the classic Minimax, that applies tree search, guided by an heuristic function that evaluates the state of the game; the more recent Monte Carlo Tree Search algorithm, that builds the search tree in an incremental and asymmetric manner by doing many simulated games, balancing between exploration and exploitation. We evaluated all the mancala algorithms we designed through an extensive series of experiments and discussed their pros and cons.

Sommario

I mancala sono un'importante famiglia di giochi da tavolo grazie alla loro storia millenaria e alle meccaniche di gioco avvincenti. L'obiettivo di questa tesi è il design algoritmi di intelligenza artificiale per i mancala e l'analisi delle prestazioni in termini di percentuale di vittorie ottenute contro altri algoritmi. Per farlo, abbiamo scelto e implementato cinque tra i più importanti mancala, ovvero Awari, Oware, Vai Lung Thlan, Ohvalhu and Kalah. Abbiamo quindi progettato e sviluppato diversi algoritmi di intelligenza artificiale: un algoritmo greedy, ispirato alle tipiche strategie presentate nei libri di mancala; tre versioni differenti di Minimax, che applica una ricerca ad albero, guidato da una funzione euristica che valuta lo stato di gioco; il più recente Monte Carlo Tree Search, che costruisce l'albero di ricerca in modo incrementale ed asimmetrico simulando molte partite e bilanciando tra esplorazione e ottimizzazione. Abbiamo condotto una estesa serie di esperimenti per valutare la performance degli algoritmi di intelligenza artificiale sviluppati discutendone i pro e contro.

Ringraziamenti

Desidero ringraziare il professor Pier Luca Lanzi che con la sua guida sapiente ed infinito supporto ha reso questa tesi possibile.

Un ringraziamento particolare va a Beatrice Danesi che mi ha aiutato con le sue capacità di design.

Ringrazio tutti i miei amici che mi sono stati vicini in questi anni di studio.

Ringrazio anche i miei genitori che mi hanno sostenuto ed hanno sempre creduto in me ed infine mio fratello, che sta muovendo i primi passi nello studio dell'informatica e a cui questo lavoro è dedicato.

Contents

List of Figures

List of Tables

List of Algorithms

List of Source Codes

Chapter 1

Introduction

This thesis focuses on the application of Artificial Intelligence to board games. This area was born in the '50s, when the first AI algorithms, developed for checkers and chess, were able to play only at the level of beginners or they could only play the final moves of the game. In the following years, thanks to the design of more advanced techniques, AI programs could compete against human-expert players. In some cases, the programs are able to solve a game, i.e. predict the result of a game from a certain state, when all the players made the optimal moves.

The aim of this thesis is to design competitive artificial intelligence algorithms for mancala games, a family of board games played all around the world. Mancala games are several thousand years old and there are more than 800 traditional known games, played in 99 countries, with almost 200 games that have been designed in more recent times. Mancala games use a board composed of two rows of usually 6 pits that contain some counters; during their moves players sow these counters around the board and can sometimes capture them. The goal of the games is to capture more counters than the opponent. No luck is involved but high intellectual skills are required.

In this thesis, we selected five well-known games of the mancala family, namely Awari, Oware, Vai Lung Thlan, Ohvalhu and Kalah, and we designed five artificial intelligence algorithms: (i) a greedy algorithm inspired by players guides to Mancala games; (ii) the well-known minimax algorithm; (iii) an alpha-beta pruning minimax algorithm using alpha-beta pruning to reduce computational time and memory consumption; (iv) an advanced heuristic minimax strategy using minimax with a more refined heuristic function, based on the work of Divilly et al. [11]; (iv) Monte Carlo tree search (MCTS) with the Upper Confidence Bounds for trees (UCT)

and three simulation strategies. We evaluated the five algorithms with an extensive set of experiments. Finally, we developed an application to allow users to play all the games we studied against all the artificial intelligence we explored.

1.1 Original Contributions

This thesis contains the following original contributions:

- The analysis and implementation of the five mancala games, Awari, Oware, Vai Lung Thlan, Ohvalhu and Kalah.
- The design and implementation of five artificial intelligence algorithms for the mancala games: greedy, basic minimax, alpha-beta pruning minimax, advanced heuristic minimax and Monte Carlo tree search.
- An extensive experimental analysis of the performance of the developed artificial intelligence algorithms on the five selected games
- The development of an application to let users play the five mancala games we studied against all the artificial intelligence players we designed.

1.2 Thesis Outline

The thesis is structured as follows. In Chapter 1, we introduce the goals of this work, showing the original contributions and the thesis structure. In Chapter 2, we present several applications of AI in board and card games and we introduce the minimax and MCTS algorithms. In Chapter 3, we introduce an overview of the history of mancala games and then describe the rules of the mancala games we selected. In Chapter 4, we present the various AI we developed for mancala games. In Chapter 5, we show and discuss the results obtained from the experiments we did with the AIs we designed. In Chapter 6, we evaluate the work done for this thesis. In Appendix A, we briefly describe the two applications we developed.

Chapter 2

Artificial Intelligence in Board Games

In this chapter, we overview the most interesting applications of Artificial Intelligence in board games related to this work. Then we introduce Minimax and Monte Carlo tree search and we compare them showing advantages and disadvantages of the two methods.

2.1 Artificial Intelligence in Board Games

Artificial Intelligence aims to develop an opponent able to simulate a rational behavior, that is, do things that require intelligence when done by humans. Board games are particularly suited for this purpose because they are difficult to solve without some form of intelligence, but are easy to model. Usually, a board configuration corresponds to a state of the game, while a legal move is modeled with an action that changes the state of the game. Therefore, the game can be modeled with a set of possible states and a set of legal actions for each state.

2.1.1 Checkers

The first applications of Artificial Intelligence to board games were presented in the '50s, when Christopher Strachey [1] designed the first program for the game of *Checkers*. Strachey wrote the program for *Ferranti Mark I* that could play a complete game of Checkers at a reasonable speed using evaluation of board positions. Later Arthur Samuel developed an algorithm to play Checkers that was able to compete against amateur players [37]. The algorithm used by Samuel was called *Minimax with alpha-beta pruning*

(Section 2.2), which then became one of the fundamental algorithm of AI. Samuel tried to improve his program by introducing a method that he called *rote learning* [36]. This technique allowed the program to memorize every position it had already seen and the reward it had received. He also tried another way of learning, he trained his artificial intelligence by let it play thousands of games against itself [27]. At the end of the '80s Jonathan Schaeffer et al. began to work on *Chinook*, a program for Checkers developed for personal computers. It was based on alpha-beta pruning and used a precomputed database with more than 400 billion positions with at most 8 pieces in play. Chinook became world champion in '94 [30]. In 2007, Schaeffer et al. [29] were able to solve the game of Checkers (in the classical board 8 x 8) by proving that the game played without errors leads to a draw.

2.1.2 Chess

Chess is more widespread than Checkers but also much more complex. The first artificial intelligence algorithm to play this game was presented in the '50s by Dietrich Prinz [2]. Prinz's algorithm was able to find the best action to perform when it was only two moves away from checkmate [1], known as the mate-in-two problem; unfortunately the program was not able to play a full game due to the low computational power of the used machine, Ferranti Mark I. In 1962, Alan Kotok et al. designed Kotok-McCarthy, which was the first computer program to play Chess convincingly. It used Minimax with alpha-beta pruning and a single move took five to twenty minutes. In 1974, *Kaissa*, a program developed by Georgy Adelson-Velsky et al., became the first world computer chess champion. Kaissa was the first program to use bitboard (a special data structure), contained an opening book (set of initial moves known to be good) with 10000 moves, used a novel algorithm for move pruning, and could search during the opponent's move. The first computer which was able to defeat a human player was *Deep Thought* in 1989 [38]. The machine, created by the computer scientist of the IBM Feng-hsiung Hsu, defeated the Master of Chess David Levy, in a challenge issued by the latter. Later, Hsu entered in the *Deep Blue* project, a computer designed by IBM to play Chess only. The strength of Deep Blue was due to its high computational power, indeed it was a massively parallel computer with 480 processors. The algorithm to play Chess was written in C and was able to compute 100 million of positions per second. Its evaluation functions were composed by parameters determined by the system itself, analyzing thousands of champions' games. The program's knowledge of Chess has been improved by the grandmaster Joel Benjamin. The opening library was

provided by grandmasters Miguel Illescas, John Fedorowicz, and Nick de Firmian [41]. In 1996 Deep Blue became the first machine to win a chess game against the reigning world champion Garry Kasparov under regular time controls. However, Kasparov won three and drew two of the following five games, beating Deep Blue. In 1997 Deep Blue was heavily upgraded and it defeated Kasparov, becoming the first computer system to defeat a reigning world champion in a match under standard chess tournament time controls. The performance of Chess software are continuously improving. In 2009 the software *Pocket Fritz 4*, installed on a smart phone, won a category 6 tournament, being able to evaluate about 20000 positions per second [39].

2.1.3 Go

Another widely studied board game in the field of artificial intelligence is *Go*, the most popular board game in Asia. The board of Go is a square of 19 cells and basically a player can put a stone wherever he wants, therefore it has a very high branching factor in search trees (361 in the first ply), hence it is not possible to use the traditional methods such as Minimax. The first Go program was created in the '60s, when D. Lefkovitz [7] developed an algorithm based on pattern matching. Later Zobrist [7] wrote the first program able to defeat an amateur human player. The Zobrist's program was mainly based on the computation of a potential function that approximated the influence of stones. In the '70s Bruce Wilcox [7] designed the first Go program able to play better than an absolute beginner. His algorithm used abstract representations of the board and reasoned about groups. To do so, he developed the theory of *sector lines*, dividing the board into zones. The next breakthroughs were at the end of the '90s, when both abstract data structures and patterns were used [27]. These techniques obtained decent results, being able to compete against player at higher level than beginners, however the best results waere found with the Monte Carlo tree search algorithm, that we discuss in Section 4.5. DeepMind Technologies Limited, a british company acquired by Google in 2014, developed AlphaGo [32], the first computer Go program able to beat a human professional Go player without handicaps on a full-sized 19x19 board. Its algorithm uses Monte Carlo tree search to find its moves based on knowledge previously learned by machine learning. This proves the capabilities of Monte Carlo tree search, since Go has been a very challenging game for AIs.

2.1.4 Other Games

Thanks to the good results with Checkers, Chess, and Go, the interest of artificial intelligence was extended to other games, one of them is *Backgammon*. The main difficulty in creating a good artificial intelligence for this game is the chance event related to the dice roll. This uncertainty makes the use of the common tree search algorithm impossible. In 1992 Gerry Tesauro [27] combining the learning method of Arthur Samuel with neural networks techniques was able to design an accurate evaluator of positions. Thanks to hundreds of millions of training games, his program *TD-Gammon* is still considered one of the most strong player in the world.

In the '90s, programs able to play *Othello* were introduced. The main applications of artificial intelligence for this game was based on Minimax with alpha-beta pruning. In 1997 the program *Logistello*, created by Micheal Buro defeated the world champion Takaeshi Murakami. Nowadays Othello is solved for the versions with board dimensions 4 x 4 and 6 x 6. In the 8 x 8 version (the standard one), although it has not been proven mathematically, the computational analysis shows a likely draw. Instead, for the 10 x 10 version or grater ones, it does not exist any estimation, except of a strong likelihood of victory for the first player [40].

With the spread of personal computers, the research field of artificial intelligence has also been extended to modern board games. Since 1983 Brian Sheppard [31] started working on a program that can play the board game *Scrabble*. His program, called *Maven*, is still considered the best artificial intelligence to play Scrabble [43] and competes at the World Championships. Maven combines a selective move generator, the simulation of plausible game scenarios, and the B* search algorithm.

The artificial intelligence algorithms have been applied to many other games. Using these algorithms it was possible to solve games like *Tic Tac Toe*, *Connect Four*, *Pentamino*, *Gomoku*, *Nim*, etc [44].

2.2 Minimax

Minimax is a tree search algorithm that computes the move that minimize the maximum possible loss (or alternatively, maximize the minimum gain). The original version assumes a two-player zero-sum game but it has also been extended to more complex games. The algorithm starts from an initial state and builds up a complete tree of the game states, then it computes the best decision doing a recursive calculus which assumes that the first player tries to maximize his rewards, while the second tries to minimize the rewards

Algorithm 1 Minimax pseudo-code

```
1: function MINIMAX(node, maximizingPlayer)
2:     if node is terminal then
3:         return the reward of node
4:     end if
5:     if maximizingPlayer then
6:         bestValue ← −∞
7:         for all child of node do
8:             val ← MINIMAX(child,False)
9:             bestValue ← MAX(bestValue, val)
10:        end for
11:        return bestValue
12:    else
13:        bestValue ← +∞
14:        for all child of node do
15:            val ← MINIMAX(child,True)
16:            bestValue ← MIN(bestValue, val)
17:        end for
18:        return bestValue
19:    end if
20: end function
```

of the former [27]. The recursive call terminates when it reaches a terminal state, which returns a reward that is backpropagated in the tree according to the Minimax policy. The Minimax pseudo-code is given in Algorithm 1. Minimax turns out to be computationally expensive, mostly in games where the state space is huge, since the tree must be completely expanded and visited. *Alpha-beta pruning* is a technique that can be used to reduce the number of visited nodes, by stopping a move evaluation when it finds at least one possibility that proves the move to be worse than a previously examined one, as shown in Algorithm 2. In the worst case scenario, that is when moves are analyzed from the worst to the best ones, *Alpha-beta pruning* Minimax is identical to standard Minimax. Given a branching factor b in the best case scenario, that is when moves are analyzed from the best to the worst ones, *Alpha-beta pruning* allows to reduce the effective branching factor to $\sqrt{b}$, drastically reducing the computational time or, in alternative, the search could go twice as deep with the same amount of computation. In an average case scenario, the reduced branching factor is $\sqrt[4]{b^3}$ [35].

Algorithm 2 Alpha-beta pruning minimax pseudo-code

```
1: function α-β-MINIMAX(node, α, β, maximizingPlayer)
2:     if node is terminal then
3:         return the reward of node
4:     end if
5:     if maximizingPlayer then
6:         bestValue ← −∞
7:         for all child of node do
8:             val ← α-β-MINIMAX(child, α, β,False)
9:             bestValue ← MAX(bestValue, val)
10:            α ← MAX(α, bestValue)
11:            if β ≤ α then
12:                break
13:            end if
14:        end for
15:        return bestValue
16:    else
17:        bestValue ← +∞
18:        for all child of node do
19:            val ← α-β-MINIMAX(child, α, β,True)
20:            bestValue ← MIN(bestValue, val)
21:            β ← MIN(β, bestValue)
22:            if β ≤ α then
23:                break
24:            end if
25:        end for
26:        return bestValue
27:    end if
28: end function
```

2.3 Monte Carlo Tree Search

The traditional artificial intelligence algorithms for games are very powerful but require high computational power and memory for problems with a huge state space or high branching factor. Methodologies to decrease the branching factor have been proposed, but they often rely on an evaluation function of the state in order to prune some branches of the tree. Unfortunately such function may not be easy to find and requires domain-knowledge experts. A possible algorithm to overcome these issues is the *Monte Carlo method*. This technique can be used to approximate the game-theoretic value of a move by averaging the reward obtained by playing that move in a random sample of games. Adopting the notation used by Gelly and Silver [14], the value of the move can be computed as

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} I_i(s,a)\, z_i$$

where $N(s,a)$ is the number of times action a has been selected from state s, $N(s)$ is the number of times a game has been played out through state s, z_i is the result of the ith simulation played out from s, and $I_i(s,a)$ is 1 if action a was selected from state s on the ith playout from state s or 0 otherwise. If the actions of a state are uniformly sampled, the method is called *Flat Monte Carlo* and this achieved good results in the games Bridge and Scrabble, proposed by Ginsberg [16] and Sheppard [31] respectively. However Flat Monte Carlo fails over some domains, because it does not allow for an opponent model. Moreover, it has no game-theoretic guarantees, i.e even if the iterative process is executed for an infinite number of times, the move selected in the end may not be optimal.

In 2006 Rémi Coulom et al. combined the traditional tree search with the Monte Carlo method and provided a new approach to move planning in computer Go, now known as *Monte Carlo Tree Search* (MCTS) [9]. Shortly after, Kocsis and Szepesvári formalized this approach into the *Upper Confidence Bounds for Trees* (UCT) algorithm, which nowadays is the most used algorithm of the MCTS family. The idea is to exploit the advantages of the two approaches and build up a tree in an incremental and asymmetric manner by doing many random simulated games. For each iteration of the algorithm, a *tree policy* is used to find the most urgent node of the current tree, it seeks to balance the exploration, that is look at areas which are not yet sufficiently visited, and the exploitation, that is look at areas which can return a high reward. Once the node has been selected, it is expanded by taking an available move and a child node is added to it. A simulation is

then run from the child node and the result is backpropagated in the tree. The moves during the simulation step are done according to a *default policy*, the simplest way is to use a uniform random sampling of the moves available at each intermediate state. The algorithm terminates when a limit of iterations, time or memory is reached, for this reason MCTS is an *anytime* algorithm, i.e. it can be stopped at any moment in time returning the current best move. A Great benefit of MCTS is that the intermediate states do not need to be evaluated, as for Minimax with alpha-beta pruning, therefore it does not require a great amount of domain knowledge, usually only the game's rules are enough.

2.3.1 State of the art

Monte Carlo Tree Search attracted the interest of researchers due to the results obtained with *Go* [5], for which the traditional methods are not able to provide a competitive computer player against humans. This is due to the fact that Go is a game with a high branching factor, a deep tree, and there are also no reliable heuristics for nonterminal game positions (states in which the game can still continue, i.e. is not terminated) [8]. Thanks to its characteristics, MCTS achieved results that classical algorithms have never reached.

Hex is a board game invented in the '40s, played on a rhombus board with hexagonal grid with dimension between 11 x 11 and 19 x 19. Unlike Go, Hex has a robust evaluation function for the intermediate states, which is why it is possible to create good artificial intelligence using minimax [4]. Starting in 2007, Arneson et al. [4] developed a program based on Monte Carlo tree search, able to play the board game Hex. The program, called *MoHex*, won the silver and the gold medal at Computer Olympiads in 2008 and 2009 respectively, showing that it is able to compete with the artificial intelligence algorithms based on minimax.

MTCS by itself is not able to deal with *imperfect information* game, therefore it requires the integration with other techniques. An example where MCTS is used in this type of games is a program for *Texas Hold'em Poker*, developed by M. Ponsen et al. in 2010 [23]. They integrated the MCTS algorithm with a Bayesian classifier, which is used to model the behavior of the opponents. The Bayesian classifier is able to predict both the cards and the actions of the other players. Ponsen's program was stronger than rule-based artificial intelligence, but weaker than the program Poki.

In 2011, Nijssen and Winands [22] used MCTS in the artificial intelligence of the board game *Scotland Yard*. In this game the players have to

reach with their pawns a player who is hiding on a graph-based map. The escaping player shows his position at fixed intervals, the only information that the other players can access is the type of location (called *station*) where they can find the hiding player. In this case, MCTS was integrated with *Location Categorization*, a technique which provides a good prediction on the position of the hiding player. Nijssen and Winands showed that their program was stronger than the artificial intelligence of the game Scotland Yard for Nintendo DS, considered to be one of the strongest player.

In 2012, P. Cowling et al. [10] used the MCTS algorithm on a simplified variant of the game *Magic: The Gathering*. Like most of the card games, Magic has a strong component of uncertainty due to the wide assortment of cards in the deck. In their program, MCTS is integrated with *determinization* methods. With this technique, during the construction of the tree, hidden or imperfect information is considered to be known by all players.

Cowling et al. [34] also developed, in early 2013, an artificial intelligence for *Spades*, a four players card game. Cowling et al. used *Information Set Monte Carlo Tree Search*, a modified version of MCTS in which the nodes of the tree represents information sets. The program demonstrated excellent performance in terms of computing time. It was written to be executed on an Android phone and to find an optimal solution with only 2500 iterations in a quarter of a second.

2.3.2 The Algorithm

The MCTS algorithm relies on two fundamental concepts:

- The expected reward of an action can be estimated doing many random simulations.
- These rewards can be used to adjust the search toward a best-first strategy.

The algorithm iteratively builds a partial game tree where the expansion is guided by the results of previous explorations of that tree. Each node of the tree represents a possible state of the domain and directed links to child nodes represent actions leading to subsequent states. Every node also contains statistics describing at least a reward value and the number of visits. The tree is used to estimate the rewards of the actions and usually they become more accurate as the tree grows. The iterative process ends when a certain computational budget has been reached, it can be a time, memory or iteration constraint. With this approach, MCTS is able to expand only the most promising areas of the tree avoiding to waste most of the computational

budget in less interesting moves. At whatever point the search is halted, the current best performing root action is returned.

The basic algorithm can be divided in four steps per iteration, as shown in Figure 2.1:

- *Selection*: Starting from the root node n_0, MCTS recursively selects the most urgent node according to some utility function until a node n_n is reached that either represents a terminal state or is not fully expanded (a node representing a state in which there are possible actions that are not outgoing arcs from this node because they have not been expanded yet). Note that can be selected also a node that is not a leaf of the tree because it has not been fully expanded.

- *Expansion*: If the state s_n of the node n_n does not represent a terminal state, then one or more child nodes are added to n_n to expand the tree. Each child node n_l represents the state s_l reached from applying an available action to state s_n.

- *Simulation* (or *Rollout* or *Playout*): A simulation is run from the new nodes n_l according to the default policy to produce an outcome (or reward) Δ.

- *Backpropagation*: Δ is backpropagated to the previous selected nodes to update their statistics; usually each node's visits count is incremented and its average rewards updated according to Δ.

These can also be grouped into two distinct policies:

- *Tree policy*: Select or create a leaf node from the nodes already contained in the search tree (Selection and Expansion).

- *Default policy*: Play out the domain from a given nonterminal state to produce a value estimate (Simulation).

These steps are summarized in Algorithm 3. In this algorithm $s(n)$ and $a(n)$ are the state and the incoming action of the node n. The result of the overall search $a(\textsc{BestChild}(n_0))$ is the action that leads to the best child of the root node n_0, where the exact definition of "best" is defined by the implementation. Four criteria for selecting the winning action have been described in [28]:

- *max child*: select the root child with the highest reward.

- *robust child*: select the most visited root child.

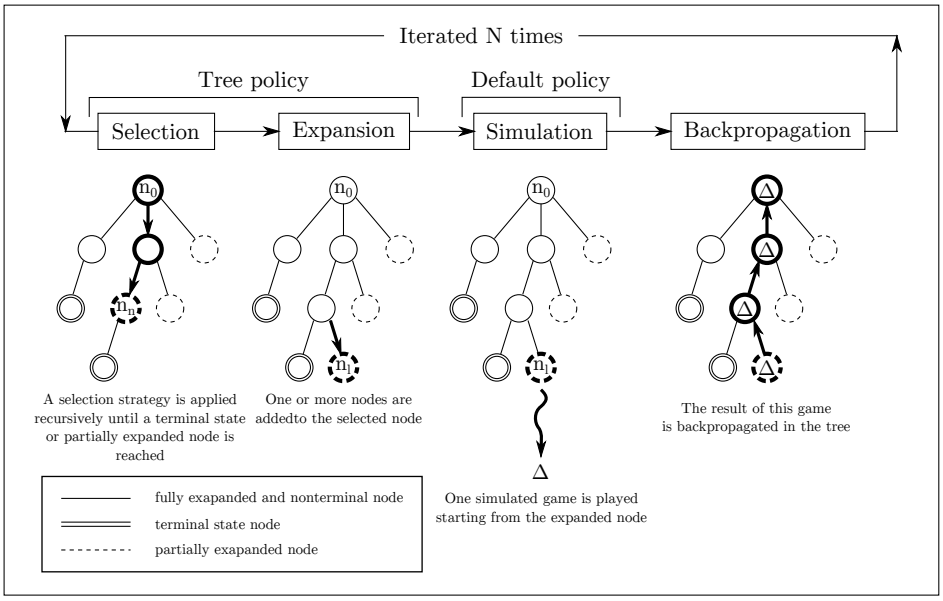


Figure 2.1: Steps of the Monte Carlo tree search algorithm.

Algorithm 3 Monte Carlo Tree Search

```
1: function MCTS(s_0)
2:     create root node n_0 with state s_0
3:     while within computational budget do
4:         n_l ← TreePolicy(n_0)
5:         Δ ← DefaultPolicy(s(n_l))
6:         Backpropagate(n_l, Δ)
7:     end while
8:     return a(BestChild(n_0))
9: end function
```

- *max-robust child*: select the root child with both the highest visits count and the highest reward; if none exists, then continue searching until an acceptable visits count is achieved.
- *secure child*: select the child which maximizes a lower confidence bound.

Note that since the MCTS algorithm does not force a specific policy, but leaves the choice of the implementation to the user, it is more correct to say that MCTS is a family of algorithms.

2.3.3 Upper Confidence Bounds for Trees

Since MCTS algorithm leaves the choice of the tree and default policy to the user, in this section we present the *Upper Confidence Bounds for Trees* (UCT) algorithm which has been proposed by Kocsis and Szepesvári in 2006 and is the most popular MCTS algorithm.

MCTS uses the tree policy to select the most urgent node and recursively expand the most promising parts of the tree, therefore the tree policy plays a crucial role in the performance of the algorithm. Kocsis and Szepesvári proposed the use of the *Upper Confidence Bound* (UCB1) policy which has been proposed to solve the *multi-armed bandit problem*. In this problem one needs to choose among different actions in order to maximize the cumulative reward by consistently taking the optimal action. This is not an easy task because the underlying reward distributions are unknown, hence the rewards must be estimated according to past observations. One possible approach to solve this issue is to use the UCB1 policy, which takes the action that maximizes the UCB1 value defined as

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

where $\bar{X}_j \in [0, 1]$ is the average reward from action j, n_j is the number of times action j was played, and n is the total number of plays. This formula faces the *exploitation-exploration dilemma*: the first addendum considers the current best action; the second term favors the selection of less explored actions.

The UCT algorithm takes the same idea of the UCB1 policy applying it to the selection step, it treats the choice of the child node to select as a multi-armed bandit problem. Therefore, at each step, it selects the child node n' that maximizes the UCT value defined as

$$UCT(n') = \frac{Q(n')}{N(n')} + 2C_p \sqrt{\frac{\ln N(n)}{N(n')}}$$

where $N(n)$ is the number of times the current node n (the parent of n') has been visited, $N(n')$ is the number of times the child node n' is visited, $Q(n')$ is the total reward of all playouts that passed through node n', and $C_p > 0$ is a constant. The term $\bar{X}_j$ in the UCB1 formula is replaced by $Q(n')/N(n')$ which is the actual average reward obtained from all the playouts. When $N(n')$ is zero, i.e. the child has not been visited yet, the UCT value goes to infinity, hence the child is going to be selected by the UCT policy. This is why in the selection step we use the UCT formula only when all the children of a node have been visited at least once. As in the UCB1 formula, there is the balance between the first (exploitation) and second (exploration) term. The contribution of the exploration term decreases as each node n' is visited, because it is at the denominator. On the other hand, the exploration term increases when another child of the parent node n is visited. In this way the exploration term ensures that even low-reward children are guaranteed to be selected given sufficient time. The constant in the exploration term C_p can be chosen to adjust the level of exploration performed. Kocsis and Szepesvári showed that $C_p = \sqrt{2}/2$ is optimal for rewards $\Delta \in [0, 1]$, this leads to the same exploration term of the UCB1 formula. If the rewards are not in this range, C_p may be determined from empirical evaluation. Kocsis and Szepesvári also proved that the probability of selecting a suboptimal action at the root of the tree converges to zero at a polynomial rate as the number of iterations grows to infinity. This means that, given enough time and memory, UCT converges to the Minimax tree and is thus optimal. Algorithm 4 shows the UCT algorithm in pseudocode. Each node n contains four pieces of data: the associated state $s(n)$, the incoming action $a(n)$, the total simulation reward $Q(n)$, and the visits count $N(n)$. $A(s)$ is the set of possible actions in state s and $f(s, a)$ is the state transition function, i.e. it returns the state s' reached by applying action a to state s. When a node is created, its values $Q(n)$ and $N(n)$ are set to zero. Note that in the UCT formula used in line 31 of the algorithm, the constant $c = 1$ means that we are using $C_p = \sqrt{2}/2$.

2.3.4 Benefits

MCTS offers three main advantages compared with traditional tree search techniques:

- *Aheuristic*: it does not require any strategic or tactical knowledge about the given game, it is sufficient to know only its legal moves and end conditions. This lack of need for domain-specific knowledge makes it applicable to any domain that can be modeled using a tree,

Algorithm 4 UCT algorithm

```
1: function UCT(s_0)
2:     create root node n_0 with state s_0
3:     while within computational budget do
4:         n_l ← TreePolicy(n_0)
5:         Δ ← DefaultPolicy(s(n_l))
6:         Backpropagate(n_l, Δ)
7:     end while
8:     return a(BestChild(n_0))
9: end function
10:
11: function TreePolicy(n)
12:     while s(n) is nonterminal do
13:         if n is not fully expanded then
14:             return Expand(n)
15:         else
16:             n ← BestUctChild(n, c)
17:         end if
18:     end while
19:     return n
20: end function
21:
22: function Expand(n)
23:     a ← choose untried actions from A(s(n))
24:     add a new child n' to n
25:     s(n') ← f(s(n), a)
26:     a(n') ← a
27:     return n'
28: end function
29:
30: function BestUctChild(n, c)
31:     return arg max_{n' ∈ children of n} Q(n')/N(n') + c sqrt(2 ln N(n) / N(n'))
32: end function
```

```
33: function DefaultPolicy(s)
34:     while s is nonterminal do
35:         a ← choose uniformly at random from A(s)
36:         s ← f(s, a)
37:     end while
38:     return reward for state s
39: end function
40:
41: function Backpropagate(n, Δ)
42:     while n is not null do
43:         N(n) ← N(n) + 1
44:         Q(n) ← Q(n) + Δ
45:         n ← parent of n
46:     end while
47: end function
```

hence the same MCTS implementation can be reused for a number of games with minimum modifications. This is the main characteristic that allowed MCTS to succeed in computer Go programs, because its huge branching factor and tree depth make it difficult to find suitable heuristics for the game. However, in its basic version, MCTS can have low performance and some domain-specific knowledge can be included in order to significantly improve the speed of the algorithm.

- *Anytime*: at the end of every iteration of the MCTS algorithm the whole tree is updated with the last calculated rewards and visits counts through the backpropagation step. This allows the algorithm to stop and return the current best root action at any moment in time. Allowing the algorithm for extra iterations often improves the result.

- *Asymmetric*: the tree policy allows spending more computational resources on the most promising areas of the tree, allowing an asymmetric growth of the tree. This makes the tree adapt to the topology of the search space and therefore makes MCTS suitable for games with high branching factor such as Go.

2.3.5 Drawbacks

Besides the great advantages of MCTS, there are also few drawbacks to take into consideration:

- *Playing Strength*: the MCTS algorithm may fail to find effective moves for even games of medium complexity within a reasonable amount of time. This is mostly due to the sheer size of the combinatorial move space and the fact that key nodes may not be visited enough times to give reliable estimates. Basically MCTS might simply ignore a deeper tactical moves combination because it does not have enough resources to explore a move near the root, which initially seems to be weaker in respect to the others.
- *Speed*: MCTS requires many iterations to converge to a good solution, for many applications that are difficult to optimize this can be an issue. Luckily, there exists a lot of improvements over the basic algorithm that can significantly improve the performance.

2.4 Summary

In this chapter, we overviewed applications of artificial intelligence in board games. We discussed artificial intelligence for Checkers, Chess, Go, and other board games. Then, we showed the well-known AI algorithm, Minimax, illustrating the alpha-beta pruning technique. Then, we introduced the Monte Carlo tree search algorithm, showing the state of the art, the basic algorithm, the Upper Confidence Bounds for Trees implementation, and benefits/drawback with respect to Minimax.

Chapter 3

Mancala

In this chapter, we present the family of board games known as mancala. We start from a brief history, present an overview of the mancala main board games, namely Wari, Awari, Oware, Ohvalhu, Vai Lung Thlan and Kalah, providing a detailed description of the rules.

3.1 History of Mancala Board Games

Mancala is a family of board games played all around the world, known also as 'sowing' games or 'count-and-capture' games. The word *mancala* comes from the Arabic word *naqala* meaning literally 'moved'. Many believe that there is an actual game named mancala, however the word *mancala* gathers hundreds of games under its name. More than 800 names of traditional mancala games are known, played in 99 countries, with almost 200 games designed in more recent times. Usually mancala games involve two players. They are played on boards with two three or four rows of pits. Sometimes they have additional pits at each end of the board, called stores. Each game begins with counters arranged in the pits according to the rules of that game. Often the goal of the game is to collect more counters than the opponent. Generally boards are made of wood, but holes can also be dug out of the earth or sand. Depending on what is readily available, the counters can be seeds, beans, nuts, pebbles, stones etc.

Figure 3.1: An example of starting board.

The origin and the diffusion of mancala gamesremain much of a mystery at the present time, but there are indications that the game is several thousand years old and was spread through the Bantu expansion, along trading routes and by the expansion of Islam. As a result, mancala games are played throughout the African continent as well as in India, Sri Lanka, Indonesia, Malaysia, the Philippines, Kazakhstan and Kyrgyzstan, etc. In many regions a single name is often used when referring to two or three different mancala games. Furthermore, it is not uncommon that neighboring african villages have different game rules for the same name, or different names for the same game. This has made recording the various games and supplying them a complete set of rules challenging. The main sources we used for rules are "The complete mancala games book" [25] and mancala.wikia.com [3].

3.2 Two-row Mancala

The most widespread type of game is the Two-Row mancala. Each row has a fixed number of pits that varies from one to fifty. The goal of the game is to gather more counters than the opponent and put them in the player's store. Usually they are games for two players and each player control the row closer to him. An example of board is shown in Figure 3.1: the south player controls the bottom row of pits and the store on the right side and plays first, the north player controls the top row of pits and the store on the left side and plays second. The move consists in choosing a pit of the player's row, picking up all the counters contained in that pit and sowing them counterclokwise usually.

Sowing is shown in Figure 3.2, the counters are put one by one in the following pits; when the end of the row is reached, the sowing continues on the opponent's row and continues this way until all the counters have been sown. Depending on the game being played, sowing either is also made in the store of the moving player or skips it. For example in Figure 3.2 the store is skipped. After the sowing, depending on the game, a capture may occur and a bonus turn may be gained. There are four types of captures [11]:

- **Number capture**: after a player has sown all of her counters and the

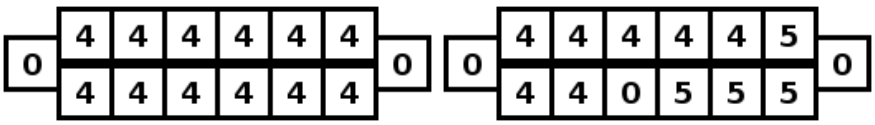


Figure 3.2: How a move works: south player moves form his third pit.

last counter sown is placed in one of their opponent's pits with that pit now containing a specific number of counters (for example, two or three counters in the games of Wari and Awari), then these counters may be captured.

- **Place capture**: after a player has sown all of her counters and the last counter sown is in one of their own pits with that pit now containing one counter, the counters in this pit and in the pit on the opposite side of the board (the opponents pits) are captured.
- **En-passant capture**: while a player is sowing her counters, a capture can occur if any of her own pits now contain a specific number of counters (for example, four counters in the games of Anywoli, Ba-awa, Pasu Pondi).
- **Store capture**: while a player is sowing her counters, if she pass over her own store, she captures one counter.

In our work we focused on Two-Row Mancala. In particular, we selected five games of the most well-known, developed and analized them in detail.

3.3 Traditional Games

3.3.1 Wari

Wari is the most widespread mancala game. It is played with the same rules in a large portion of Africa (Senegal, Gabon, Mali, Burkina Faso, Nigeria, Ghana, etc) and in the Caribbean (Antigua and Barbados) with different names (Owari, Ayo, Oware, Weri, Ouril, Awale, etc). Wari is played on a board of two rows, each consisting of six pits, that have a store at either end. Initially there are four counters in each pit. When a player is sowing, if the starting pit contained twelve counters or more (meaning that the move will complete a full wrap around the board), it is skipped. The capture type is number capture: if the last counter was placed into an opponent's pit that brought its total to two or three, all the counters in that pit are

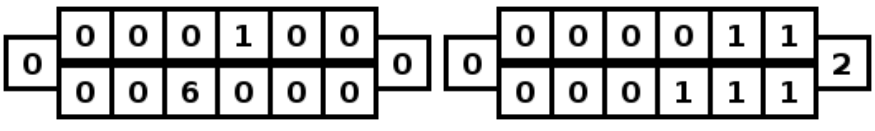
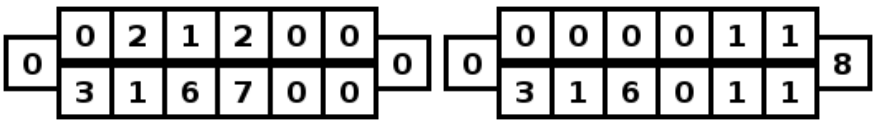


Figure 3.3: Wari: how a capture works. South player moves form her third pit and captures all the counters contained in the third pit of the opponent.

Figure 3.4: Wari: how a multiple capture works. South player moves form her fourth pit and captures all the counters contained in the third, fourth and fifth pit of the opponent.



captured and placed in the player's store. An example of capture is shown in Figure 3.3. Furthermore, if the previous-to-last counter also brought an opponent's pit to two or three, these are captured as well, and so on, as shown in Figure 3.4.

However, if a move would capture all the opponent's counters (a move known as *grand slam*), the counters are not captured and are left on the board, since this would prevent the opponent from continuing the game. A player has to make a move that allows the opponent to continue playing ('to feed')[1], if possible. If it is not possible to feed the opponent, the game ends. For example Figure 3.5 shows two forbidden moves. When an endless cycle is reached, that is a board position that repeats itself during the last stages of play, the players can agree to end the game. When the game ends, either because of a cycle or because a player cannot move, the remaining counters are given to the player that controls the pits they are in. The player who captured more counters wins the game. If both players have captured twenty-four counters, the game is declared a draw.

There are several variats of Wari. Grand slam Wari introduces the possibility to make a *grand slam* move. Since this move leaves the opponent without available moves, the player that makes a grand slam, captures all the counters left on the board and the game ends. Other variations of Wari that focus on *grand slam* require that either: (i) *grand slam* moves are forbidden;

[1]This rule has an interesting origin. Most probably, it comes from the mutual help between farmer: despite being rivals, a farmer would help another one during bad years.

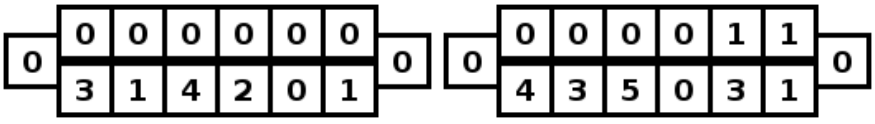


Figure 3.5: Wari: in the picture on the left south player must move from the third or sixth pit to give north player the possibility to move; in the other picture, south player cannot move from the third and fifth pit because she would capture all the counters on the opponent side of the board.

(ii) *grand slam* captures are allowed, however, all remaining counters on the board are awarded to the opponent; (iii) *grand slam* captures are legal, but the last (or first) pit is not captured. Another variant, called Cross-Wari, was invented by W. Dan Troyka in 2001 and modifies Wari rules as follows: the players make the moves from pits containing an odd number of counters clockwise and the moves from pits containing an even number of counters counterclockwise; when the game ends, no player captures the counters left on the table.

3.3.2 Awari

Awari was invented by Victor Allis, Maarten van der Meulen, and H. Jaap van den Herik in 1991 [19]. It is based on the rules of Wari and tweaked to better suit in AI experiments:

- The game ends if a position is repeated and the players are awarded the counters left on their side.
- A grand slam move is not allowed. The only exception to this rule happens when all the available moves are grand slam. In this case, the counters left on the board are awarded to the player on whose side they are (that is the player doing the grand slam).

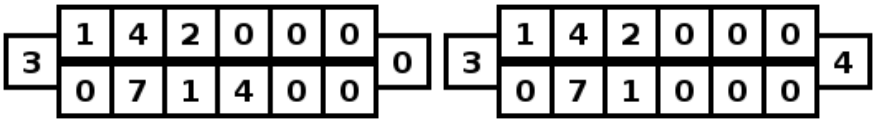
3.3.3 Oware

Oware was first mentioned by E. Sutherland, in her 1962 book on children's activities in Africa. Oware is a variant of Wari. In Oware, a capture happens also with four counters: if the last counter was placed into an opponent's pit that brought its total to two, three or four, all the counters in that pit are captured and placed in the player's store. If the previous-to-last counter also brought an opponent's pit to two, three or four these are captured as well, and so on. We implemented it using the rules of Awari (Subsection 3.3.2) to better fit AI experiments.

3.3.4 Vai Lung Thlan

Vai Lung Thlan is a game played by the Lushei Kuki clans in Assam, India. It is played on a board with two rows, each consisting of six pits that have a store at either end. Initially there are five counters in each pit. The capture is number capture, except that it can also happen in the moving player's pits: if the last counter is dropped into an empty pit on either side of the board, the player captures it as well as all the counters which precede this

Figure 3.6: Vai Lung Thlan: south player moves from her fourth pit and captures a chain of 4 single counters.



pit by an unbroken chain of single counters. Captured counters are removed and placed in the player's store. Figure 3.6 shows an example of a capture move. If a player has no counter in her pits and therefore no available move, she must pass. The game ends when no counters are left on the board or a player captured more than 30 counters. The player who has captured more counters wins. If each player has captured 30 counters, the game is a draw.

3.3.5 Ohvalhu

Ohvalhu is a game from the Maldives. Ohvalhu is also referred in the literature as Dakon [20]. It is played on a two-row board, each row consisting of eight pits, and has a store at either end. Initially there are eight counters in each pit. This game features the so called multi lap sowing that works as follows: if the last counter of the sowing falls into an occupied pit, all the counters are removed from that pit and are sown starting from that pit. The process continues until the last counter falls into a player's store, or an empty pit. This causes a feeling of randomness in a human player since the multi lap sowing might last for long, but is not random. If the last counter sown falls into a player's own store, they immediately gain another turn. If the last counter sown falls into an empty pit on the current player's side, then the player captures all the counters in the pit directly across from this

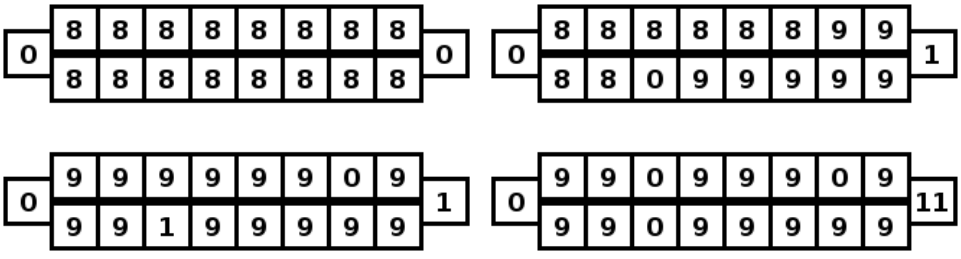


Figure 3.7: Ohvalhu: south player moves from her third pit. The result of the first lap is in the second board. The last counter is sown in the second pit of the opponent. From here the second lap starts. The result is shown in the third board. Then south player makes a capture, as shown in the fourth board.

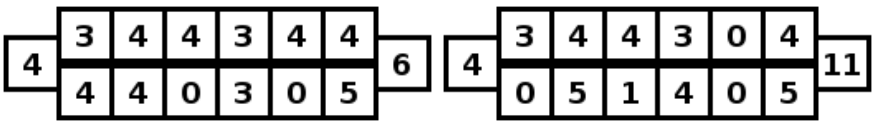
one, on the opponent's side and the counter causing the capture (place capture). If the opposing pit is empty, no counter is captured. Figure 3.7 shows how a move and captures work. The first round of the game ends when a player has no counters in her pits at the start of his turn. The remaining counters are awarded to his opponent. The counters are then redistributed as follows: from the left to the right, both players use the counters in their store and fill their pits with eight counters. Remaining counters are put in the store. If the number of counters is too low to fill all the pits, the pits that cannot be filled are marked with a leaf and are skipped in the next round. The player that won the last round starts playing. The game is over when at the end of a round one of the players cannot even fill one pit. For our work, we decided to implement only the first round and deem the player with the most counters following one round to be the winner. If each player has captured 64 counters, the game is a draw.

When the game starts, multi lap sowing and the extra turn rules allow the south player to move for several consecutive turns. Human players have found some sequences of moves that result in a win for the starting player without letting her opponent make a move. For example, here it is a 20 moves long winning sequence, counting pits from 1 to 8 from left to right: 1-8-5-4-8-3-5-2-7-7-8-3-2-1-3-6-8-2-8-7.

3.4 Kalah

Kalah was invented by William Julius Champion Jr., a graduate of Yale University, in 1940 [42]. The game is played on a board consisting of six pits per row and two stores; at the beginning there are four counters in each pit. The players control the pits on their side of the board and the store on their right. There are two ways to capture counters. Store capture: while a player is sowing counters and pass over their store, they sow in their store, too; it should be noted that this does not happen when passing over an opponent store. Place capture: when a player moves and their last counter is sown in one of her own pits which was previously empty (so that now it contains one counter), that counter and all the counters in the pit on the other side of the board are captured as shown in the example of Figure 3.8. Kalah has a rule that allows players to gain additional turns. This happens when the last counter of a move is sown in the player's store, as shown in Figure 3.9. Due to this rule, multiple moves can be made by the same player in sequence by properly choosing the pit to move from. The game ends when a player no longer has counters in any of his pits. The remaining counters are captured by his opponent. The player who has captured most counters is declared

Figure 3.8: Kalah: south player moves from her first pit and captures all the counters in her fifth pit and in her opponent's second pit.



the winner. If each player has captured twenty-four counters, the game is a draw.

3.5 Number of Positions on a Board

Mancala games might seem simple in that, they are deterministic, have perfect information and the available moves are always less or equal to the number of pits in a row (6 or 8 in the games we consider). However, a single move can have an effect on the contents of all the pits on the board and games can last for a great number of turns (in the order of 10^2). The number of possible positions may give insight on the complexity of mancala games. A position in mancala games consists of a certain distribution of the counters over the pits of the board, but also includes the captured counters in the stores [12]. Furthermore, a position includes the knowledge which player is to move next. The number of possible positions depends on the number of pits, stores and on the number of counters and it is computed as:

$$P(k, n, m) = k * \binom{n + m - 1}{m}$$

where k is the number of players, m is the total number of counters, n is the total number of pits and stores. The number of possible positions P(k, n, m) increases very rapidly. Table 3.1 has some example of this. Of course, only a fraction of all possible positions can actually appear during a game. This depends on the starting position of the game and on the rules. For instance, in Kalah, only about 5% of the possible positions can appear. That means 6.56622x10^{11} possible positions in Kalah.

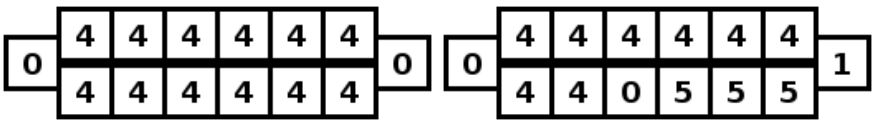


Figure 3.9: Kalah: a possible starting move of south player that allows her to capture a counter and get an extra turn by sowing the last counter in her store.

m	p
1	28
2	210
3	1,120
4	4,760
5	17,136
12 (1 counter per pit)	10,400,600
24 (2 counters per pit)	7,124,934,600
36 (3 counters per pit)	$5.25 \text{x} 10^{11}$
48 (4 counters per pit)	$1.31 \text{x} 10^{13}$
60 (5 counters per pit)	$1.72 \text{x} 10^{14}$
72 (6 counters per pit)	$1.47 \text{x} 10^{15}$

Table 3.1: Number of possible positions in a mancala board with two rows of six pits and two stores for two players.

3.6 Artificial Intelligence in Mancala

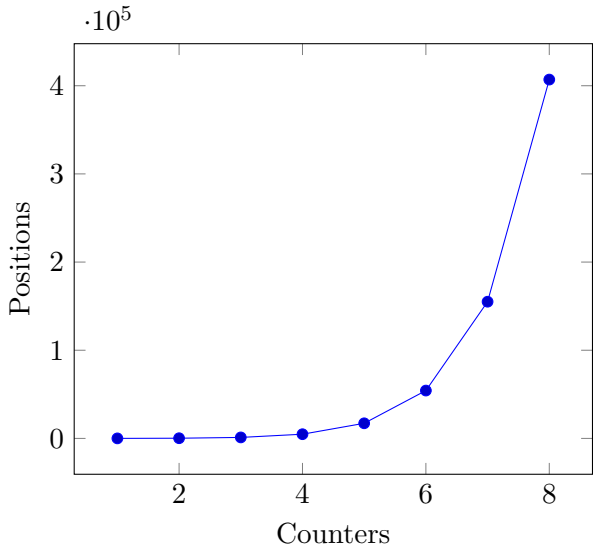
Mancala games have been studied relatively less than other games as chess or backgammon and most research is restricted to only two games: Kalah and Awari. Kalah has been studied as early as 1964 by Richard Russel [26]. In 1968, A.G. Bell wrote another computer program that could learn in some way from the errors that it made [6]. A year later, Slagle and Dixon used the game of Kalah to illustrate algorithms for playing several board games [33]. After that, Kalah lost the interest of AI game community until 2000, when, using some advanced techniques that were developed for chess, Irving, an undergraduate student at Caltech University, was able to find the winning strategy for Kalah [17].Concerning Awari, the interest in it started by the construction of a program called 'OLITHIDION' [21] and has been growing steadily since then. Romein and Bal solved Awari, using a large computing cluster. It executed a parallel search algorithm that determined the results for 889 billion positions in a 178 gigabyte database. The game is a draw when both players play optimally [24]. They also provided the database, the statistics, and an (infallible) awari-playing program online, but unfortunately the web site is no longer available. The game of Awari is the only mancala game that is played on the computer olympiad. This is an event in which all kind of computer programs compete in several classical games like Chess, Checkers and Go.
Although Awari is played with the same amount of pits and counters per pit as Kalah, it appears to have a higher complexity. The branching factor

of both is 6, but on average a game of Kalah lasts shorter than a game of Awari. This is due to the fact that Kalah allows store captures, ensuring that the number of counters in play is reduced faster.

3.7 Summary

In this chapter, we presented the family of board games known as mancala. We depicted its history and we illustrated the general rules of two-row mancala games. Then we explained the detailed rules of some of the main mancala games, Wari, Awari, Oware, Vai Lung Thlan, Ohvalhu, Kalah. Finally we gave an insight about their complexity and talked about previous research in mancala games. In the next chapter, we will see the artificial intelligence algorithms we have designed for mancala games.

Figure 3.10: Number of possible positions in a mancala board with two rows of six pits and two stores for two players.



Chapter 4

Artificial Intelligence for Mancala

In this chapter, we present the artificial intelligence strategies we developed for the mancala games considered in this work: (i) a greedy algorithm that applies well-known mancala strategies; (ii) an algorithm using minimax guided by a simple heuristic function; (iii) another minimax algorithm that uses alpha-beta pruning, a technique to reduce computational time and memory consumption; (iv) another alpha-beta pruning minimax algorithm that uses an improved heuristic function to evaluate mancala board; (v) finally, Monte Carlo Tree Search.

4.1 Greedy

The greedy player we implemented is based on the work of Neelam Gehlot from University of Southern California [13]. It applies basic rules to capture the most counters in one move. The algorithm considers the current state of the board and all the available moves. It evaluates the consequences of applying one move to the current board and selects the move that corresponds to the highest difference between the counters in the player's store and the ones in the opponent's store. when the move considered allows for an extra turn, the greedy algorithm analyzes in the same way the candidate moves of then next turn. This foresight is limited only to one extra turn.

The pseudo-code of the greedy strategy is shown in Algorithm 5. The best value is initialized to $-\infty$; the list of available moves is retrieved; for each move the evaluation function is called which returns the difference between the counters in the player's store and the ones in the opponent's store after playing the move; if this value is greater than the current best

Algorithm 5 Greedy

```
1: function GREEDY(board, player, opponent)
2:     bestValue ← −∞
3:     chosenMove ← 0
4:     M ← AVAILABLEMOVES(board)
5:     for all m in M do
6:         if EVALUATE(m, board, player, opponent) ≥ bestValue then
7:             bestValue ← EVALUATE(m, board, player, opponent)
8:             chosenMove ← m
9:         end if
10:    end for
11:    return chosenMove
12: end function
13:
14: function EVALUATE(move, board, player, opponent)
15:    evalBoard ← DOMOVE(board, move)
16:    return STORE(evalBoard, player) - STORE(evalBoard, opponent)
17: end function
```

value, the best value and the chosen move are updated; after the algorithm has considered all the available moves, it returns the chosen move. Note that the algorithm is *deterministic*.

4.2 Basic Minimax

This algorithm is based on Minimax (see Section 2.2) and it is inspired to the work of Neelam Gehlot from University of Southern California [13]. The root node of the tree is the current state of the board. From the root node, the algorithm considers the available moves and plays them one by one to obtain a new state of the board that represents a child of the root node. From this child node, the algorithm uses the same mechanism to get the children of this node, until the algorithm reaches a leaf, a node that either is a terminal state of the game or a node that reached a predefined depth of the tree. In the leaf, the algorithm applies an evaluation function that computes the difference between the counters in the player's store and the ones in the opponent's store. These values are then backpropagated from the leaves up to the root. In the end the root node obtains the values of every available move and selects the move with the highest value.

The pseudo-code of the implementation is in Algorithm 6. In Line 2

and 3 the algorithm checks if the node is a leaf and, in that case, returns the value of the evaluation function. From Line 5 to Line 12 the algorithm handles the case in which the turn player of the node is the player that is using minimax: the best value is first initialized to $-\infty$. All the children of this node are computed using the children function, for each child the minimax is called and the returned value is saved; if this value is higher than the best value found until now, the best value is replaced with it. After evaluating all the moves, the algorithm returns the best value. Line 13 to Line 20 are used when the turn player is the opponent: the procedure is different in that the best value is initialized to $+\infty$ and the minimum value is chosen as best value. Children function (Line 28) calls the available moves function to get all the possible moves in the board and then plays the moves one by one to obtain the child states; it adds them to a list that it returns to the caller. Note that the algorithm is *deterministic*.

4.3 Alpha-Beta Pruning Minimax

This algorithm uses Minimax with the Alpha-beta pruning technique (see Section 2.2) and is based on the work of Neelam Gehlot from University of Southern California [13]. Algorithm 8 shows its pseudo-code. Alpha and beta are initialized to $-\infty$ and ∞ respectively; while the turn player of the node is the player of the algorithm, if the best value obtained from the evaluation is higher than Alpha, Alpha is updated to that value (Line 12). When the turn player of the node is the opponent, if the best value obtained from the evaluation is less than Beta, Beta is updated to that value (Line 24). The algorithm also checks if Beta is less than or equal to Alpha (Line 13 and 25); in that case the cycle is stopped and the following branches are skipped, as their move is of no interest. We want to compare this algorithm to the previous one in order to test the benefits of Alpha-Beta Pruning. We expect this algorithm to considerably reduce computational time and memory consumption and be equally robust.

4.4 Advanced Heuristic Minimax

This minimax uses a more refined heuristic function, based on the work of Divilly et al. [11]. They developed a set of heuristics that could be applied to different mancala games, namely Awari, Oware, Vai Lung Thlan, Érhérhé. The heuristics are the following:

- *H1: Hoard as many counters as possible in one pit.* This heuristic,

Algorithm 6 Basic minimax pseudo-code

```
1: function MINIMAX(node, player, opponent, depth)
2:     if node is terminal OR depth = 0 then // leaf node
3:         return EVALUATE(board, player, opponent)
4:     end if
5:     if TURNPLAYER(board) = player then // player playing: maximize
6:         bestValue ← −∞
7:         children ← CHILDREN(board)
8:         for all child in children do
9:             val ← MINIMAX(child, player, opponent, depth − 1)
10:            bestValue ← MAX(bestValue, val)
11:        end for
12:        return bestValue
13:    else// opponent playing: minimize
14:        bestValue ← +∞
15:        children ← CHILDREN(board)
16:        for all child in children do
17:            val ← MINIMAX(child, player, opponent, depth − 1)
18:            bestValue ← MIN(bestValue, val)
19:        end for
20:        return bestValue
21:    end if
22: end function
23:
24: function EVALUATE(board, player, opponent)
25:    return STORE(board, player) - STORE(board, opponent)
26: end function
27:
28: function CHILDREN(board)
29:    M ← AVAILABLEMOVES(board)
30:    for all m in M do
31:        child ← DOMOVE(board, m)
32:        Add child to Children
33:    end for
34:    return Children
35: end function
```

Algorithm 7 Alpha-beta pruning minimax pseudo-code

```
1: function α-β-MINIMAX(node, α, β, player, opponent, depth)
2:     if node is terminal OR depth = 0 then // leaf node
3:         return EVALUATE(board, player, opponent)
4:     end if
5:     if TURNPLAYER(board) = player then // player playing: maximize
6:         bestValue ← −∞
7:         children ← CHILDREN(board)
8:         for all child in children do
9:             val ← α-β-MINIMAX(child, α, β, callingPlayer, depth − 1)
10:            bestValue ← MAX(bestValue, val)
11:            α ← MAX(α, bestValue)
12:            if β ≤ α then // pruning
13:                break
14:            end if
15:        end for
16:        return bestValue
17:    else// opponent playing: minimize
18:        bestValue ← +∞
19:        children ← CHILDREN(board)
20:        for all child in children do
21:            val ← α-β-MINIMAX(child, α, β, callingPlayer, depth − 1)
22:            bestValue ← MIN(bestValue, val)
23:            β ← MIN(β, bestValue)
24:            if β ≤ α then // pruning
25:                break
26:            end if
27:        end for
28:        return bestValue
29:    end if
30: end function
31:
32: function EVALUATE(board, player, opponent)
33:    return STORE(board, player) - STORE(board, opponent)
34: end function
35:
36: function CHILDREN(board)
37:    M ← AVAILABLEMOVES(board)
38:    for all m in M do
39:        child ← DOMOVE(board, m)
40:        Add child to Children
41:    end for
42:    return Children
43: end function
```

Table 4.1: Weights of the evolved player.

Heuristic	H1	H2	H3	H4	H5	H6
Weight	0.198649	0.190084	0.370793	1	0.418841	0.565937

with a look ahead of one move, works by attempting to keep as many counters as possible in the left-most pit on the board. At the end of the game, all the counters on a side of the board are awarded to that player's side. There is some evidence in literature that this is the best pit in which hoard counters [15].

- *H2: Keep as many counters on the players own side.* This heuristic is a generalized version of H1 and is based on the same principles.
- *H3: Have as many moves as possible from which to choose.* This heuristic has a look ahead of one and explores the possible benefit of having more moves to choose from.
- *H4: Maximize the amount of counters in a players own store.* This heuristic aims to pick a move that will maximize the amount of counters captured. It has a look ahead of one.
- *H5: Move the counters from the pit closest to the opponents side.* This heuristic, with a look ahead of one, aims to make a move from the right-most pit on the board. If it is empty, then the next pit is checked and so on. It was chosen because it has a good performance in Kalah [18] and the perfect player's opening move in Awari is to play from the rightmost pit [24].
- *H6: Keep the opponents score to a minimum.* This heuristic, with a look ahead of two moves, attempts to minimize the number of counters an opponent can win on their next move.

In their work [11], Divilly et al. applied a genetic algorithm to find a strong strategy that combines different heuristics. The formula of the evolved player is computed as:

$$V = H1 \times W1 + H2 \times W2 + H3 \times W3 + H4 \times W4 + H5 \times W5 - H6 \times W6$$

where V is the value of a move, $H_1 ... H_6$ are the heuristics, $W_1 ... W_6$ are the weights. Table 4.1 shows the weights of the evolved player. This evolved player performs strongly in Awari, Oware, Érhérhé, while its performance is not robust in Vai Lung Thlan.

Algorithm 8 Advanced heuristic function for minimax

```
1: function EVALUATE(board, player, opponent)
2:     H1 ← COUNTERSINLEFTMOSTPIT(board, player)
3:     H2 ← COUNTERSINALLPITS(board, player)
4:     H3 ← NUMBEROFNON-EMPTYPITS(board, player)
5:     H4 ← STORE(board, player)
6:     if PREVIOUSMOVE(board, player) was the rightmost then
7:         H5 ← 1
8:     else
9:         H5 ← 0
10:    end if
11:    H6 ← STORE(board, opponent)
12:
13:    return H1*W1+H2*W2+H3*W3+H4*W4+H5*W5−H6*W6
14: end function
```

Our advanced heuristic function uses this heuristic function to enhance the pruning of the tree and to select the best move. Its heuristic function is illustrated in Algorithm 8. Furthermore, the algorithm applies *beam search*: it considers a subset of the most promising moves and picks one of them randomly. Given the root node, its set of moves $m_0...m_x$, their relative values $v_0...v_x$ and the best value v_k, the result move is picked randomly between the moves that have a value $v_i \geq 0.99 \times v_k$ if the value is positive, or $v_i \geq 1.01 \times v_k$ if the value is negative. The goal is to remove the effect that small differences of values between two or more moves may have: this way, a move with a value slighty lower than the best value, has the same probability of being played of the move with the best value.

4.5 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) is a versatile algorithm. In comparison to Minimax it does not need an heuristic function. Each node represents a state of the board of the mancala game and the root node represents the current state. Each node provides the following informations: the available moves; if it is a terminal state, that is the game is over, and in that case which player won (or if it was a tie). MCTS uses these informations to play simulation games: from the root node, it picks moves until it reaches a terminal state, balancing between moves already played (exploitation) or moves never played (exploration). The result of the simulated game is then

backpropagated to the root node. After MCTS played all the simulation games, it chooses the best move.

We set the *computational budget* as *number of iterations*: the algorithm stops after a *number of iterations*, that is the number of nodes visited by the algorithm. The *tree policy* is the *upper confidence bounds for trees* (UCT) with constant $C_p = \sqrt{2}/2$: during the selection step the nodes are picked using the UCT formula (see Subsection 4.5). The UCT formula considers only the score of the player acting in that node. In the *expansion step*, the move to expand the node is chosen randomly from the moves available in the corresponding state. The *default policy* used is the *random sampling*: during the simulation step of the algorithm the simulated game is played using moves picked randomly within the available moves. The backpropagated reward is a vector containing the score of each player. The reward term for a victory is 1, the reward for a defeat is 0. At the end of the algorithm, the incoming move of the most visited root node is selected.

4.5.1 Other Simulation strategies

In the basic version of the MCTS algorithm we used a *Random Simulation* strategy, this is very efficient, but it does not always give good results. Previous studies [10] suggest that using heuristics in the simulation step can increase the performance of the algorithm. Therefore, we designed two other simulation strategies:

- *Greedy Simulation*: It simply uses the Greedy strategy to play the game until the end.
- *Epsilon-Greedy Simulation*: At each turn, it plays at random with probability ϵ, otherwise it plays the move chosen by the Greedy strategy.

Greedy Simulation, relying on a non-random simulation strategy, can be too restrictive, in fact, the results of a simulation from a given state will be always equal. Therefore, we think that adding some random factor might be beneficial.

4.6 Summary

In this chapter, we presented the artificial intelligence algorithms we used in our work, namely greedy, basic minimax, alpha-beta pruning minimax, advanced heuristic minimax and Monte Carlo tree search. We explained the

choices we made while developing them and provided their pseudo-code. In the next chapter, we will test them in several experiments to evaluate their performance.

Chapter 5

Experiments

In this chapter, we discuss the experiments we performed to evaluate the playing strength of the various strategies we developed. We start analyzing what happens when the players play randomly. Then, we evaluate the performance of the artificial intelligence algorithms we developed in the previous chapter: (i) the greedy algorithm; (ii) the basic minimax algorithm; (iii) the alpha-beta pruning minimax algorithm; (iv) the advanced heuristic minimax algorithm; (v) the Monte Carlo tree search algorithm. At the end, we discuss the performance of the simulation strategies we developed for Monte Carlo tree search.

5.1 Random Playing

In the first experiment [1], we used the random strategy for both players. We played 1000 times for each one of the five mancala games we developed (Awari, Oware, Vai Lung Thlan, Ohvalhu, Kalah). With this experiment we wanted to find out how much the games are biased towards a specific player. Table 5.1 shows the winning rates and the percentages of ties for all the games we chose when both players apply a random strategy. Interestingly some games are biased towards the south player (the first one to play), while others towards the north player (the second one to play). Kalah, Vai Lung Thlan and Ohvalhu have a bias towards the south player when playing randomly. In these games, there are sequence of moves that brings the south player to make a capture as soon as her second turn, not considering store capture. The first possible capture of the north player can happen on her second turn, thus later in the game. Furthermore, Kalah and Ohvalhu allows to sow in a player's own store, so the player going first is the first one able

[1] In all the experiments, we used a Dual Xeno 8-core with 64GB of RAM.

to gain multiple turns while capturing some counters. These facts give the south player an advantage that is especially great in Ohvalhu (the difference of the winning rate is 17.62%), less remarkable in Kalah (3.75%) and even lesser in Vai Lung Thlan (1.52%). The data are consistent with what we expected: Vai Lung Thlan is the longest game and as such the advantage gained by the first player is less impactful, while Ohvalhu is considerably faster due to multi lap sowing, therefore going first is really favorable.

When playing randomly, Awari and Oware are biased towards the north player, and in fact, the considerations on the sequences of moves needed to make a capture are reversed: the fastest sequences of moves that bring to a capture are in favour of the second player. She might make a capture on her second turn, while the fastest capture of the first player can happen only in her third turn. No more considerations arise, since store captures do not exist in these game. Thus, the north player has an advantage: the difference of the winning rate is 3.4% in Awari and 4.47% in Oware. To further support our hypothesis, Oware has a greater difference in winning rates, since it allows easier captures with regards to Awari (Awari allows captures when a pit contains 2 or 3 counters, Oware allows captures when a pit contains 4 counters, too). In fact, because Oware games are shorter, going second is a more impactful advantage.

Table 5.1: Random versus Random

Game	South Player	North Player	Ties
Awari	45.40%	48.8%	5.8%
Oware	45.34%	49.81%	4.85%
Vai Lung Thlan	47.45%	45.93%	6.62%
Ohvalhu	57.23%	39.61%	3.16%
Kalah	48.64%	44.89%	6.47%

5.2 Greedy

In the second experiment, we wanted to evaluate the performance of the greedy player we developed (see Section 4.1). We played 200 games for each of the five mancala games we developed (Awari, Oware, Vai Lung Thlan, Ohvalhu, Kalah) when it is playing versus random. The starting player (the south player) alternates between greedy and random. Table 5.2 shows the results. As expected, we note that even a simple greedy strategy is a huge improvement when compared to the random strategy. Winning rates of greedy against random are considerably high, over 75%. The game in

which greedy obtains the lowest winning rate is Vai Lung Thlan. This one is the longest game, suggesting that in longer games some kind of foresight might be needed.

Then, we played 200 games for each of the five mancala games when greedy plays against itself. Greedy is a deterministic algorithm, thus, when facing another deterministic algorithm, it outputs always the same set of moves. This fact lead to the repetition of the same game. To avoid this, we set that the first 10 moves of each player are randomly chosen. Table 5.3 shows the results. We note that the south player has an advantage. This fact is in contrast with what happened in the case random against random, where two games, Awari and Oware, were biased towards the second player, however it can be easily explained. After the initial ten random moves, the board is likely to present some capture moves, thus, the first player is the first one to have the chance to make such moves and take the lead. The Table 5.4 shows the very tiny time the greedy algorithm takes to choose its moves during a game.

Table 5.2: Winning rates of greedy against random. The columns South and North contain the winning rate when the player is first and second, respectively. The column Wins is the average of these values, that is the winning rates when each player plays an equal number of games as first and as second.

	Greedy			Random		
Game	Wins	South	North	Wins	South	North
Awari	97.5%	96%	99%	1.5%	0%	3%
Oware	89.5%	86%	93%	6%	2%	10%
Vai Lung Thlan	75%	74%	76%	21%	21%	21%
Ohvalhu	99.5%	100%	99%	0%	0%	0%
Kalah	96%	97%	95%	3.5%	5%	2%

Table 5.3: Greedy versus greedy

Game	South	North	Ties
Awari	48%	44%	8%
Oware	55%	39.5%	5.5%
Vai Lung Thlan	50%	46%	4%
Ohvalhu	57.5%	40%	2.5%
Kalah	53%	39.5%	7.5%

Table 5.4: The average time the greedy algorithm takes for a game.

Game	Time
Awari	0.0048
Oware	0.0024
Vai Lung Thlan	0.0599
Ohvalhu	0.0033
Kalah	0.001

5.3 Basic Minimax

In this experiment, we want to evaluate the performance of the basic minimax player (BMM) we developed (see Section 4.2). We chose three depths for the search tree: 1, 4 and 8 (denoted as BMM1, BMM4, BMM8). We selected these depths to cover a wide spectrum of skill: BMM1 represents a beginner player, that looks only at the immediate consequences of its moves; BMM4 is an intermediate player, able to see ahead of 4 moves; BMM8 is an avanced player, that examinates what will happen in the next 8 turns. Then, for each of these depths and for each of the mancala games we developed, we played 200 games comparing BMM against random, against greedy and against itself. In 100 of these games BMM played first (south player) and in the remaining 100 it played second (north player).

BMM proves to be considerably stronger than the random player (Tables 5.5 and 5.5), especially if we use higher depth. In our test, BMM8 wins always against random. Still, BMM1 wins 75.5% of the games in Vai Lung Thlan. This is because Vai Lung Thlan is the longest game we developed and more in depth search is needed to achieve good results.

Because BMM is a deterministic algorithm, similarly to what we did with greedy, we set that the first 10 moves of each player are randomly chosen in BMM against greedy and in BMM against BMM, to avoid the repetition of the same game. Tables 5.7, 5.8 and 5.9 report the results of BMM against greedy. Note that in Ohvalhu and Kalah, greedy has an edge against BMM1. One may argue that BMM1 should perform similarly to greedy: this is generally true, in fact, these algorithms are substantially even in Awari, Oware and Vai Lung Thlan. However, we implemented greedy such that it is able to exploit the extra turns that Kalah and Ohvalhu allow. Prevedibly, with higher depths BMM gets stronger and achieves a higher winrate against greedy in all the games (Table 5.8).

When BMM plays against itself (from Table 5.10 to Table 5.15), it is clear that the algorithm with higher depth is stronger, especially in Awari,

Oware and Vai Lung Thlan. When playing Ohvalhu and Kalah, however, a greater depth of search results in a smaller advatage. When both BMM algorithms have the same depth, we note that in general all the games favour the south player and that the higher the depth the higher is the advantage of the south player. For example, in Oware when BBM4 plays against BBM4, the difference in winning rates in favour of the south player is 2.5%, however when BBM8 plays against BBM8 the difference in winning rates in favour of the south player rises to 9%. This fact confirms that going first is better, especially if the player going first is strong.

Table 5.5: BMM1 versus random

	BMM1			random		
Game	Wins	South	North	Wins	South	North
Awari	94.5%	95%	94%	3.5%	3%	4%
Oware	90%	89%	91%	7.5%	7%	8%
Vai Lung Thlan	75.5%	72%	79%	20.5%	17%	24%
Ohvalhu	96%	100%	92%	4%	8%	0%
Kalah	92.5%	92%	93%	6%	5%	7%

Table 5.6: BMM4 versus random

	BMM4			random		
Game	Wins	South	North	Wins	South	North
Awari	100%	100%	100%	0%	0%	0%
Oware	99.5%	100%	99%	0%	0%	0%
Vai Lung Thlan	99.5%	100%	99%	0.5%	1%	0%
Ohvalhu	100%	100%	100%	0%	0%	0%
Kalah	98.5%	97%	100%	1%	0%	2%

Table 5.7: BMM1 versus greedy

	BMM1			greedy		
Game	Wins	South	North	Wins	South	North
Awari	44%	43%	45%	51%	52%	50%
Oware	46%	46%	46%	50%	48%	52%
Vai Lung Thlan	49%	45%	53%	44.5%	44%	45%
Ohvalhu	40%	51%	29%	56.5%	68%	45%
Kalah	38.5%	41%	36%	57%	62%	52%

Table 5.8: BMM4 versus greedy

	BMM4			greedy		
Game	Wins	South	North	Wins	South	North
Awari	88%	84%	92%	9%	5%	13%
Oware	85.5%	91%	80%	13.5%	18%	9%
Vai Lung Thlan	97%	97%	97%	2.5%	2%	3%
Ohvalhu	60%	72%	48%	38%	50%	26%
Kalah	71.5%	74%	69%	24.5%	26%	23%

Table 5.9: BMM8 versus greedy

	BMM8			greedy		
Game	Wins	South	North	Wins	South	North
Awari	95.5%	94%	97%	3.5%	3%	4%
Oware	83%	89%	77%	16%	23%	9%
Vai Lung Thlan	97.5%	98%	97%	2%	3%	1%
Ohvalhu	62.5%	70%	55%	35%	41%	29%
Kalah	75%	84%	66%	19.5%	29%	10%

Table 5.10: BMM1 versus BMM1

Game	South	North	Ties
Awari	46.5%	46.5%	7%
Oware	49.5%	46.5%	4%
Vai Lung Thlan	42%	53.5%	4.5%
Ohvalhu	59%	38%	3%
Kalah	52.5%	37.5%	10%

Table 5.11: BMM1 versus BMM4

	BMM1			BMM4		
Game	Wins	South	North	Wins	South	North
Awari	13%	14%	12%	84%	85%	83%
Oware	16%	19%	13%	82.5%	85%	80%
Vai Lung Thlan	3%	3%	3%	94%	94%	94%
Ohvalhu	23.5%	29%	18%	75%	82%	68%
Kalah	20%	21%	19%	75.5%	79%	72%

Table 5.12: BMM1 versus BMM8

	BMM1			BMM8		
Game	Wins	South	North	Wins	South	North
Awari	5.5%	11%	0%	93%	99%	87%
Oware	14%	13%	15%	83.5%	83%	84%
Vai Lung Thlan	1.5%	1%	2%	97.5%	98%	97%
Ohvalhu	25%	33%	17%	72%	78%	66%
Kalah	16%	23%	9%	79.5%	85%	74%

Table 5.13: BMM4 versus BMM4

Game	South	North	Ties
Awari	49%	49.5%	1.5%
Oware	49.5%	47%	3.5%
Vai Lung Thlan	52%	46.5%	1.5%
Ohvalhu	55.5%	44%	0.5%
Kalah	55.5%	39.5%	5%

Table 5.14: BMM4 versus BMM8

	BMM4			BMM8		
Game	Wins	South	North	Wins	South	North
Awari	19%	21%	17%	80%	82%	78%
Oware	26%	29%	23%	71.5%	74%	69%
Vai Lung Thlan	6%	8%	4%	93%	96%	90%
Ohvalhu	46.5%	54%	39%	53.5%	61%	46%
Kalah	38%	42%	34%	58.5%	63%	54%

Table 5.15: BMM8 versus BMM8

Game	South	North	Ties
Awari	51%	43%	6%
Oware	52%	43%	5%
Vai Lung Thlan	52.5%	44.5%	3%
Ohvalhu	56%	43%	1%
Kalah	54%	38.5%	7.5%

5.4 Alpha-Beta Pruning Minimax

In the third experiment, we aimed at evaluating the performance of the alpha-beta pruning minimax algorithm (ABMM) we developed (see Section 4.3). We selected three depths for the search tree: 1, 4 and 8 (denoted as ABMM1, ABMM4, ABMM8). These depths show three different level of skills: ABMM1 represents a beginner player, that looks only at the immediate consequences of its moves; ABMM4 is an intermediate player, able to see ahead of 4 moves; ABMM8 is an avanced player, that examinates what will happen in the next 8 turns. Then, for each of these depths and for each of the mancala games we developed, we played 200 games to compare the performance of ABMM against the previous algorithms and against itself. In 100 of these games ABMM was the first one to play (south player) and in the remaining 100 it was second (north player).

ABMM is much stronger than the random player (Tables 5.16 and 5.17), especially with higher depth. In our test, ABMM8 won always against random.

Because ABMM is a deterministic algorithm, when ABMM is facing another deterministic algorithm, we set that the first 10 moves of each player are randomly chosen, to avoid the repetition of the same game, as we did in previous experiments. We also measured the computational time of the algorithms when ABMM plays against BMM to quantify the time saved using the alpha-beta pruning technique.

When ABMM1 plays against greedy (Table 5.18), the results show that the algorithms perform similarly in the games that do not allow extra turns, like Awari, Oware and Vai Lung Thlan. In the games that allow extra turns, like Ohvalhu and Kalah, the greedy player is stronger, because it is able to take advantage of the extra turns. As expected, with higher depths, ABMM achieves a higher winrate against the greedy player (Table 5.19). Still, when playing Ohvalhu ABMM8 has a small winrate against the greedy strategy, only 51% (Table 5.20). On the one hand this shows that in Ohvalhu a deeper search does not result in higher winrate as much as in other mancala games; on the other hand it shows that the ability to exploit extra turns of the greedy algorithm is particularly strong in Ohvalhu.

When ABMM plays against itself (from Table 5.21 to Table 5.26) with different depths, we note that the higher the depth is, the more ABMM performs. Higher depths are especially advantegeous in Awari, Oware and Vai Lung Thlan; in these games tree search with higher depths is much more beneficial than in Ohvalhu and Kalah. For example when ABMM4 plays against ABMM8 (Table 5.25), ABMM8 wins 83.5% of the games in

Awari, while it wins only 53% of the games in Kalah.

At the end, we compare ABMM and BMM (from Table 5.27 to Table 5.36): the results show that BMM and ABMM have similar performance. Their winrates are very similar when facing the same type of algorithm: for example, when playing Awari against the greedy player, BMM4 and ABMM4 winrates are 88% (Table 5.8) and 87% (Table 5.19), respectively; when playing Oware against the greedy player, BMM8 and ABMM8 winrates are 97.5% (Table 5.9) and 98% (Table 5.20), respectively. This is even more noticeable when they face each other with the same depth (Tables 5.27, 5.31 and 5.36). It might seem that BMM is worse than ABMM in Oware, looking at their winrates with depth 1 (44.5% and 54%, respectively), however the data are capsized with depth 4 (50% and 45%, respectively). In particular, increasing the depth does not favour one algorithm or the other one, supporting our hypothesis. Yet, the substantial difference between the two algorithms is the computational cost. Tables 5.33 and 5.37 show the time the algorithms take for a game and the time saved with alpha-beta pruning, used by ABMM. When the depth of the algorithms is 4 the time is approximatively halved, however with depth 8 the time saved is approximatively 90%. Ohvalhu is the game that benefits the most from alpha beta pruning, probably because it has the higher branching factor, having up to eight moves available per turn (other games have up to six moves per turn).

Table 5.16: ABMM1 versus random

	ABMM1			random		
Game	Wins	South	North	Wins	South	North
Awari	95.5%	92%	99%	3%	1%	5%
Oware	86.5%	82%	91%	12%	9%	15%
Vai Lung Thlan	73%	66%	80%	25%	19%	31%
Ohvalhu	93.5%	96%	91%	4.5%	5%	4%
Kalah	94.5%	95%	94%	4%	5%	3%

Table 5.17: ABMM4 versus random

	ABMM4			random		
Game	Wins	South	North	Wins	South	North
Awari	100%	100%	100%	0%	0%	0%
Oware	100%	100%	100%	0%	0%	0%
Vai Lung Thlan	100%	100%	100%	0%	0%	0%
Ohvalhu	100%	100%	100%	0%	0%	0%
Kalah	97.5%	97%	98%	2%	2%	2%

Table 5.18: ABMM1 versus greedy

	ABMM1			greedy		
Game	Wins	South	North	Wins	South	North
Awari	49%	54%	44%	47%	53%	41%
Oware	40%	41%	39%	53%	53%	53%
Vai Lung Thlan	43%	44%	42%	53%	53%	53%
Ohvalhu	40%	57%	23%	58%	74%	42%
Kalah	38.5%	48%	29%	57.5%	68%	47%

Table 5.19: ABMM4 versus greedy

	ABMM4			greedy		
Game	Wins	South	North	Wins	South	North
Awari	87%	89%	85%	11%	14%	8%
Oware	84.5%	80%	89%	13.5%	10%	17%
Vai Lung Thlan	96%	95%	97%	3%	2%	4%
Ohvalhu	53.5%	65%	42%	42.5%	54%	31%
Kalah	66.5%	74%	59%	25.5%	30%	21%

Table 5.20: ABMM8 versus greedy

	ABMM8			greedy		
Game	Wins	South	North	Wins	South	North
Awari	93.5%	96%	91%	5%	7%	3%
Oware	86%	87%	85%	12%	12%	12%
Vai Lung Thlan	98%	99%	97%	2%	3%	1%
Ohvalhu	51%	62%	40%	47%	59%	35%
Kalah	71.5%	80%	63%	24%	30%	18%

Table 5.21: ABMM1 versus ABMM1

Game	South	North	Ties
Awari	47%	46.5%	6.5%
Oware	50.5%	41%	8.5%
Vai Lung Thlan	48.5%	48%	3.5%
Ohvalhu	58%	40.5%	1.5%
Kalah	50.5%	41%	8.5%

Table 5.22: ABMM1 versus ABMM4

	ABMM1			ABMM4		
Game	Wins	South	North	Wins	South	North
Awari	12%	13%	11%	86.5%	88%	85%
Oware	19%	23%	15%	78.5%	83%	74%
Vai Lung Thlan	1.5%	1%	2%	97%	96%	98%
Ohvalhu	31%	38%	24%	66.5%	75%	58%
Kalah	17%	22%	12%	80.5%	86%	75%

Table 5.23: ABMM1 versus ABMM8

	ABMM1			ABMM8		
Game	Wins	South	North	Wins	South	North
Awari	5.5%	7%	4%	92%	94%	90%
Oware	10%	11%	9%	88%	88%	88%
Vai Lung Thlan	0.5%	0%	1%	99.5%	99%	100%
Ohvalhu	27.5%	32%	23%	71.5%	77%	66%
Kalah	16%	21%	11%	81%	87%	75%

Table 5.24: ABMM4 versus ABMM4

Game	South	North	Ties
Awari	46.5%	47%	6.5%
Oware	52%	46%	2%
Vai Lung Thlan	52.5%	44.5%	3%
Ohvalhu	61.5%	37.5%	1%
Kalah	50.5%	45%	4.5%

Table 5.25: ABMM4 versus ABMM8

	ABMM4			ABMM8		
Game	Wins	South	North	Wins	South	North
Awari	13.5%	19%	8%	83.5%	90%	77%
Oware	18.5%	25%	12%	79%	85%	73%
Vai Lung Thlan	7%	7%	7%	92.5%	92%	93%
Ohvalhu	37%	39%	35%	60.5%	63%	58%
Kalah	40.5%	52%	29%	53%	60%	46%

Table 5.26: ABMM8 versus ABMM8

Game	South	North	Ties
Awari	46%	46.5%	7.5%
Oware	51%	43.5%	5.5%
Vai Lung Thlan	48%	47%	5%
Ohvalhu	62%	36.5%	1.5%
Kalah	54.5%	40.5%	5%

Table 5.27: BMM1 versus ABMM1

	BMM1			ABMM1		
Game	Wins	South	North	Wins	South	North
Awari	51.5%	53%	50%	40%	37%	43%
Oware	44.5%	42%	47%	54%	52%	56%
Vai Lung Thlan	47%	40%	54%	46.5%	41%	52%
Ohvalhu	54%	60%	48%	44%	49%	39%
Kalah	52%	54%	50%	42%	41%	43%

Table 5.28: BMM1 versus ABMM4

	BMM1			ABMM4		
Game	Wins	South	North	Wins	South	North
Awari	13.5%	15%	12%	83.5%	86%	81%
Oware	20%	21%	19%	77.5%	77%	78%
Vai Lung Thlan	4.5%	4%	5%	94.5%	94%	95%
Ohvalhu	30%	33%	27%	68.5%	73%	64%
Kalah	21%	25%	17%	73.5%	81%	66%

Table 5.29: BMM1 versus ABMM8

	BMM1			ABMM8		
Game	Wins	South	North	Wins	South	North
Awari	3%	3%	3%	94.5%	96%	93%
Oware	13%	16%	10%	84.5%	87%	82%
Vai Lung Thlan	1%	1%	1%	98.5%	99%	98%
Ohvalhu	28.5%	31%	26%	69%	73%	65%
Kalah	17.5%	23%	12%	78%	84%	72%

Table 5.30: BMM4 versus ABMM1

	BMM4			ABMM1		
Game	Wins	South	North	Wins	South	North
Awari	85.5%	89%	82%	9.5%	13%	6%
Oware	77.5%	81%	74%	18%	22%	14%
Vai Lung Thlan	95%	95%	95%	2.5%	3%	2%
Ohvalhu	64%	77%	51%	32%	45%	19%
Kalah	76%	83%	69%	16.5%	23%	10%

Table 5.31: BMM4 versus ABMM4

	BMM4			ABMM4		
Game	Wins	South	North	Wins	South	North
Awari	48.5%	56%	41%	46.5%	55%	38%
Oware	50%	54%	46%	45%	50%	40%
Vai Lung Thlan	47%	49%	45%	50%	51%	49%
Ohvalhu	53%	64%	42%	45%	57%	33%
Kalah	45.5%	49%	42%	48%	50%	46%

Table 5.32: BMM4 versus ABMM8

	BMM4			ABMM8		
Game	Wins	South	North	Wins	South	North
Awari	12%	12%	12%	86.5%	86%	87%
Oware	24%	29%	19%	75%	80%	70%
Vai Lung Thlan	5.5%	3%	8%	93.5%	90%	97%
Ohvalhu	43%	55%	31%	54.5%	67%	42%
Kalah	36.5%	43%	30%	57%	66%	48%

Table 5.33: Time saved with alpha beta pruning in BMM4 versus ABMM4. The values reported are the average time the algorithms take for a game.

Game	BMM4 time	ABMM4 time	Time Saved
Awari	0.1084	0.0509	53.049%
Oware	0.08	0.0374	53.2558%
Vai Lung Thlan	0.2082	0.101	51.479%
Ohvalhu	0.1387	0.0456	67.14%
Kalah	0.0252	0.02	20.834%

Table 5.34: BMM8 versus ABMM1

	BMM8			ABMM1		
Game	Wins	South	North	Wins	South	North
Awari	92.5%	88%	97%	6.5%	3%	10%
Oware	83.5%	87%	80%	14.5%	18%	11%
Vai Lung Thlan	97%	95%	99%	3%	1%	5%
Ohvalhu	71.5%	71%	72%	26.5%	26%	27%
Kalah	80.5%	84%	77%	18%	21%	15%

Table 5.35: BMM8 versus ABMM4

	BMM8			ABMM4		
Game	Wins	South	North	Wins	South	North
Awari	85.5%	92%	79%	13%	18%	8%
Oware	74%	74%	74%	23%	25%	21%
Vai Lung Thlan	92.5%	93%	92%	6%	6%	6%
Ohvalhu	59%	65%	53%	38.5%	46%	31%
Kalah	58.5%	71%	46%	37.5%	49%	26%

Table 5.36: BMM8 versus ABMM8

	BMM8			ABMM8		
Game	Wins	South	North	Wins	South	North
Awari	45.5%	43%	48%	50%	48%	52%
Oware	48.5%	55%	42%	49%	55%	43%
Vai Lung Thlan	50.5%	49%	52%	44%	44%	44%
Ohvalhu	48.5%	56%	41%	47%	55%	39%
Kalah	53%	61%	45%	40.5%	51%	30%

Table 5.37: Time saved with alpha beta pruning in BMM8 versus ABMM8. The values reported are the average time the algorithms take for a game.

Game	BMM4 time	ABMM4 time	Time Saved
Awari	78.35	3.63	95.367%
Oware	25.74	1.75	93.2%
Vai Lung Thlan	160.85	9.2	94.28%
Ohvalhu	119.81	3.54	97.044%
Kalah	9.51	1.14	88.013%

5.5 Advanced Heuristic Minimax

In the next experiment, we wanted to evaluate the performance of the advanced heuristic minimax algorithm (HMM) we developed (see Section 4.4). We decided three depths for the tree search: 1, 4 and 8 (denoted as HMM1, HMM4, HMM8). Similarly to the previous experiments, these three depths represent a beginner player, an intermediate player and an expert player. Then, we played 200 games for each of the mancala games we developed to compare the performance of HMM against the previous algorithms and against itself. In 100 of these games HMM was the south player and in the remaining 100 it was the north player.

When HMM plays against the random player (Tables 5.38 and 5.39), HMM performs better than BMM and ABMM, providing an initial hint on the robustness of this algorithm. For example, when playing Kalah against the random player ABMM1 wins 94.5% (Table 5.16), while HMM1 wins 99.5% (Table 5.38). In our test, HMM8 won always against the random player.

To be consistent with previous experiments, we set that the first 10 moves of each player are randomly chosen when HMM is facing a deterministic opponent. When HMM is against the greedy strategy, it outperforms BMM and ABMM, winning an higher % of games. Note that HMM1 performs better than its opponent against the greedy player also in Kalah and Ohvalhu (Table 5.40). The greedy algorithm has a mechanism that allows it to exploit extra turns, giving it an edge in these two games. Still, it is not enough to obtain a winrate higher than 50% against HMM1, as it was against BMM and ABMM (Tables 5.7 and 5.18). However, we note that with higher depths HMM improves its winrates against the greedy player in Awari, Oware and Vai Lung Thlan, but not quite as much in Ohvalhu and Kalah (Table 5.42. This is coherent with our previous results: we have already observed that deeper tree search results in a smaller advantage in these two games.

When HMM is facing BMM and ABMM (from Table 5.43 to Table 5.60), HMM is clearly superior: it has better performance in every game against BMM and ABMM, when both the algorithms that are playing have the same depth. As expected, thanks to the method used to obtain its heuristic function, HMM is particularly strong in Awari, Oware and Vai Lung Thlan. Surprisingly, in some of the tests involving the games Awari and Oware, HMM obtains a higher winrate than its opponent against BMM and ABMM, despite using a lower depth (Tables 5.44, 5.48, 5.53 and 5.57). This fact points out that a good heuristic function can be better than an higher depth.

When HMM plays against itself (from Table 5.61 to Table 5.66), we note that generally the HMM player with higher depth has an higher winrate, however the difference in winrates between the players is higher in some games more than in others. For example when HMM4 is facing HMM8 (Table 5.65), the difference of winrates in favour of HMM8 is 55.5% in Awari, while it is only 6.5% in Ohvalhu, because in this game deeper search is not as much beneficial. When both HMM players have the same depth, we note that the advantage of the south player is not clear as it was in ABMM versus ABMM and BMM versus BMM. While Awari, Ohvalhu and Kalah favour the south player, in Oware and Vai Lung Thlan it seems unclear which player has an advantage on the other. Still, the south player should be favoured, maybe only by a small margin. In fact, with the exception of Ohvalhu, as the depth increases, the difference in winrates of the players shrinks. This might mean that when the players are stronger, the advantage of the south player is lower.

Table 5.38: HMM1 versus random

	HMM1			random		
Game	Wins	South	North	Wins	South	North
Awari	100%	100%	100%	0%	0%	0%
Oware	100%	100%	100%	0%	0%	0%
Vai Lung Thlan	98%	98%	98%	2%	2%	2%
Ohvalhu	100%	100%	100%	0%	0%	0%
Kalah	99.5%	100%	99%	0.5%	1%	0%

Table 5.39: HMM4 versus random

	HMM4			random		
Game	Wins	South	North	Wins	South	North
Awari	100%	100%	100%	0%	0%	0%
Oware	100%	100%	100%	0%	0%	0%
Vai Lung Thlan	98.5%	98%	99%	1.5%	1%	2%
Ohvalhu	100%	100%	100%	0%	0%	0%
Kalah	100%	100%	100%	0%	0%	0%

Table 5.40: HMM1 versus greedy

	HMM1			greedy		
Game	Wins	South	North	Wins	South	North
Awari	86%	91%	81%	13.5%	18%	9%
Oware	79%	83%	75%	18.5%	20%	17%
Vai Lung Thlan	98.5%	99%	98%	1%	1%	1%
Ohvalhu	53.5%	60%	47%	45%	53%	37%
Kalah	64%	66%	62%	30%	31%	29%

Table 5.41: HMM4 versus greedy

	HMM4			greedy		
Game	Wins	South	North	Wins	South	North
Awari	90.5%	92%	89%	9%	11%	7%
Oware	86%	90%	82%	13%	17%	9%
Vai Lung Thlan	99.5%	100%	99%	0.5%	1%	0%
Ohvalhu	60%	68%	52%	39%	46%	32%
Kalah	69.5%	72%	67%	23.5%	29%	18%

Table 5.42: HMM8 versus greedy

	HMM8			greedy		
Game	Wins	South	North	Wins	South	North
Awari	95.5%	96%	95%	4.5%	5%	4%
Oware	88.5%	90%	87%	11%	13%	9%
Vai Lung Thlan	100%	100%	100%	0%	0%	0%
Ohvalhu	60.5%	67%	54%	38.5%	45%	32%
Kalah	71.5%	82%	61%	24%	34%	14%

Table 5.43: HMM1 versus BMM1

	HMM1			BMM1		
Game	Wins	South	North	Wins	South	North
Awari	85%	86%	84%	12.5%	13%	12%
Oware	79%	82%	76%	19.5%	22%	17%
Vai Lung Thlan	95.5%	98%	93%	4.5%	7%	2%
Ohvalhu	69%	72%	66%	28.5%	31%	26%
Kalah	73.5%	73%	74%	21.5%	24%	19%

Table 5.44: HMM1 versus BMM4

	HMM1			BMM4		
Game	Wins	South	North	Wins	South	North
Awari	56%	55%	57%	42.5%	42%	43%
Oware	57%	61%	53%	42%	46%	38%
Vai Lung Thlan	38%	34%	42%	58.5%	56%	61%
Ohvalhu	49%	60%	38%	47%	57%	37%
Kalah	45%	50%	40%	49.5%	53%	46%

Table 5.45: HMM1 versus BMM8

	HMM1			BMM8		
Game	Wins	South	North	Wins	South	North
Awari	28%	30%	26%	72%	74%	70%
Oware	38%	40%	36%	59%	61%	57%
Vai Lung Thlan	10.5%	11%	10%	86%	85%	87%
Ohvalhu	35.5%	42%	29%	63%	71%	55%
Kalah	29.5%	32%	27%	63.5%	66%	61%

Table 5.46: HMM4 versus BMM1

	HMM4			BMM1		
Game	Wins	South	North	Wins	South	North
Awari	95%	97%	93%	5%	7%	3%
Oware	88%	87%	89%	11.5%	11%	12%
Vai Lung Thlan	100%	100%	100%	0%	0%	0%
Ohvalhu	70.5%	78%	63%	28%	34%	22%
Kalah	78.5%	83%	74%	17%	18%	16%

Table 5.47: HMM4 versus BMM4

	HMM4			BMM4		
Game	Wins	South	North	Wins	South	North
Awari	86.5%	89%	84%	12%	14%	10%
Oware	72%	80%	64%	26.5%	35%	18%
Vai Lung Thlan	78.5%	81%	76%	17%	20%	14%
Ohvalhu	52%	61%	43%	46%	55%	37%
Kalah	61%	67%	55%	32.5%	39%	26%

Table 5.48: HMM4 versus BMM8

	HMM4			BMM8		
Game	Wins	South	North	Wins	South	North
Awari	56.5%	56%	57%	41.5%	41%	42%
Oware	59%	65%	53%	39%	44%	34%
Vai Lung Thlan	35%	39%	31%	57%	60%	54%
Ohvalhu	46.5%	48%	45%	49%	53%	45%
Kalah	43.5%	56%	31%	48.5%	61%	36%

Table 5.49: HMM8 versus BMM1

	HMM8			BMM1		
Game	Wins	South	North	Wins	South	North
Awari	95.5%	98%	93%	4.5%	7%	2%
Oware	91%	94%	88%	8%	12%	4%
Vai Lung Thlan	100%	100%	100%	0%	0%	0%
Ohvalhu	63.5%	70%	57%	33%	41%	25%
Kalah	83.5%	88%	79%	15.5%	20%	11%

Table 5.50: HMM8 versus BMM4

	HMM8			BMM4		
Game	Wins	South	North	Wins	South	North
Awari	96%	96%	96%	3.5%	4%	3%
Oware	85.5%	86%	85%	12.5%	12%	13%
Vai Lung Thlan	93%	94%	92%	4%	5%	3%
Ohvalhu	53.5%	67%	40%	45.5%	58%	33%
Kalah	64.5%	69%	60%	31%	36%	26%

Table 5.51: HMM8 versus BMM8

	HMM8			BMM8		
Game	Wins	South	North	Wins	South	North
Awari	87%	87%	87%	11%	12%	10%
Oware	80.5%	82%	79%	18%	19%	17%
Vai Lung Thlan	86%	87%	85%	10%	10%	10%
Ohvalhu	52.5%	62%	43%	47%	56%	38%
Kalah	61%	66%	56%	34%	39%	29%

Table 5.52: HMM1 versus ABMM1

	HMM1			ABMM1		
Game	Wins	South	North	Wins	South	North
Awari	87.5%	86%	89%	10.5%	8%	13%
Oware	81.5%	83%	80%	18%	19%	17%
Vai Lung Thlan	96.5%	98%	95%	1.5%	3%	0%
Ohvalhu	66%	70%	62%	30.5%	33%	28%
Kalah	73%	78%	68%	20%	22%	18%

Table 5.53: HMM1 versus ABMM4

	HMM1			ABMM4		
Game	Wins	South	North	Wins	South	North
Awari	58.5%	63%	54%	38.5%	44%	33%
Oware	53%	53%	53%	43.5%	43%	44%
Vai Lung Thlan	35.5%	38%	33%	55.5%	57%	54%
Ohvalhu	49%	54%	44%	49.5%	56%	43%
Kalah	44%	54%	34%	51%	63%	39%

Table 5.54: HMM1 versus ABMM8

	HMM1			ABMM8		
Game	Wins	South	North	Wins	South	North
Awari	33%	34%	32%	61.5%	65%	58%
Oware	39%	40%	38%	57.5%	57%	58%
Vai Lung Thlan	15.5%	13%	18%	82.5%	80%	85%
Ohvalhu	45%	60%	30%	52.5%	68%	37%
Kalah	22.5%	33%	12%	71%	82%	60%

Table 5.55: HMM4 versus ABMM1

	HMM4			ABMM1		
Game	Wins	South	North	Wins	South	North
Awari	91.5%	90%	93%	6.5%	7%	6%
Oware	87%	85%	89%	12%	11%	13%
Vai Lung Thlan	100%	100%	100%	0%	0%	0%
Ohvalhu	69%	80%	58%	28%	38%	18%
Kalah	79.5%	85%	74%	17.5%	20%	15%

Table 5.56: HMM4 versus ABMM4

	HMM4			ABMM4		
Game	Wins	South	North	Wins	South	North
Awari	88%	85%	91%	10.5%	6%	15%
Oware	79%	84%	74%	19.5%	23%	16%
Vai Lung Thlan	68%	67%	69%	25.5%	24%	27%
Ohvalhu	56.5%	61%	52%	42%	46%	38%
Kalah	59.5%	71%	48%	33%	44%	22%

Table 5.57: HMM4 versus ABMM8

	HMM4			ABMM8		
Game	Wins	South	North	Wins	South	North
Awari	54.5%	57%	52%	42.5%	47%	38%
Oware	57%	67%	47%	39%	50%	28%
Vai Lung Thlan	34%	34%	34%	59%	56%	62%
Ohvalhu	45%	55%	35%	53%	63%	43%
Kalah	46.5%	59%	34%	49%	62%	36%

Table 5.58: HMM8 versus ABMM1

	HMM8			ABMM1		
Game	Wins	South	North	Wins	South	North
Awari	96.5%	99%	94%	3.5%	6%	1%
Oware	87%	92%	82%	9.5%	13%	6%
Vai Lung Thlan	100%	100%	100%	0%	0%	0%
Ohvalhu	70.5%	76%	65%	28.5%	33%	24%
Kalah	82.5%	87%	78%	15.5%	20%	11%

Table 5.59: HMM8 versus ABMM4

	HMM8			ABMM4		
Game	Wins	South	North	Wins	South	North
Awari	92.5%	94%	91%	6.5%	8%	5%
Oware	86%	88%	84%	12.5%	14%	11%
Vai Lung Thlan	97.5%	97%	98%	1%	1%	1%
Ohvalhu	55%	61%	49%	43.5%	50%	37%
Kalah	64.5%	71%	58%	28%	33%	23%

Table 5.60: HMM8 versus ABMM8

	HMM8			ABMM8		
Game	Wins	South	North	Wins	South	North
Awari	83%	86%	80%	14.5%	17%	12%
Oware	78%	78%	78%	21%	22%	20%
Vai Lung Thlan	81%	81%	81%	11.5%	10%	13%
Ohvalhu	53%	61%	45%	44%	52%	36%
Kalah	49.5%	54%	45%	44%	47%	41%

Table 5.61: HMM1 versus HMM1

Game	South	North	Ties
Awari	54.5%	39.5%	6%
Oware	52.5%	46.5%	1%
Vai Lung Thlan	46%	47%	7%
Ohvalhu	62%	36.5%	1.5%
Kalah	45.5%	50%	4.5%

Table 5.62: HMM1 versus HMM4

	HMM1			HMM4		
Game	Wins	South	North	Wins	South	North
Awari	24%	22%	26%	75%	73%	77%
Oware	27%	34%	20%	70%	78%	62%
Vai Lung Thlan	6.5%	6%	7%	90.5%	90%	91%
Ohvalhu	48.5%	58%	39%	48%	56%	40%
Kalah	31.5%	45%	18%	63.5%	74%	53%

Table 5.63: HMM1 versus HMM8

	HMM1			HMM8		
Game	Wins	South	North	Wins	South	North
Awari	8%	10%	6%	90%	94%	86%
Oware	20.5%	22%	19%	78%	80%	76%
Vai Lung Thlan	1%	0%	2%	98%	97%	99%
Ohvalhu	37%	44%	30%	60.5%	66%	55%
Kalah	25.5%	33%	18%	71%	79%	63%

Table 5.64: HMM4 versus HMM4

Game	South	North	Ties
Awari	54%	41.5%	4.5%
Oware	52%	45%	3%
Vai Lung Thlan	49.5%	41%	9.5%
Ohvalhu	55%	43.5%	1.5%
Kalah	60%	34%	6%

Table 5.65: HMM4 versus HMM8

	HMM4			HMM8		
Game	Wins	South	North	Wins	South	North
Awari	20%	25%	15%	75.5%	77%	74%
Oware	36%	40%	32%	60.5%	65%	56%
Vai Lung Thlan	9.5%	11%	8%	81%	80%	82%
Ohvalhu	45.5%	51%	40%	52%	59%	45%
Kalah	36%	37%	35%	59%	59%	59%

Table 5.66: HMM8 versus HMM8

Game	South	North	Ties
Awari	50%	44.5%	5.5%
Oware	45.5%	47.5%	7%
Vai Lung Thlan	41%	43.5%	15.5%
Ohvalhu	60.5%	39%	0.5%
Kalah	50%	43%	7%

5.6 Monte Carlo Tree Search

In the next experiment, we evaluated the performance of the Monte Carlo tree search (MCTS) we developed (see Section 4.5). We decided a set of values of numbers of iterations as computational budget: 10, 100, 250, 500, 1000, 2000, 4000 (denoted as MCTS10, ..., MCTS4000). This wide array of values allows to test how the performances of MCTS change when the number of iterations is modified. The simulation strategy used is random sampling. Then, we played 200 games for each of the mancala games we developed, comparing MCTS against the previous algorithms and against itself. In 100 of these games MCTS was the first one to play (south player) and in the remaining 100 it was second (north player).

When MCTS is facing the random player, we note that even with such a small number of iterations MCTS10 is able to obtain a winrate higher than 79% in every game, (Table 5.68). MCTS100 is close to reach 100% winrate versus the random player in every game (Table 5.69). In our test MCTS500 always won against the random player.

When MCTS is facing the greedy algorithm (from Table 5.71 to Table 5.75), we note that MCTS10 obtains poor results, except in Vai Lung Thlan where it almost has an even winrate against greedy, 46% and 48.5% respectively. This result confirms that greedy has poor performance in Vai Lung Thlan. With higher number of iterations, MCTS increases its winrates in all the games: MCTS100 has higher winrates than the greedy strategy in all the games, except for Ohvalhu. The winrates of MCTS in Ohvalhu and Kalah remain lower, because greedy is able to exploit the extra turn feature of these games.

The plots from Figure 5.1 to Figure 5.10 show the results of MCTS playing against BMM and ABMM. We note that MCTS1000 is close to 100% winrate against BMM1 and ABMM1 and gets closer to winning always with an higher number of iterations. MCTS obtains lower results against BMM and ABMM when they have higher depths. When playing Awari, Oware and Vai Lung Thlan against BMM4 and ABMM4, MCTS wins a considerably high % of games, provided that it uses a number of iterations high enough. For example, MCTS4000 has a winrate over 80% in these games. In the game Kalah, MCTS4000 performs similarly to BMM4 and ABMM4 when MCTS plays against them. We will talk in detail of Ohvalhu in the next subsection. When MCTS plays against BMM8 and ABMM8, MCTS performances are lower: in Oware, MCTS has a winrate higher than 50%; in Awari, Vai Lung Thlan and Kalah MCTS has a lower winrate, between 20% and 50%, in particular in the game Kalah it is a little over 20%.

The plots from Figure 5.11 to Figure 5.15 show the results of MCTS playing against HMM. We note that MCTS performs much worse against HMM in comparison to what it did against BMM and ABMM. When MCTS4000 is playing against HMM1, even with a high number of iterations, MCTS4000 obtains only a winrate $\approx 80\%$ in Awari, Oware, and Kalah; in Vai Lung Thlan the winrate of MCTS4000 is closer to 100%. When MCTS is playing against HMM4 and HMM8, its performances are considerably worse: against HMM8, MCTS4000 obtains a 23.5% winrate in Awari and 31% winrate in Oware. MCTS4000 is better in the game Vai Lung Thlan, where it obtains 52% winrtate against HMM8, but it is much worse in the game Kalah, where it obtains only 12.5% winrate against HMM8. We will talk in detail of Ohvalhu in the next subsection. The unsatisfying results of MCTS against HMM attest the robustness of HMM, that is the best algorithm we designed. Also, HMM uses less computational time than MCTS. For example, in the game Vai Lung Thlan, where MCTS4000 performs slightly better than HMM8, it should be noted that the average time HMM takes for a game is $60.6s$, while MCTS takes $120.3s$. In other games the difference in time spent is higher: for example, when MCTS4000 plays Awari against HMM8, the average time HMM takes for a game is $21.7s$, while MCTS takes $107.8s$.

5.6.1 Ohvalhu

Ohvalhu requires specific considerations. When playing Ohvalhu against minimax opponents, MCTS performs much worse than in the other four mancala games (except when the search depth is 1, that is, when facing BMM1, ABMM1 or HMM1). This is due to the peculiar rules of Ohvalhu that grants player additional moves when the current sowing ends in an occupied pit, thus, allowing long sequences of sowing. In fact, there exist sequences of moves that allow the first player to win the game without giving the chance to the second player to play even one move, abusing the extra turn rule and the multi lap sowing. Thus, in Ohvalhu the first move can have devastating effects and when an algorithm can find a sufficiently long sequence of initial moves can basically win the game against any opponent. The analysis of the played games show that, when BMM4, ABMM4, HMM4 play first, they are able to find sequences of moves that cause multiple sowing thus winning the game in their very first turn by collecting a large number of points. In contrast, MCTS4000 manages only to find a small number of consecutive moves. The ability to chain a lot of consecutive moves is really useful not only during the first turn of play, but also during all the stages of

	Iterations	Wins	South	North	Ties
MCTS	32000	36.5%	36.5%	0%	11%
MCTS	64000	45%	45%	0%	1.5%

Table 5.67: MCTS versus HMM8 using 32000 and 64000 iterations.

the game: MCTS fails at this, while the minimax strategies we developed are especially strong because they are guided by an heuristic function that seems to push them to make these consecutive moves. In particular BMM, ABMM, HMM with depth equal to 4 or higher are playing as perfect players when they are first.

Table 5.67 reports the winning rate of MCTS using 32000 and 64000 iterations playing against HMM8. As can be noted as the number of iterations increases, MCTS can find more effective first moves and thus wins several more games against our best minimax algorithm. Note however that all the victories are as first player (i.e. South player) since in Ohvalhu when facing strong players (like HMM8) the match must be won with the very first move.

Table 5.68: MCTS10 versus random

	MCTS10			random		
Game	Wins	South	North	Wins	South	North
Awari	80.5%	87%	74%	16%	22%	10%
Oware	82.5%	77%	88%	16.5%	12%	21%
Vai Lung Thlan	84%	83%	85%	13%	14%	12%
Ohvalhu	79.5%	82%	77%	18%	22%	14%
Kalah	84%	83%	85%	11.5%	10%	13%

Table 5.69: MCTS100 versus random

	MCTS100			random		
Game	Wins	South	North	Wins	South	North
Awari	99.5%	99%	100%	0%	0%	0%
Oware	100%	100%	100%	0%	0%	0%
Vai Lung Thlan	98.5%	99%	98%	1%	1%	1%
Ohvalhu	100%	100%	100%	0%	0%	0%
Kalah	99.5%	100%	99%	0.5%	1%	0%

Table 5.70: MCTS250 versus random

	MCTS250			random		
Game	Wins	South	North	Wins	South	North
Awari	100%	100%	100%	0%	0%	0%
Oware	100%	100%	100%	0%	0%	0%
Vai Lung Thlan	99%	98%	100%	0.5%	0%	1%
Ohvalhu	99%	100%	98%	1%	2%	0%
Kalah	100%	100%	100%	0%	0%	0%

Table 5.71: MCTS10 versus greedy

	MCTS10			greedy		
Game	Wins	South	North	Wins	South	North
Awari	16.5%	20%	13%	82.5%	85%	80%
Oware	32.5%	32%	33%	62.5%	59%	66%
Vai Lung Thlan	46%	45%	47%	48.5%	46%	51%
Ohvalhu	0.5%	1%	0%	99.5%	100%	99%
Kalah	18.5%	21%	16%	81%	84%	78%

Table 5.72: MCTS100 versus greedy

	MCTS100			greedy		
Game	Wins	South	North	Wins	South	North
Awari	71%	71%	71%	23.5%	22%	25%
Oware	76.5%	75%	78%	19.5%	21%	18%
Vai Lung Thlan	88.5%	90%	87%	11%	13%	9%
Ohvalhu	10%	4%	16%	87.5%	81%	94%
Kalah	56.5%	65%	48%	37%	45%	29%

Table 5.73: MCTS250 versus greedy

	MCTS250			greedy		
Game	Wins	South	North	Wins	South	North
Awari	85.5%	85%	86%	11%	10%	12%
Oware	93.5%	95%	92%	3.5%	5%	2%
Vai Lung Thlan	95.5%	95%	96%	4%	4%	4%
Ohvalhu	30%	19%	41%	63.5%	48%	79%
Kalah	65.5%	72%	59%	26.5%	33%	20%

Table 5.74: MCTS500 versus greedy

	MCTS500			greedy		
Game	Wins	South	North	Wins	South	North
Awari	90.5%	88%	93%	6.5%	6%	7%
Oware	97%	96%	98%	2%	2%	2%
Vai Lung Thlan	97.5%	97%	98%	2%	2%	2%
Ohvalhu	47.5%	39%	56%	42%	26%	58%
Kalah	81%	90%	72%	14%	21%	7%

Table 5.75: MCTS1000 versus greedy

	MCTS1000			greedy		
Game	Wins	South	North	Wins	South	North
Awari	97.5%	98%	97%	1%	0%	2%
Oware	99%	99%	99%	0.5%	1%	0%
Vai Lung Thlan	99%	100%	98%	1%	2%	0%
Ohvalhu	73.5%	74%	73%	18%	14%	22%
Kalah	87.5%	96%	79%	5.5%	10%	1%

Figure 5.1

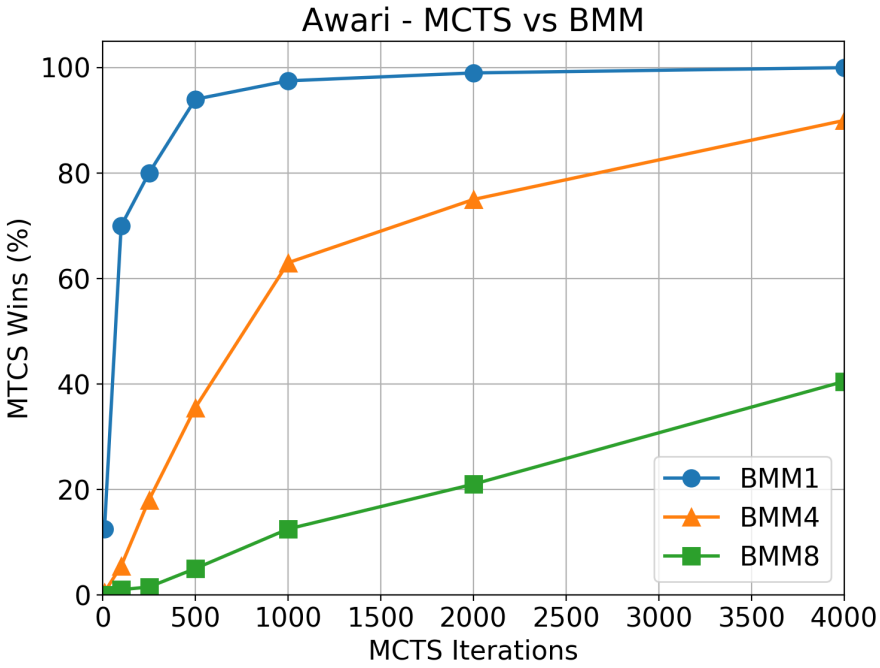


Figure 5.2

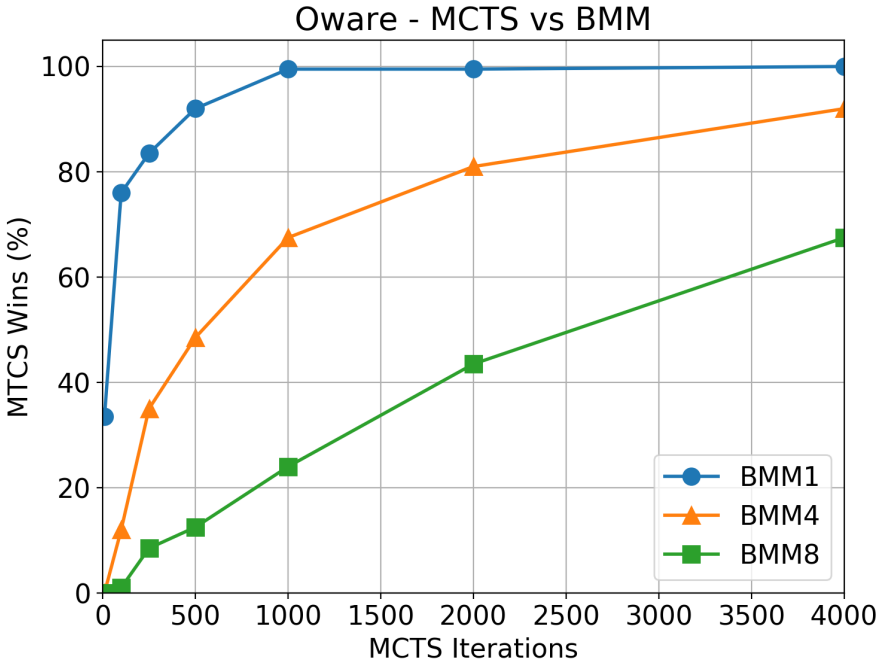


Figure 5.3

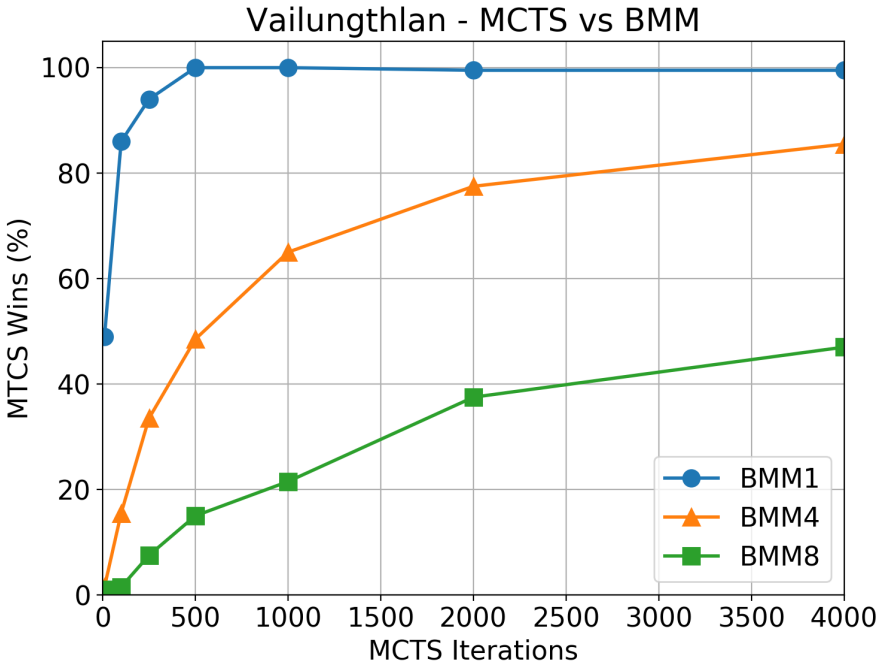


Figure 5.4

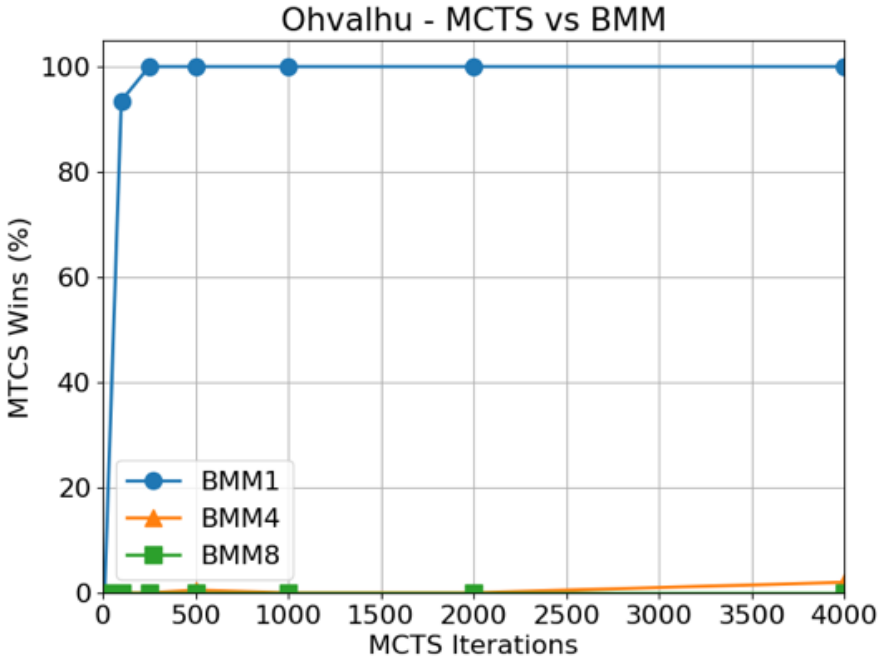


Figure 5.5

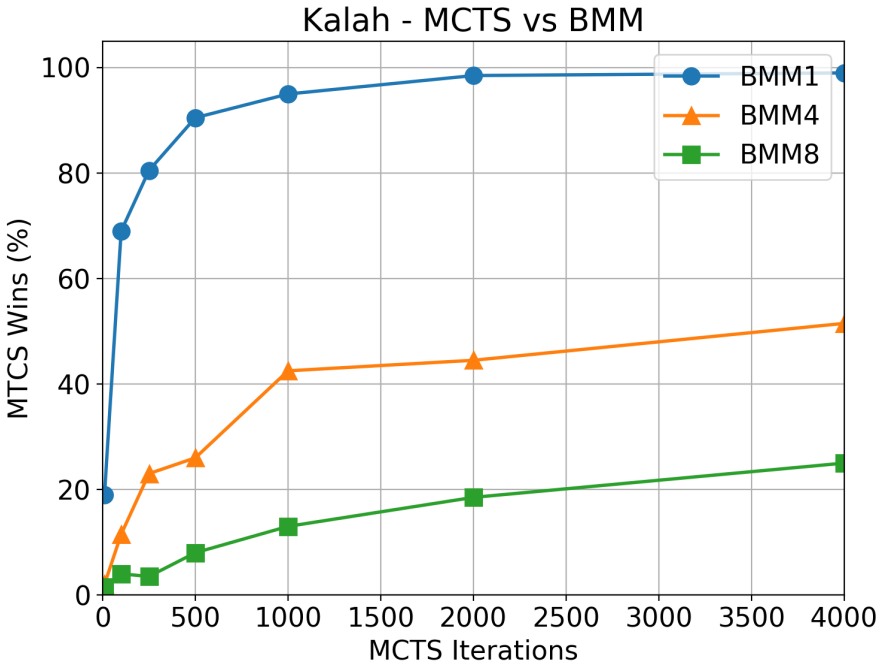


Figure 5.6

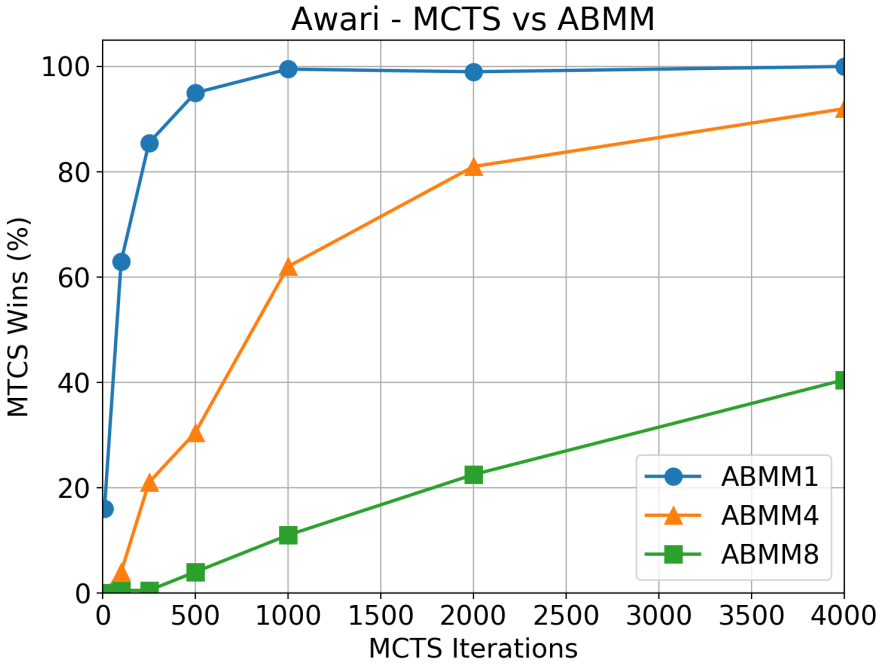


Figure 5.7

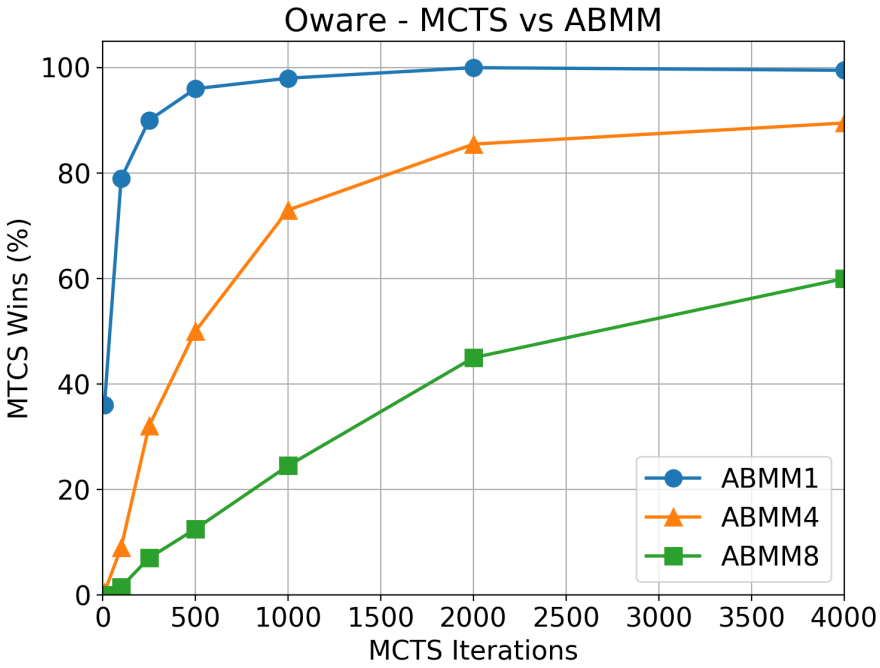


Figure 5.8

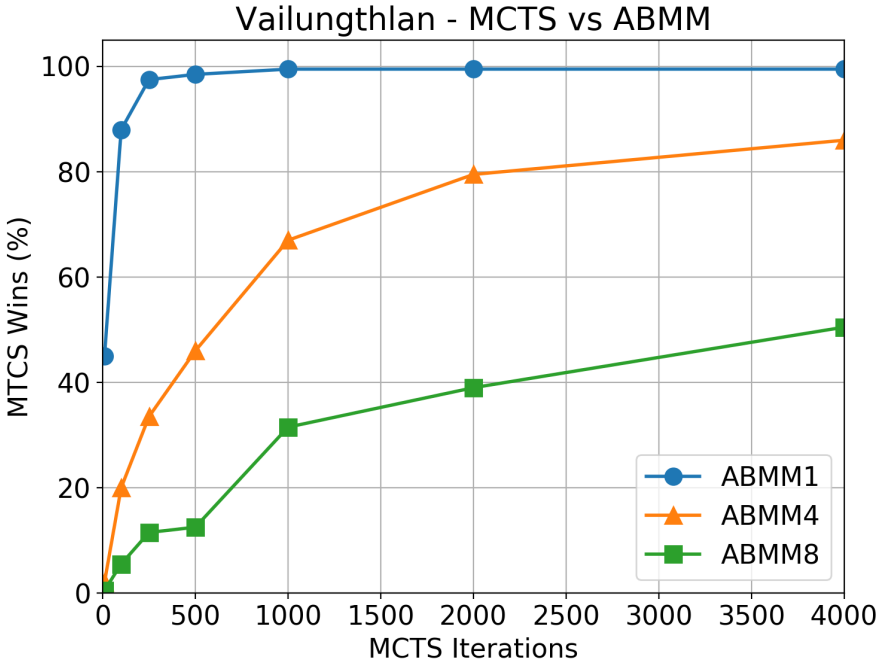


Figure 5.9

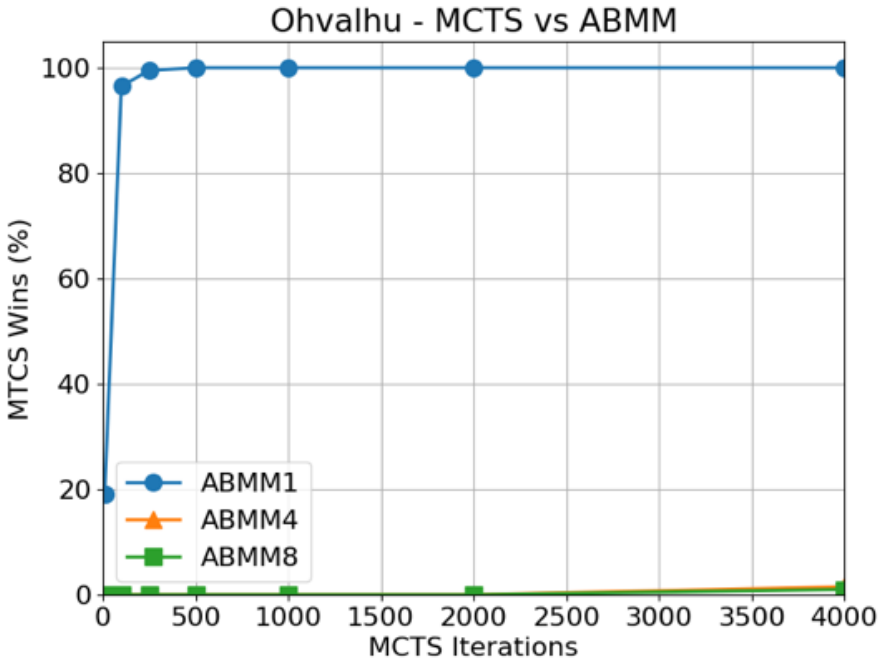


Figure 5.10

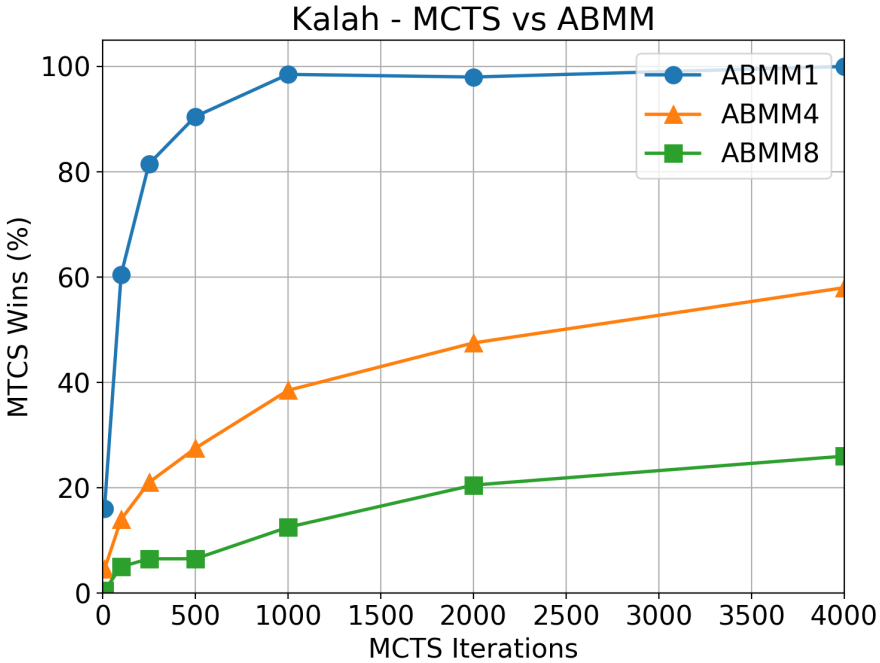


Figure 5.11

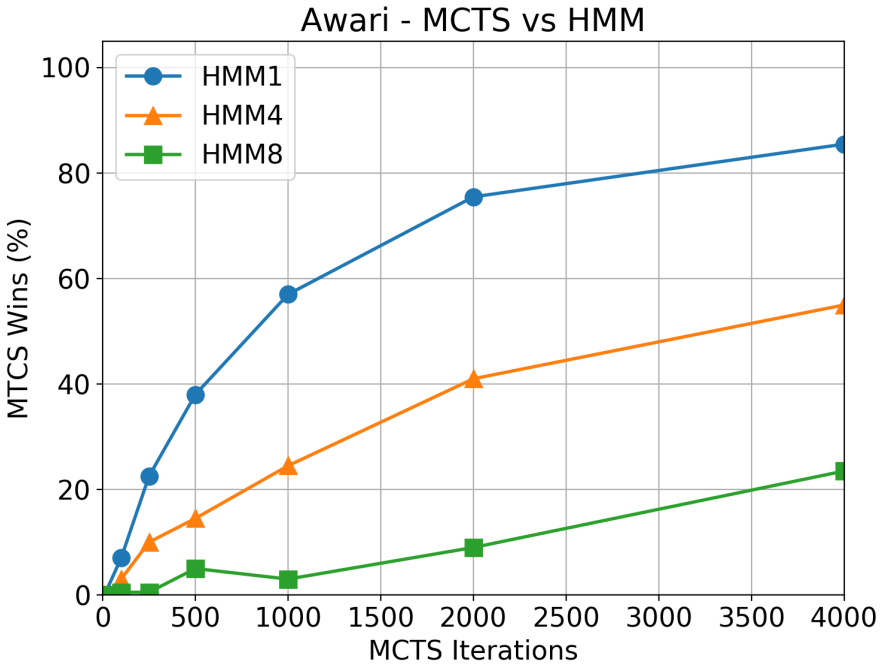


Figure 5.12

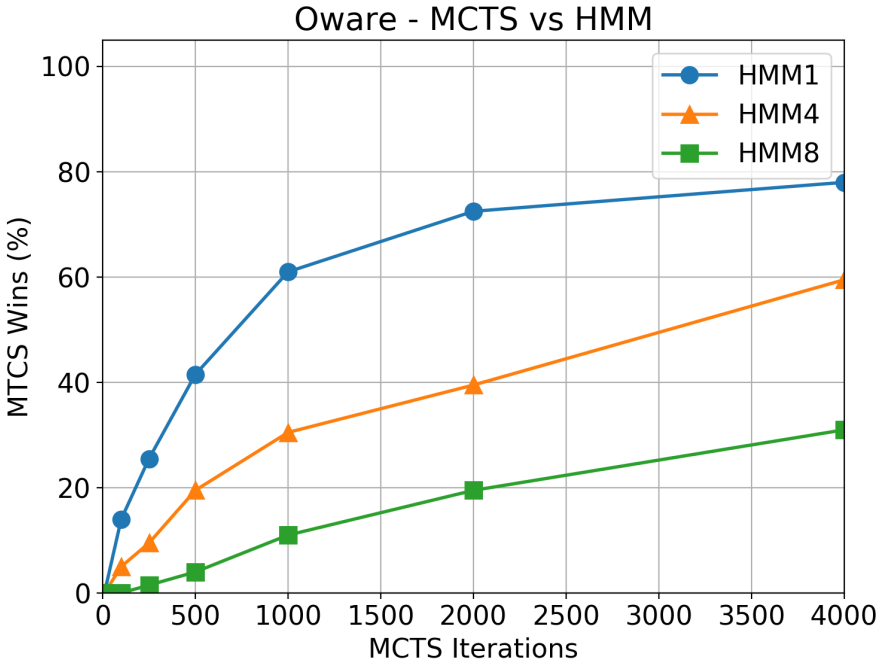


Figure 5.13

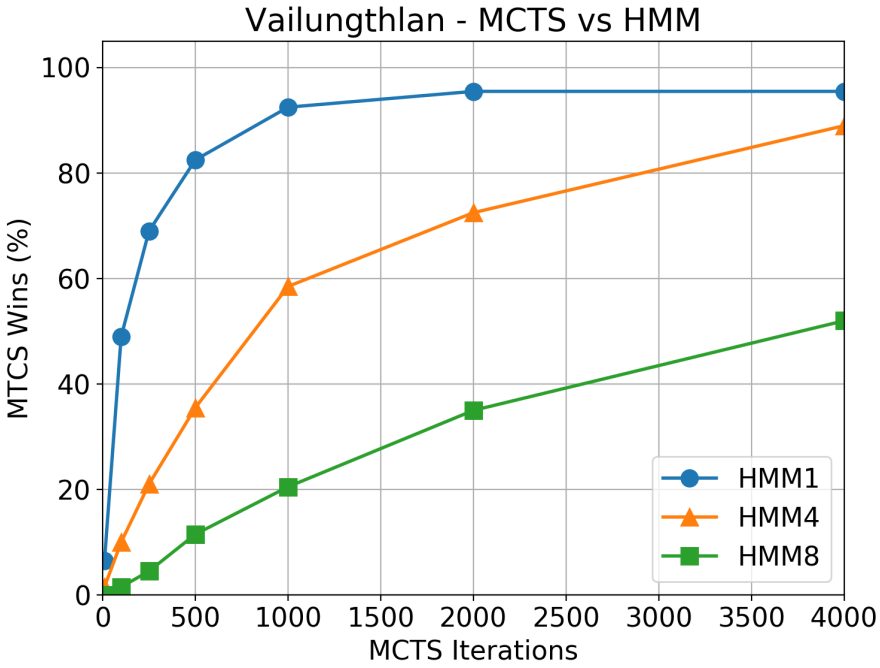


Figure 5.14

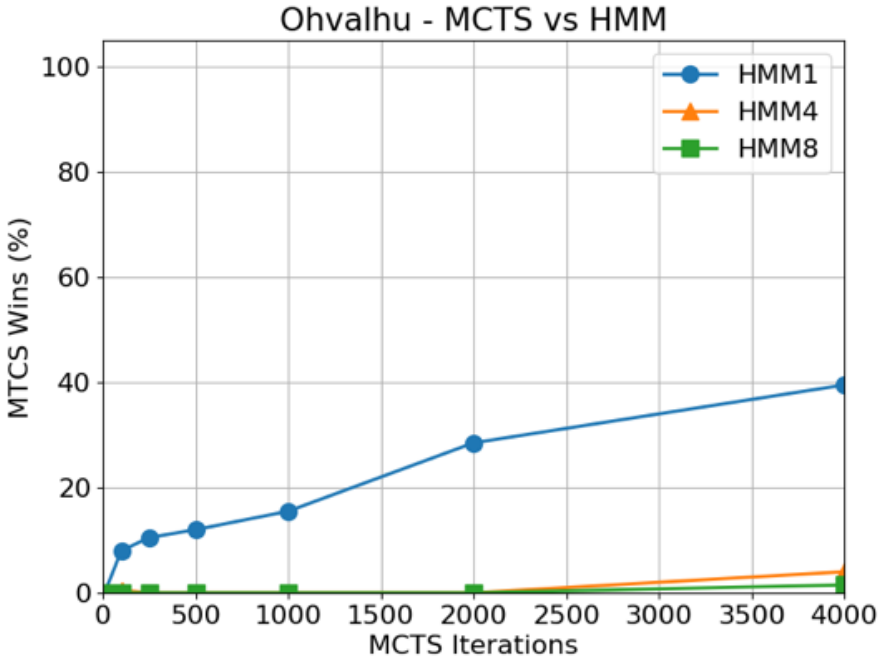


Figure 5.15

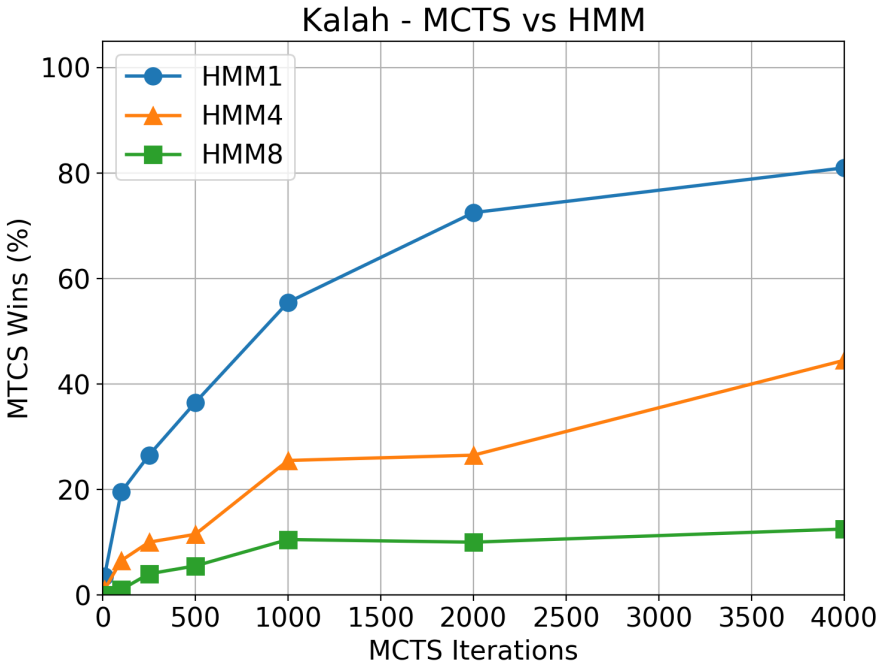


Figure 5.16

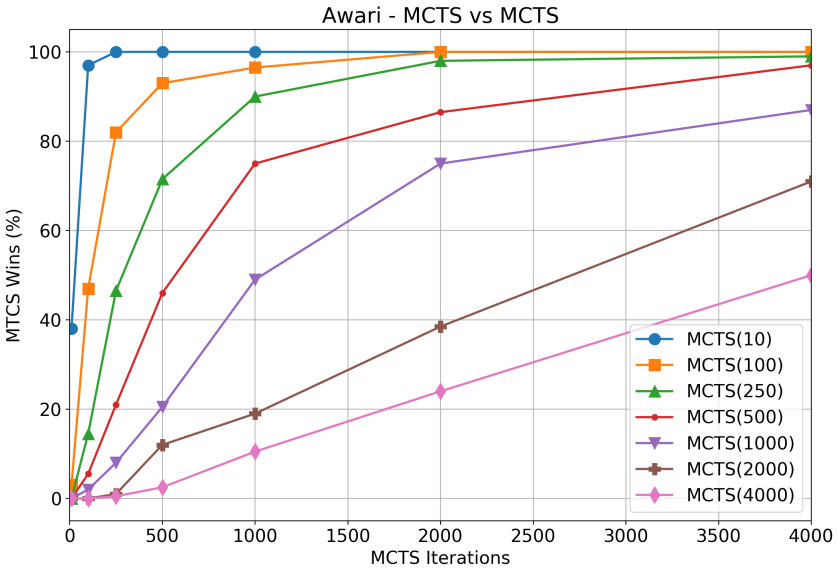


Figure 5.17

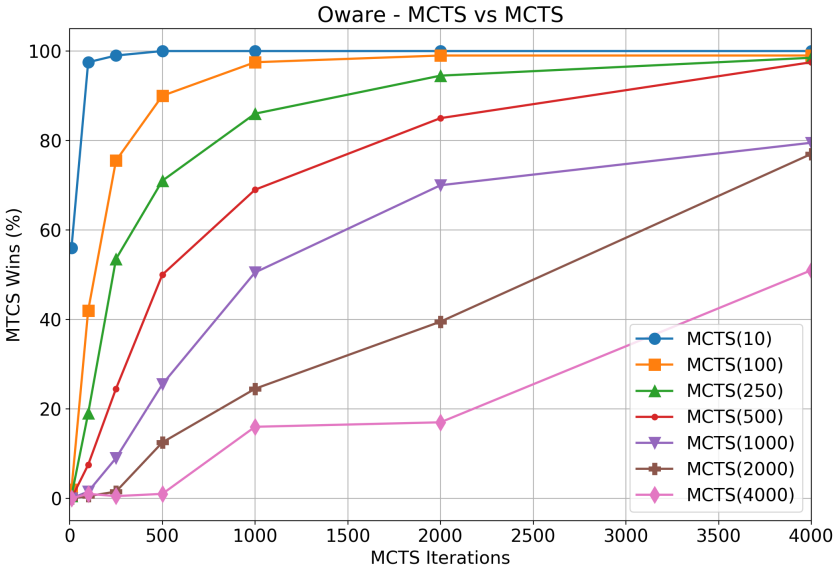


Figure 5.18

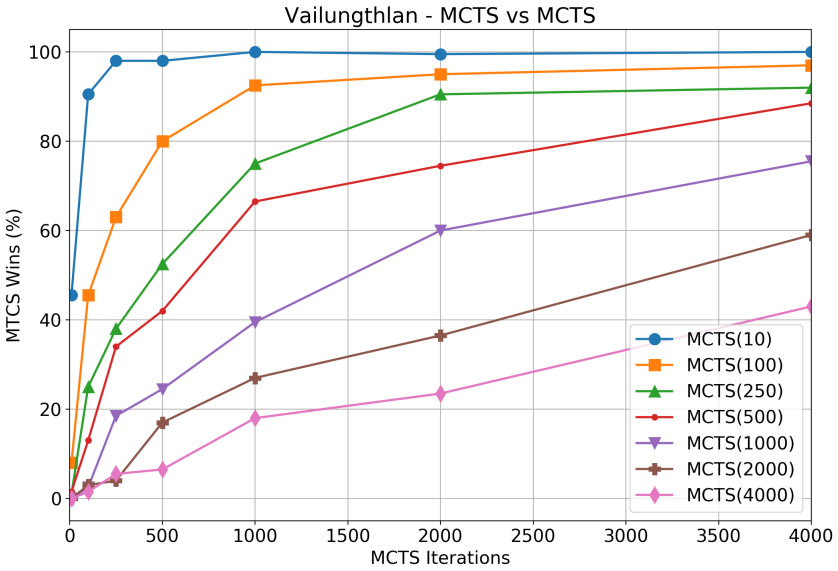


Figure 5.19

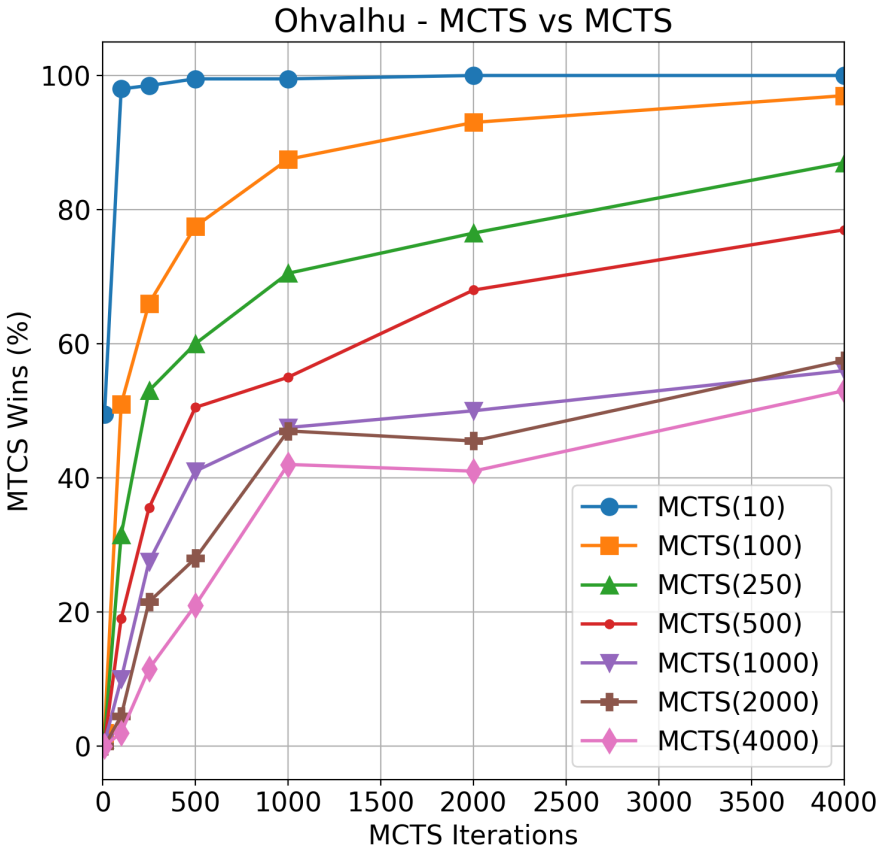
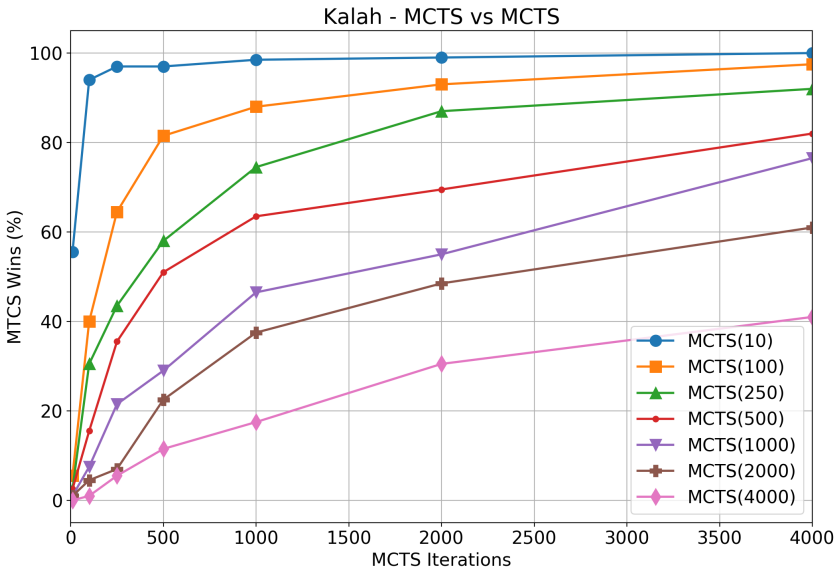

Ohvalhu - MCTS vs MCTS
MTCS Wins (%)
MCTS Iterations
0
20
40
60
80
100
0
500
1000
1500
2000
2500
3000
3500
4000
MCTS(10)
MCTS(100)
MCTS(250)
MCTS(500)
MCTS(1000)
MCTS(2000)
MCTS(4000)

Figure 5.20



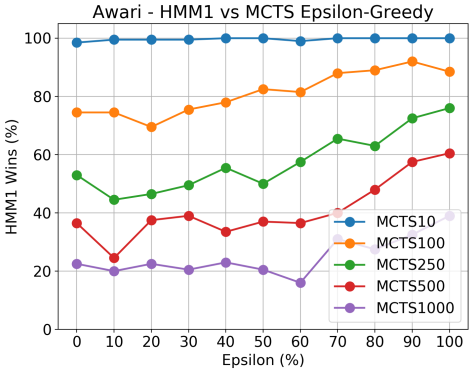
5.6.2 ε-Greedy Strategy

MCTS requires the estimation of the state's value of leaf nodes accordingly and at each node it applies a simulation strategy to play a game from a given state until the end so as to obtain an evaluation of the state value. In the previous experiments, we used the standard simulation strategy that applies a basic random strategy to reach the end state and it is equivalent to playing against a random opponent. Previous studies [10] suggested that using heuristics in the simulation step can increase the performance of the algorithm. Therefore in the final set of experiments we compared the performance of MCTS using ad ε-greedy strategy and different number of iterations (namely, 10, 100, 250, 500, and 1000) against the best performing algorithm we developed, the advanced heuristic minimax (HMM), with a search depth of 1, 4, and 8. The ε-greedy applies with probability ε the usual random strategy and with probability $1 - \varepsilon$ it applies the greedy strategy. Thus, when $\varepsilon = 0$, ε-greedy is equivalent to the greedy strategy employed in the previous experiments, when $\varepsilon = 1$, ε-greedy is equivalent to the plain random strategy.

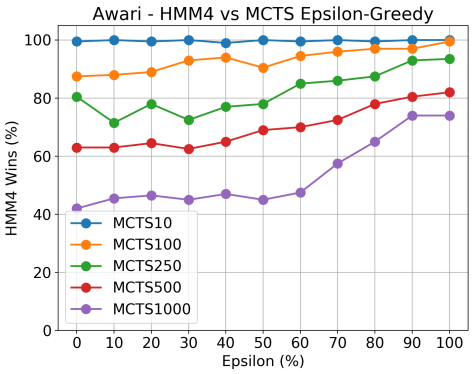
Figure 5.21 shows the winning rate of HMM when playing Awari against of MCTS using an ε-greedy strategy. As the value of ε increases, and the simulation strategy becomes more and more random, HMM winning rate

linearly increases. As already shown in the previous experiments, the higher the number of iterations used by MCTS the lower the win rate of HMM. Overall HMM confirms to be a very strong player. Even with a search depth of 1 (Figure 5.21a) MCTS needs at least 500 iterations and more than 70% of greedy simulated actions to win slightly more than half of the matches. When HMM uses a search depth of 4 (Figure 5.21) MCTS with 1000 iteration needs at least 50% of greedy simulated actions to win slightly more than half of the matches.

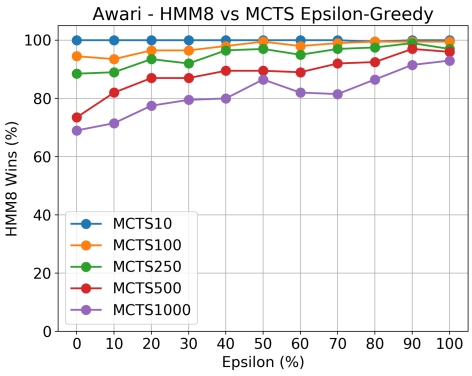
The results for Oware and Kalah (Figure 5.22 and Figure 5.24) do not show the same trend in fact, as the value of ε increase, and the simulation strategy becomes more and more random, HMM winning rate increases only for a depth of 1 (Figure 5.22a and Figure 5.24a) whereas it remains basically stable for a search depth of 4 and 8. Again overall, HMM proves to be a very strong player. MCTS can only win more 50% of the games when HMM search depth is 1 and MCTS employs at least 250 iterations using more than 50% of greedy simulated actions. The same plots for Vai Lung Thlan (Figure 5.23) confirms HMM as a very strong player winning more than 80% of the matches when a depth search of 4 and 8 are used. Interestingly the plot for HMM1 (Figure 5.23a) suggests that in Vai Lung Thlan the ε-greedy simulation strategy might perform worse than plain random simulation strategy. In fact, as the percentage of random simulated actions increases the winning rate of HMM decreases. A similar behavior is noticeable also for HMM4 and HMM8: when the probability of performing a random simulated action increases HMM winning rate decreases; with a depth of 4, HMM winning rate goes slightly below 50% whereas with a depth of 8, although decreasing for values of ε between 30% and 80%, HMM maintains a winning rate way above 50%.



(a)

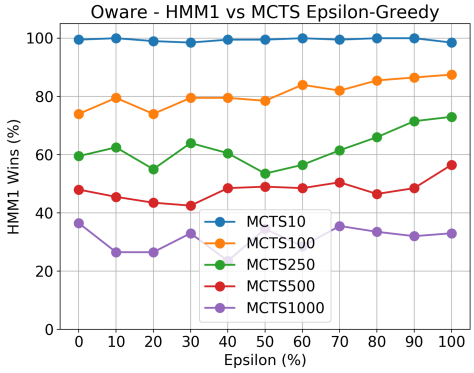


(b)

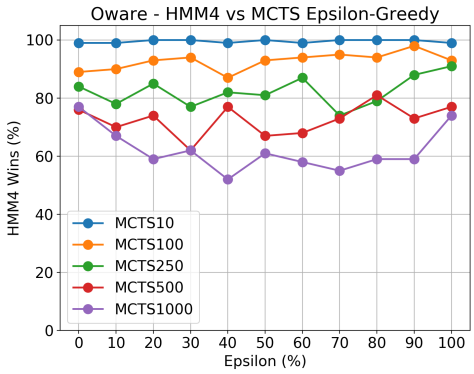


(c)

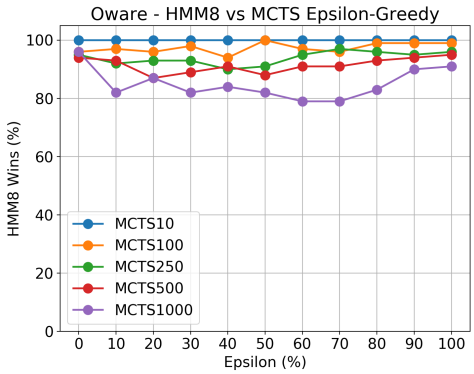
Figure 5.21: Winning rates of the advanced heuristic minimax (HMM) with depth of (a) 1, (b) 2, and (c) 4 when playing Awari against MCTS using the ε-greedy strategy with different number of iterations (10, 100, 250, 500, and 1000) and different values of ε.



(a)

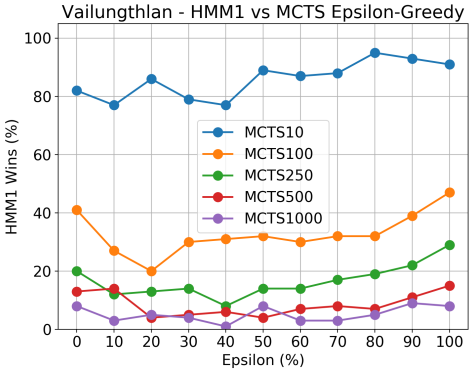


(b)

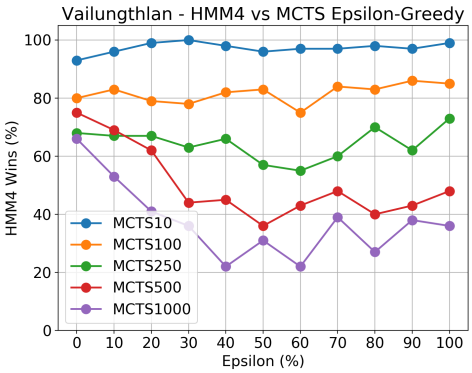


(c)

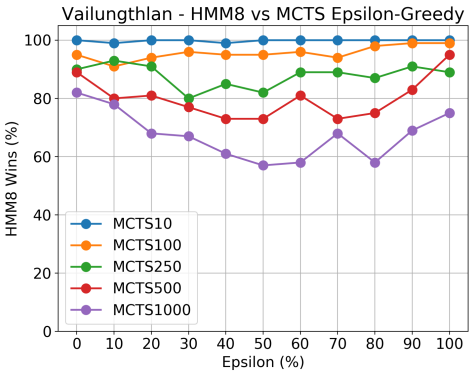
Figure 5.22: Winning rates of the advanced heuristic minimax (HMM) with depth of (a) 1, (b) 2, and (c) 4 when playing Oware against MCTS using the ε-greedy strategy with different number of iterations (10, 100, 250, 500, and 1000) and different values of ε.



(a)

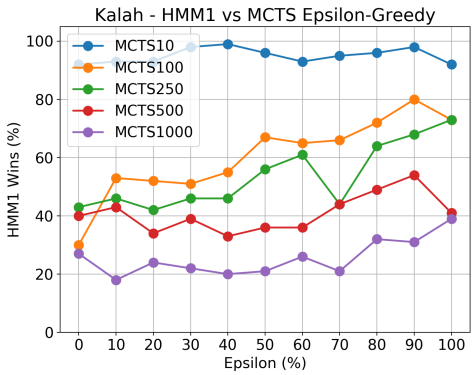


(b)

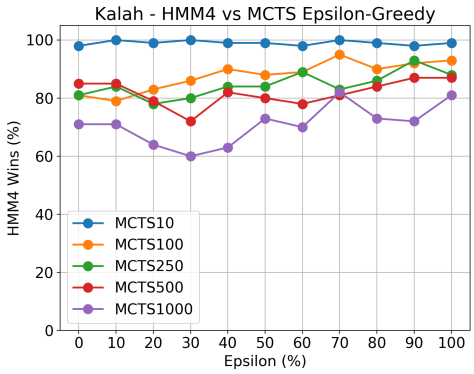


(c)

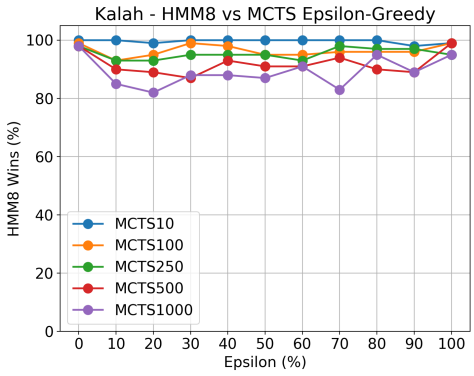
Figure 5.23: Winning rates of the advanced heuristic minimax (HMM) with depth of (a) 1, (b) 2, and (c) 4 when playing Vai Lung Thlan against MCTS using the ε-greedy strategy with different number of iterations (10, 100, 250, 500, and 1000) and different values of ε.



(a)



(b)



(c)

Figure 5.24: Winning rates of the advanced heuristic minimax (HMM) with depth of (a) 1, (b) 2, and (c) 4 when playing Kalah against MCTS using the ε-greedy strategy with different number of iterations (10, 100, 250, 500, and 1000) and different values of ε.

5.7 Summary

In this chapter, we discussed the experiments we performed to evaluate the playing strength of the Artificial Intelligence algorithms we designed. The first experiment showed that, if playing at random, the south player is favoured in Vai Lung Thlan, Ohvalhu and Kalah, while in Awari and Oware the north player is the one favoured. In the second experiment, we tested the greedy algorithm, showing its performance. In the third experiment, we tested the basic minimax algorithm that is much stronger than the greedy strategy when using high enough search depths. The fourth experiment showed that the alpha-beta pruning minimax algorithm has performances comparable to basic minimax, yet it is less computational expensive, especially with high search depths. In the fifth experiment, we tested the advanced heuristic minimax algorithm that turned out to be the best, having robust performance especially in Awari and Oware. In the sixth experiment, we tested the Monte Carlo tree search algorithm: at first we used the random simulation strategy, then we used the ε-greedy strategy. Monte Carlo tree search obtains good results, however it requires a high number of iterations to perform on par with the minimax algorithms, thus resulting in a high computational time. Overall, heuristic minimax proved to be the strongest algorithm developed.

Chapter 6

Conclusions

We designed and evaluated artificial intelligence algorithms for five mancala games, Awari, Oware, Vai Lung Thlan, Ohvalhu and Kalah. We developed five different algorithms. The greedy algorithm is inspired by players guides to Mancala games and applies well-known strategies to capture the most counters in one move; when the move considered allows for an extra turn, the greedy algorithm analyzes in the same way the candidate moves of its next turn. The basic minimax algorithm (BMM) uses the well-known Minimax algorithm to find the best move, guided by an heuristic function that computes the difference between the counters in the player's store and the ones in the opponent's store. The alpha-beta pruning minimax algorithm (ABMM) uses Minimax with the alpha-beta pruning technique to reduce computational time and memory consumption. The advanced heuristic minimax strategy (HMM) uses Minimax with a more refined heuristic function, based on the work of Divilly et al. [11]. The Monte Carlo tree search strategy (MCTS) uses the Upper Confidence Bounds for Trees (UCT) and one of the simulation strategies we implemented: (i) the random strategy; (ii) the greedy strategy presented previously; (iii) the ε-greedy strategy that plays at random with probability ϵ, otherwise it plays the move chosen by the greedy strategy.

We evaluated all the algorithms through a series of experiments. Our results show that the greedy strategy is computational inexpensive and obtains good results in the games Ohvalhu and Kalah. BMM with high search depths performs better but it is, as expected, computational demanding. ABMM performs similarly to BMM and it saves a lot of computational time, especially with higher search depths. HMM is the best algorithm we developed, because it is guided by a robust heuristic function, that works especially well in Awari and Oware. MCTS with a random simulation strat-

egy plays acceptably considering it does not have any knowledge about the game nor it employs any state evaluation function. However MCTS requires a high number of iterations to obtain results on par with the Minimax algorithms, resulting in a high computational time. Finally, we tested MCTS with the ε-greedy simulation strategy. The results show that the best ε value for Awari is 0 (that is a full greedy simulation strategy), while data for other games were not decisive enough to indicate a definitive value of ε. Still, MCTS with a ε-Greedy simulation strategy performs worse than than HMM, which confirms to be the stronger player we developed. As result of this thesis, we developed an application, using the game engine Unity, to play all the five games against all the five artificial intelligence strategies we developed.

Appendix A

The Applications

In this appendix, we show the main features of the two versions of the application we developed for this thesis: one to conduct the experiments, and one to let the users play it.

A.1 The Experiments Framework

In order to conduct the experiments we needed for this work, we developed a console application written in C# with the Microsoft Integrated Development Environment (IDE) Visual Studio 2015. We designed the framework in such a way that it is easy to use it also with other games. For this purpose, we defined two interfaces:

- *IGameState*: it represents the state of a game and includes methods to get the available moves, apply a move, check if it is a terminal state, get the scores fo the game, know the last player to move, clone the state, and get a simulation move.
- *IGameMove*: it represents a move of the game. It does not define any particular method because the moves of different games can be very different. We need only to test if a move is equal to another one, but this is already provided by the Object interface by overriding the Equals method.

Then, we implemented these interfaces with the classes MancalaGameState and MancalaMove. We implemented five subclasses of MancalaGameState, one for each mancala games we developed. The parent class MancalaGameState contains general mancala rules such as how a move work, while the subclasses deal with game specific rules such as captures and when the game is over. The subclasses also contain boolean flags that tell the parent class

the details of how a move works in that game. For example, the boolean extraTurn is set false in Awari because it does not allow extra turns and it is true in Kalah, since players can gain additional turn in this game.

After that, we implemented the different algorithms presented. For example, Source Code A.1 shows the core of advanced heuristic minimax and Source Code A.2 shows the implementation of MCTS.

Source Code A.1: advanced heuristic minimax algorithm implementation

```
1  float Iterate(Node node, int depth, float alpha, float beta, int callingPlayer) {
2      // Leaf node
3      if (depth == 0 || node.IsTerminal()) {
4          return node.GetTotalScore(callingPlayer);
5      }
6      // player playing: maximize
7      if (node.mancalaGameState.currentPlayer == callingPlayer) {
8          float v = -1000;
9          foreach (Node child in node.Children()) {
10             v = Math.Max(v, Iterate(child, depth - 1, alpha, beta, callingPlayer));
11             alpha = Math.Max(alpha, v);
12
13             if (alpha >= beta) {
14                 break;
15             }
16         }
17         return v;
18     } else {
19         // opponent playing: minimize
20         float v = 1000;
21         foreach (Node child in node.Children()) {
22             v = Math.Min(v, Iterate(child, depth - 1, alpha, beta, callingPlayer));
23             beta = Math.Min(beta, v);
24             if (beta <= alpha) {
25                 break;
26             }
27         }
28         return v;
29     }
30 }
```

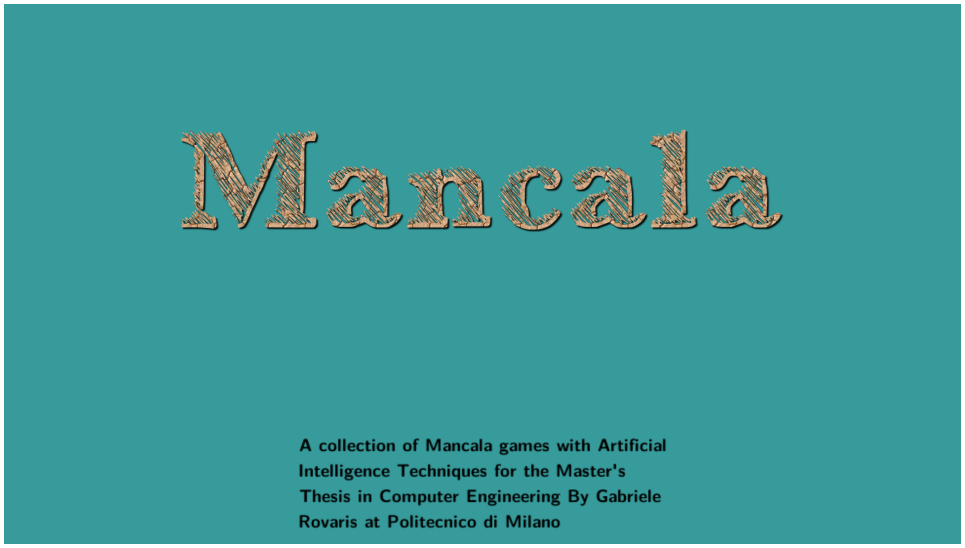
Source Code A.2: Monte Carlo Tree Search algorithm implementation

```
1  IGameMove Search(IGameState rootState, int iterations) {
2      ITreeNode rootNode = treeCreator.GenRootNode(rootState);
3      for (int i = 0; i < iterations; i++) {
4          ITreeNode node = rootNode;
5          IGameState state = rootState.Clone();
6
7          // Select
8          while (!node.HasMovesToTry() && node.HasChildren()) {
9              node = node.SelectChild();
10             state.DoMove(node.Move);
11         }
12
13         // Expand
14         if (node.HasMovesToTry()) {
15             IGameMove move = node.SelectUntriedMove();
16             state.DoMove(move);
17             node = node.AddChild(move, state);
18         }
19
20         // Rollout
21         while (!state.IsTerminal()) {
22             state.DoMove(state.GetSimulationMove());
23         }
24
25         // Backpropagate
26         while (node != null) {
27             node.Update(state.GetResult(node.PlayerWhoJustMoved));
28             node = node.Parent;
29         }
30     }
31
32     return rootNode.GetBestMove();
33 }
```

A.2 The User Application

In order to let the user play with our artificial intelligence, we developed a user application with the cross-platform game creation system Unity. It includes a game engine and the open-source IDE MonoDevelop. The game engine's scripting is built on Mono, the open-source implementation of .NET Framework. It provides a number of libraries written in C# and Javascript in order to manage graphics, physics, sound, network, and inputs of a game. One of the greatest advantages of Unity is that it allows the deployment of the same code on several platforms. In fact, it is able to build an application for Windows, Mac OS X, Linux, iOS, Android, BlackBerry, Windows Phone, web browsers, and game consoles. It allows also to develop a 2D game by means of dedicated libraries and components. For the development of our game we used Unity 5.5 with the C# programming language, so that we can

Figure A.1: The opening screen of the application.



use the same code we created for the console application. Beatrice Danesi, student of fashion design from Politecnico di Milano, helped us with the graphical aspect of the application.

The application provides a menu (Figure A.2), in which the user can choose the game, the players and, in the case of an AI player, the level of difficulty. The menu provides also a small description of the rules of the game currently selected. When the user presses the 'Start' button, the board of the game is presented and the game begins (Figure A.3). To choose a move, the player must click the pit she wants to move from. In the case the player cannot move from that pit, a message 'Invalid Move' appears. When the AI is thinking its move, a rotating gear is shown, indicating that the AI is considering its options. When the game is over, a message declares the winner (Figure A.4). The user can press the 'Esc' button to open a small menu that gives the options to go back to the main menu or to restart the game with the same settings.

Figure A.2: The main menu.

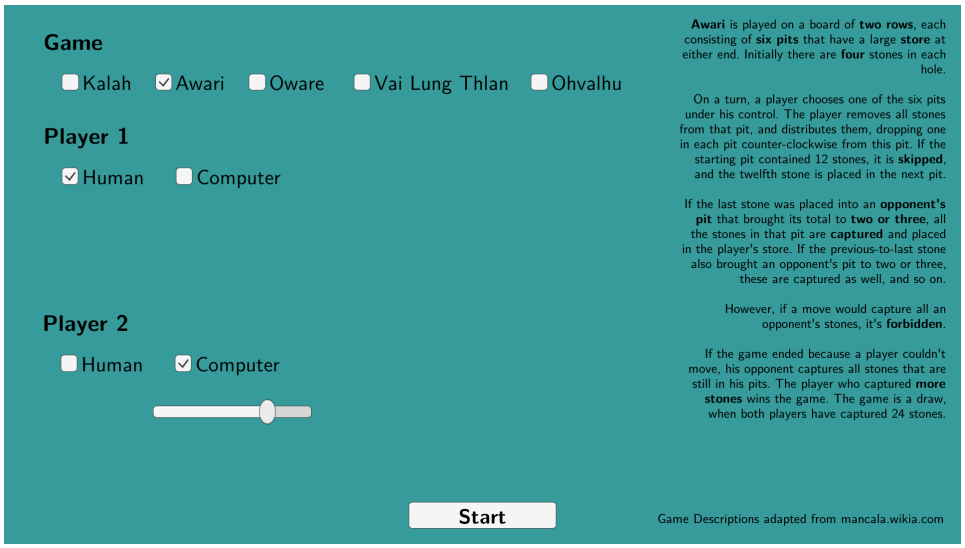


Figure A.3: The game starts.

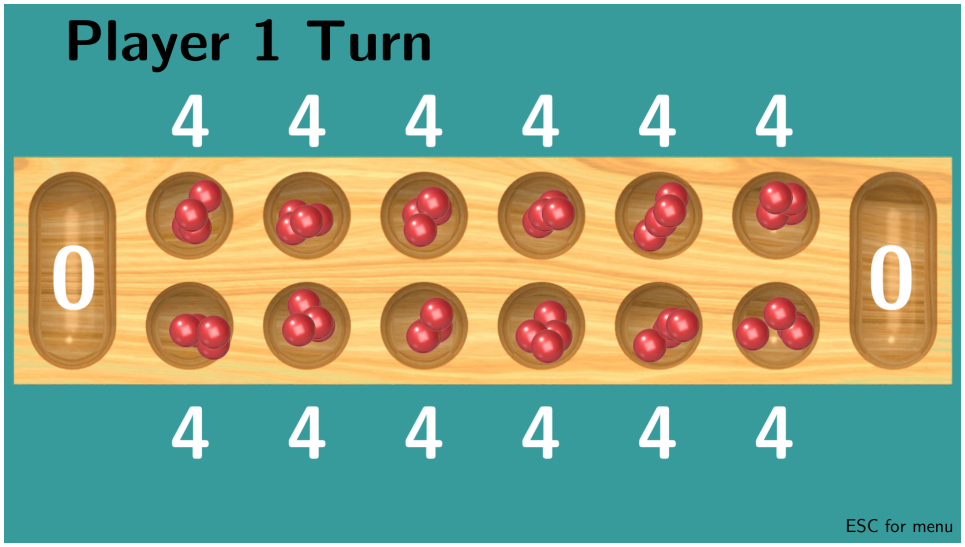
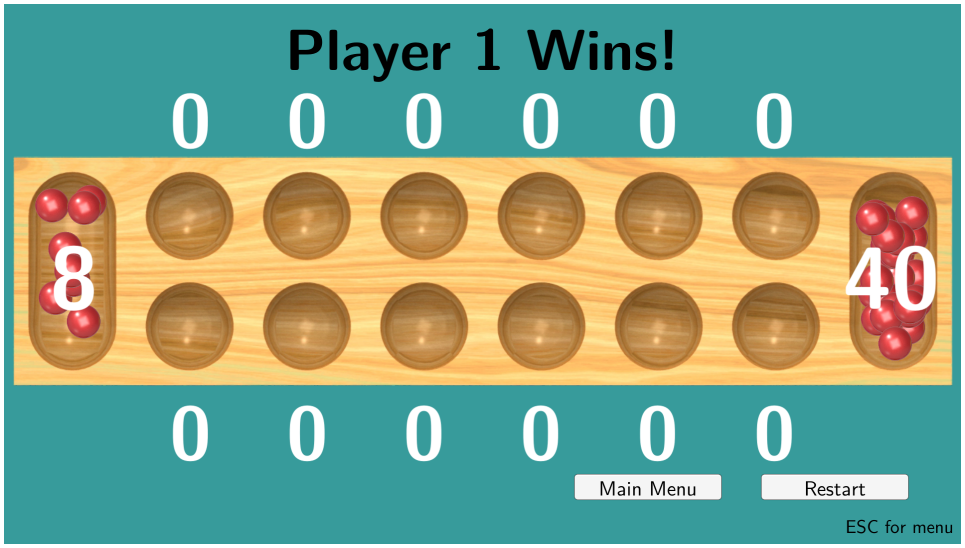


Figure A.4: The end of a game.



Bibliography

[1] Chess programming. http://chessprogramming.wikispaces.com.

[2] History of ai. http://www.alanturing.net/turing_archive/pages/Reference%20Articles/BriefHistofComp.html.

[3] Mancala world. http://mancala.wikia.com.

[4] Broderick Arneson, Ryan B Hayward, and Philip Henderson. Monte carlo tree search in hex. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):251–258, 2010.

[5] Computer Go Group at the University of Alberta. Fuego. http://fuego.sourceforge.net/.

[6] A. G. Bell. *Kalah on Atlas. In D. Mitchie*, volume Machine Intelligence 3. Edinburgh: University Press, 1968.

[7] Bruno Bouzy and Tristan Cazenave. Computer go: an ai oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[8] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[9] UK. Computational Creativity Group (CCG) of the Department of Computing, Imperial College London. Monte carlo tree search. http://mcts.ai/about/index.html.

[10] Peter I Cowling, Colin D Ward, and Edward J Powley. Ensemble determinization in monte carlo tree search for the imperfect information card game magic: The gathering. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(4):241–257, 2012.

[11] Colin Divilly, Colm O'Riordan, and Seamus Hill. Exploration and analysis of the evolution of strategies for mancala variants. *IEEE Conference on Computational Inteligence in Games (CIG)*, 2013.

[12] Jeroen Donkers, Jos Uiterwijk, and Alex de Voogt. Mancala games - topics in mathemathics and artificial intelligence. *The Journal of Machine Learning Research*, 2001.

[13] Neelam Gehlot. Mancala game playing agent. `https://github.com/neelamgehlot/Mancala-Game-Playing-Agent`.

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

[15] Dayo Ajayi Chris Gifford, James Bley, and Zach Thompson. Searching and game playing: An artificial intelligence approach to mancala. Technical report, Information Telecommunication and Technology Center, 2008.

[16] Matthew L Ginsberg. Gib: Imperfect information in a computationally challenging game. *J. Artif. Intell. Res.(JAIR)*, 14:303–358, 2001.

[17] G. Irving, H. H. L. M. Donkers, and J. W. H. M. Uiterwijk. Solving kalah. *ICGA Journal*, 2000.

[18] Damien Jordan and Colm O'Riordan. Evolution and analysis of strategies for mancala games. *GAMEON*, 2011.

[19] HJ Van den Herik LV Allis, M Van der Meulen. Databases in awari. *Heuristic Programming in Artificial Intelligence*, 1991.

[20] Donkers H. H. L. M., Voogt A. J. de, and Uiterwijk J. W. H. M. Human versus machine problem-solving: Winning openings in dakon. *Board Games Studies*, pages 79–88, 2000.

[21] M. v. d. Meulen, L. V. Allis, and H. J. v. d. Herik. Lithidion, an awari-playing program. Technical report, Maastricht: Universiteit Maastricht, 1990.

[22] JAM Nijssen and Mark HM Winands. Monte-carlo tree search for the game of scotland yard. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 158–165. IEEE, 2011.

[23] Marc JV Ponsen, Geert Gerritsen, and Guillaume Chaslot. Integrating opponent models with monte-carlo tree search in poker. In *Interactive Decision Theory and Game Theory*, 2010.

[24] John W. Romein and Henri E. Bal. Solving the game of awari using parallel retrograde analysis. Technical report, Vrije Universiteit, Faculty of Sciences, Department of Mathematics and Computer Science, Amsterdam, The Netherlands, 2003.

[25] Larry Russ. *The complete mancala games book*. publishers group west, 2000.

[26] Richard Russel. Kalah - the game and the program. *Artificial Intelligence Project Memo nr 22. University of Stanford*, 1964.

[27] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall, 2010.

[28] Frederik Christiaan Schadd. Monte-carlo search techniques in the modern board game thurn and taxis. *M. sc, Maastricht University, Netherlands*, 2009.

[29] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

[30] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook the world man-machine checkers champion. *AI Magazine*, 17(1):21, 1996.

[31] Brian Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134(1):241–275, 2002.

[32] David Silver and Aja Huang. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[33] J. R. Slagle and J. K. Dixon. Experiments with some programs that search game trees. *Journal of the Association for Computing Machinery*, 1969.

[34] Daniel Whitehouse, Peter I Cowling, Edward J Powley, and Jeff Rollason. Integrating monte carlo tree search with knowledge-based methods to create engaging play in a commercial mobile game. In *AIIDE*, 2013.

[35] Wikipedia. Alpha–beta pruning — wikipedia, the free encyclopedia, 2017. https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning.

[36] Wikipedia. Arthur samuel — wikipedia, the free encyclopedia, 2017. http://en.wikipedia.org/w/index.php?title=Arthur_Samuel.

[37] Wikipedia. Artificial intelligence (video games) — wikipedia, the free encyclopedia, 2017. http://en.wikipedia.org/w/index.php?title=Artificial_intelligence_(video_games).

[38] Wikipedia. Chess — wikipedia, the free encyclopedia, 2017. http://en.wikipedia.org/w/index.php?title=Chess.

[39] Wikipedia. Computer chess — wikipedia, the free encyclopedia, 2017. http://en.wikipedia.org/w/index.php?title=Computer_chess.

[40] Wikipedia. Computer othello — wikipedia, the free encyclopedia, 2017. http://en.wikipedia.org/w/index.php?title=Computer_Othello.

[41] Wikipedia. Deep blue (chess computer) — wikipedia, the free encyclopedia, 2017. http://en.wikipedia.org/w/index.php?title=Deep_Blue_(chess_computer).

[42] Wikipedia. Kalah — wikipedia, the free encyclopedia, 2017. https://en.wikipedia.org/wiki/Kalah.

[43] Wikipedia. Maven (scrabble) — wikipedia, the free encyclopedia, 2017. http://en.wikipedia.org/w/index.php?title=Maven_(Scrabble).

[44] Wikipedia. Solved game — wikipedia, the free encyclopedia, 2017. http://en.wikipedia.org/w/index.php?title=Solved_game.