Автономная некоммерческая организация
высшего образования
«Университет Иннополис»

Autonomous noncommercial organization
of higher education
“Innopolis University”

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
ПО НАПРАВЛЕНИЮ ПОДГОТОВКИ
09.03.01 – «ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

GRADUATE THESIS FILED OF STUDY
09.03.01 – «COMPUTER SCIENCE»

НАПРАВЛЕННОСТЬ (ПРОФИЛЬ) ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ
«ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»
AREA OF SPECIALIZATION / ACADEMIC PROGRAM TITLE:
«COMPUTER SCIENCE »

Тема

Topic **Quoridor Agent using Monte Carlo Tree Search**

Работу выполнил /
Thesis is executed by **Victor Massagué Respall**

подпись / signature

Научный руководитель /
Thesis supervisor **Joseph Alexander Brown**

подпись / signature

Иннополис, Innopolis, 2018

Contents

1 Introduction **2**

2 Literature Review **4**
2.1 Quoridor 4
2.1.1 Rules of the game 4
2.1.2 Notation 6
2.1.3 Game complexity 8
2.2 Agents in different board games 9
2.2.1 Chess 9
2.2.2 Carcassonne 9
2.2.3 Checkers 10
2.3 Agents using MCTS 10
2.3.1 Go 10
2.3.2 Backgammon 10
2.3.3 Scrabble 10
2.4 Previous agents for Quoridor 11
2.5 Game tree 11
2.6 Monte Carlo Tree Search 12
2.7 Genetic Algorithm 14

3 Methodology **17**

4 Implementation **19**

5 Evaluation and Discussion **25**
5.1 Experimental Settings 25
5.1.1 120k Simulations Agent 25
5.1.2 60k Simulations Agent 25
5.1.3 Alternative Agent 25
5.1.4 Genetic Algorithm Agent 26
5.2 Results 27

6 Conclusion **29**

List of Tables

2.1 Example of sequence moves in Figure 2.5 8
2.2 Complexities of Some Well-Known Games 9

5.1 Population used for the tournament with weights of each feature and fitness 27
5.2 Comparison of the 120k Agent to Other Agent Types (Significant at 95% Confidence in Bold using a Binomial exact test) 28

List of Figures

2.1 Initial state of the board with painted baseline of each player . . 5
2.2 Pawn's basic moves 5
2.3 Allowed movements for lower pawn 6
2.4 Board coordinates [1] 7
2.5 Notation used for the project [1] 7
2.6 Partial game tree for Tic Tac Toe [2] 12
2.7 Outline of Monte Carlo Tree Search 13
2.8 One generation is broken onto a selection phase and recombination phase. This figure shows strings being assigned into adjacent slots during selection. In fact, they can be assigned slots randomly in order to shuffle the intermediate population. Mutation (not shown) can be applied after crossover. [3] 15
2.9 Example of Roulette-wheel where chromosome 1 has the highest probability to be selected 16

3.1 Breadth First Search example 18

4.1 Example of graph representation of a board from pink player point of view 20
4.2 Moving a pawn 20
4.3 Adding a fence 21
4.4 Monte Carlo tree search for deciding which move to perform next 22
4.5 Selection phase 22
4.6 Expansion phase 22
4.7 Backpropagation phase 23
4.8 Simulation phase 23
4.9 Heuristic function used for simulating the game 23
4.10 Internal representation of the board 24

Abstract

This thesis presents a preliminary study using Monte Carlo Tree Search (MCTS) upon the board game of Quoridor. Quoridor is an interesting game for expansion of player agents in MCTS due to having a mechanically simple rule set, however, Quoridor has a state-space complexity similar to Chess and a higher game-tree complexity. The system is shown to perform well against current existing methods, defeating a set of player agents drawn from an existing digital implementation as well as a previous method using Genetic Algorithms.

Chapter 1

Introduction

With the emergence of computers, building artificial intelligence agents for different games have become an exciting sphere. An agent is an autonomous character that takes in information from the game data, determine what actions to take based on the information, and carry out those actions [4].

The goal is to develop an agent able to beat human players in one particular game, or in other words, solving the game. The first attempts were trying to generate the whole game tree, but in most of the games, this is not efficient due to the size of the tree and the computational time required. The biggest challenge is to find alternatives and new techniques to solve games in a reasonable computational time.

There has been a predominant trend of introducing artificial intelligence in games. Strategy based games are the most appealing to be played against an artificial player. However, generating winning solutions via an algorithm is quite challenging. Taking for example *Tic Tac Toe*, it is affordable to use the *Minimax* algorithm but looking into a more complicated game like *Chess* [5] [6] [7] and *Go* [8] [9] [10], generating the whole game tree is not affordable. Sometimes using techniques such as alpha-beta pruning [11] [12], which reduces the size of the tree significantly, is not feasible either. This is why new methods to build an agent are emerging.

Monte Carlo Tree Search (MCTS) [13] [14] is a technique well-known these days [15] due to the efficient results obtained in the board game *Go* [8]. This game, produces difficulty for an AI expert to create an agent, due to its space-complexity, branching factor and difficulties to evaluate the state of the game in the middle. To deal with these, Monte Carlo tree search was used because of its following properties. It uses *UCT* (Upper Confidence Bound applied to Trees) [16] for evaluating the final states of the game. Also, it consists of Monte Carlo roll-outs, explained later in section 2.6. , to estimate the value of each state in the search tree. As the tree grows larger more accurate values are generated. The average of these roll-outs can provide an effective position evaluation achieving accurate performance in games such as Backgammon [17] and Scrabble [18].

This thesis presents the research on the board game Quoridor to develop an artificial player agent using the Monte Carlo tree search algorithm. Due to the lack of research done on this board game, we decided to create our agent using this technique because of its efficient performance in *Go* [8], and favourable results in other board games as Backgammon [17] and Scrabble [18]. Quoridor has a state-space complexity similar to *Chess* [5] and a higher game-tree complexity (explained in section 2.1.3). These characteristics have led to the application of the MCTS algorithm for *Go* [8].

The remainder of the paper is organized as follows. Chapter 2 explains Quoridor with the rules and notation used for this project, related works and basic knowledge about Monte Carlo Tree search and Genetic Algorithm. Chapter 3 the method used. Implementation of this method has been explained in Chapter 4. Chapter 5 presents the experiments and results obtained. Finally the conclusions extracted from all the research done are provided in Chapter 6.

Chapter 2

Literature Review

2.1 Quoridor

Quoridor [19] is a strategy board game for two to four players. It was created in France by Gigamic in 1997 as a result of the evolution of a game created by Mirko Marchesi in 1975. Quoridor received the *Mensa Mind Game prize* in 1997 and the *Game Of The Year* in the USA, France, Canada and Belgium.

Compared to well-known games like Chess and Go, Quoridor is a relatively new game and has not been extensively analysed. This game is not solved and a little information about winning strategies can be found.

2.1.1 Rules of the game

Quoridor is played in a nine by nine board, and this project is focused only on the two-player version. Each player is represented by a pawn which begins at the centre space in opposite edges of the board (Figure 2.1), the baselines. The goal is to be the first player to move their pawn from its side to the opposite side of the board, the opposite baseline.

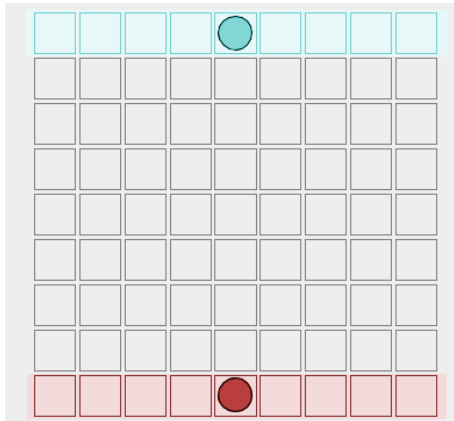

Figure 2.1: Initial state of the board with painted baseline of each player

The main feature that makes this game interesting and tactical is its fences. Fences are flat two-space-wide pieces which can be placed in the groove between the squares of the board. Fences have the ability to facilitate the player's progress or block the path of the pawns, which must go around them. Each player has ten fences at the start of the game, and once placed, cannot be moved or removed.

Each player at his turn can choose to move his pawn or to place one of his fences. once the player runs out of fences, its pawn must be moved. Pawns are moved one square at a time, horizontally or vertically, forwards or backwards (Figure 2.2).

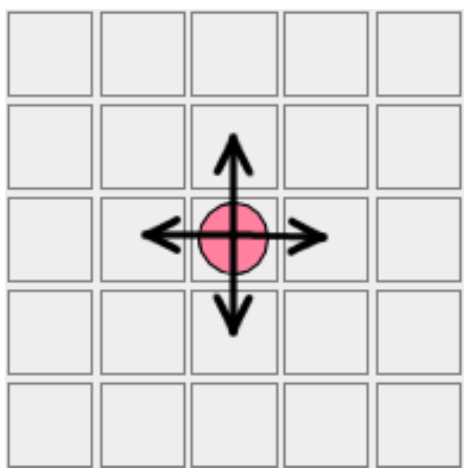

Figure 2.2: Pawn's basic moves

When two pawns face each other on neighbouring squares which are not

separated by a fence, the player whose turn is it can jump over the opponent's pawn and place himself behind the opponent's pawn, thus advancing an extra square. If there is a fence behind the said pawn, the player can place his pawn to the left or the right of the opponent's pawn. Fences may not be jumped, including when moving laterally due to a fence being behind a jumped pawn (Figure 2.3).

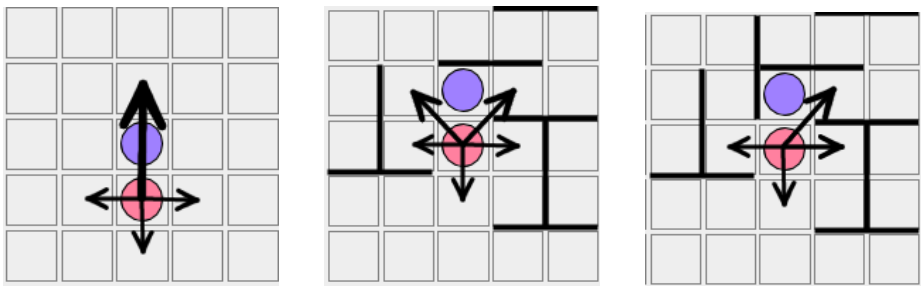

Figure 2.3: Allowed movements for lower pawn

Fences can be placed directly between two spaces, in any groove not already occupied by a fence. However, a fence may not be placed which cuts off the only remaining path of any pawn to the goal. The first player who reaches one of the 9 squares of his opponent's base line is the winner.

2.1.2 Notation

In order to allow for the games to be examined, we must come up with a novel notation for the moves made by both players. There is no official notation for Quoridor, so for this project, we use a notation from a community of players [1].

The notation proposed shown in Figure 2.4 is similar to algebraic Chess notation. Each square gets a unique letter-number designation. Each column is given a letter *A-I* and each row is given a number *1-9*. A move is recorded as the column followed by the row. The first player always starts on *E1* and the second player always starts on *E9*. This marks the top and the bottom of the board, which are needed for recording fence placement.

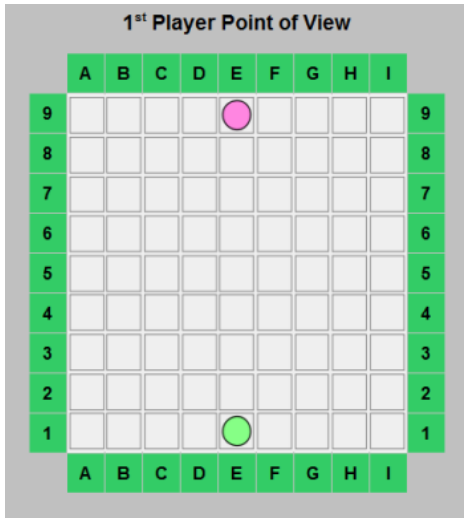

Figure 2.4: Board coordinates [1]

Each fence move is defined by the lower left square they touch along with their direction, horizontal or vertical. Every fence touches four squares, so we denote the position of the fence by the square closest to the *A1* corner of the board (Figure 2.5).

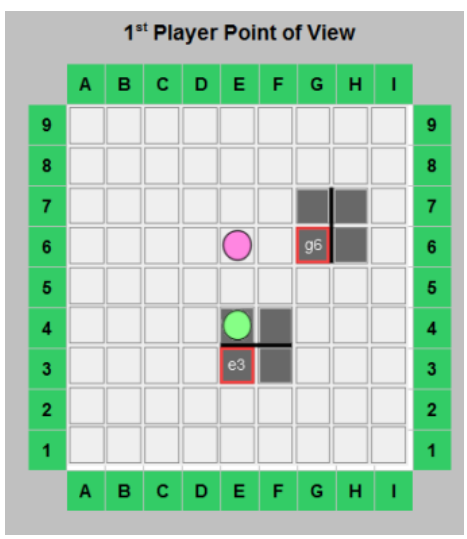

Figure 2.5: Notation used for the project [1]

	Player 1	Player 2
1	E2	E8
2	E3	E7
3	E4	E6
4	E3h	G6v

Table 2.1: Example of sequence moves in Figure 2.5

To distinguish between if it is a pawn move or a fence move, we can see in Table 2.1 that pawn moves are just denoted as a cell coordinate in comparison with fence moves that need to indicate the orientation, either horizontally (denoted *cellCoordinate* + *h*) or vertically (denoted *cellCoordinate* + *v*).

2.1.3 Game complexity

The state-space complexity of a game is the number of legal game positions reachable from the initial position of the game [20]. If this is too complex to calculate, an upper bound can often be computed by including illegal positions or positions that can never arise in the course of a game.

Quoridor has the number of possible ways to determine a fast upper bound on complexity, such as pawns moves multiplied by the number of possible ways to place the fences. Since the board has eighty one squares, we can place the first pawn in any of them, and the second one in eighty, due to the first pawn already placed on the board. Hence, the total number of positions, S_p, with two pawns is given by the following equation:

$$S_p = 81 * 80 = 6480 \tag{2.1}$$

Further, for the fences, since each fence occupies 2 squares, there are eight ways to place a fence in one row. Given that there are eight rows, there are sixty four possible places to put a fence horizontally. Since the board is a square, we have the same number of rows and columns, one fence can be put in one hundred and twenty eight places. We have to take into account that one fence occupies four fence positions, except for the squares on the border. So the total number of positions of the twenty fences, S_f, is given by the following equation [21]:

$$S_f = \sum_{i=0}^{20} \prod_{j=0}^{i} (128 - 4i) = 6.1582 * 10^{38} \tag{2.2}$$

Finally, the upper bound of the size of the state space is [21]:

$$S = S_p * S_f = 6480 * 6.1582 * 10^{38} = 3.9905 * 10^{42} \tag{2.3}$$

Table 2.2 shows a comparison between complexities of different well-known games. Quoridor has a state-space complexity similar to Chess and a higher game-tree complexity, shown to be 10^{162} in [21].

Game	State-space complexity [1]	Game-tree complexity [1]
Tic-tac-toe	3	5
Checkers [22] [23]	20 or 18	31
Connect Four [22] [24]	13	18
Nine Men's Morris [22]	10	50
Reversi [22]	28	58
Chess [25]	47	123
Connect6 [26]	172	140
Backgammon [27]	20	144
Quoridor [28]	**42**	**162** [21]
Carcassonne [29]	>40	195
Go (19x19) [22] [30] [31]	170	360

Table 2.2: Complexities of Some Well-Known Games

2.2 Agents in different board games

In this section some characteristics of relevant AI agents for board games are explained. Algorithms used, features and results obtained for each one of them.

2.2.1 Chess

Deep Blue [5] was a chess-playing computer developed by IBM, and the first computer chess-playing system to win both a chess game and a chess match against a reigning world champion under regular time controls. The system derived its playing strength mainly from brute force computing power. It was a parallel system designed specially for searching into chess game tree. It was capable of evaluating 200 million positions per second. Deep Blue's evaluation function was initially written in a generalized form, with many to-be-determined parameters.

2.2.2 Carcassonne

Carcassonne is a recently board game for 2 to 5 players which the entire state of the game is fully observable to each of the players. It is a non-deterministic game because randomly the player has to take tiles at each turn. The algorithm used for developing this agent is Expectimax search [32], which is based on Minimax search. Applying negamax algorithm [33] and alpha-beta pruning [11] [34] the performance was improved reducing the branching factor number. The level achieved allows to win against advanced human players.

[1] As *log* to base 10

2.2.3 Checkers

Chinook is a computer program that plays checkers [35]. It was developed by a team led by Jonathan Schaeffer. It is the first computer program to win the world champion title in a competition against humans. Chinook's algorithm includes an opening book, a library of opening moves from games played by grandmasters, a deep search algorithm, a good move evaluation function and an end-game database for all positions. The evaluation function considers several features of the game board, including piece count, kings count, trapped kings, turn and runaway checkers.

2.3 Agents using MCTS

This section describes the most relevant board games that successfully applied MCTS.

2.3.1 Go

AlphaGo is an agent that plays the board game Go [9], developed by Google. In October 2015, AlphaGo became the first computer Go program to beat a human professional Go player without handicaps on a full-sized 19x19 board. This agent uses a Monte Carlo tree search algorithm to find moves based upon knowledge by machine learning, specifically by an artificial neural network, extensively trained both from human and computer play.

2.3.2 Backgammon

McGammon [36] is an agent that uses MCTS in Backgammon for training a neural network by playing millions of offline games. It is presented a contrasting approach consisting of using only online simulated games from the current position. A simplification of the game rules is made for computing random games faster. The agent is able to play 6500 approximately per second on a quad-opteron 2.6 GHz CPU. It is able to find some expert moves just even with a simplified version of the game.

2.3.3 Scrabble

MAVEN [18] was the first program to demonstrate that it is possible to beat human players in Scrabble. It is a game of imperfect information with a large branching factor. The techniques successfully applied in two-player games such as chess do not work here. MAVEN combines a selective move generator, simulations of likely game scenarios, and the B algorithm (basically techniques used in MCTS) to produce a world-championship-caliber Scrabble-playing program. If a perfect human player play against MAVEN he could only win 51 percent of the games.

2.4 Previous agents for Quoridor

As mentioned earlier, there is not much research on this game as it is relatively new. The main attempts to build an agent on Quoridor are mentioned as follows:

1. *Mastering Quoridor by Lisa Glendenning [28]*: Agent development has been done by implementing a search algorithm, using iterative-deepening alpha-beta negamax search algorithm with few modifications. For the evaluation function, the ten features chosen are summarized as:

   - Shortest path to the goal for both players
   - Markov Chain for both players
   - Manhattan Distance for both players
   - Pawn Distance
   - Goal Side for both players
   - Number of fences of the player

   For the learning algorithm, a variant of a genetic algorithm is using as a fitness evaluation function, the number of games that a chromosome wins against the others inside the population. This agent was shown to be easy to beat by human play.

2. *A Quoridor-playing Agent* [21]: MiniMax algorithm is used in this case with Alpha-Beta pruning, but the game tree is to large to perform MiniMax search all the way down to the leaves of the tree. Therefore, the solution is limiting the depth of the MiniMax search. Further, an evaluation function is applied to determine the value of a position in order to allow for a quicker return. The result obtained is a weak Quoridor agent as it is unable to see depth in the game.

   MCTS balances the want for a fast evaluation mixed with the ability to see beyond a set horizon depth.

2.5 Game tree

A game tree is a directed graph whose nodes are states of a game and each edge represents a move. A game tree is complete if it contains all possible moves or possible states of the game from the initial position.

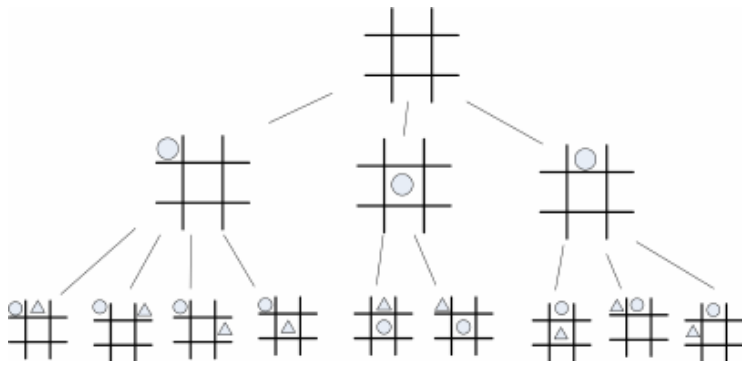

Figure 2.6: Partial game tree for Tic Tac Toe [2]

In Figure 2.6 is shown a part of Tic Tac Toe game tree. The root determines the current state of the game and also the player that plays next. The children of it are all possible moves that the current player can perform.

Game trees are so important in artificial intelligence because one way of deciding which move is better is applying some search algorithm on them like Minimax algorithm or MCTS. Specifically for Tic Tac Toe it is very convenient to build the complete game tree because it has a low branching factor. But for bigger branching factors such as Chess, it is too large to compute it completely. Instead of using a complete game tree, for this kind of games you have to work with partial game trees.

2.6 Monte Carlo Tree Search

Monte Carlo Tree Search is a probabilistic search algorithm with a unique decision-making ability because of its efficiency in open-ended environments with an enormous amount of possibilities. To deal with the size of the game tree, it applies Monte Carlo method [37]. It is based on simulating games where the AI agent and the opponent player play pseudo-random moves. As it is based on random sampling of game states, it does not need to use brute force. This characteristic allows us to simulate a big number of simulation games to collect information about movement perform.

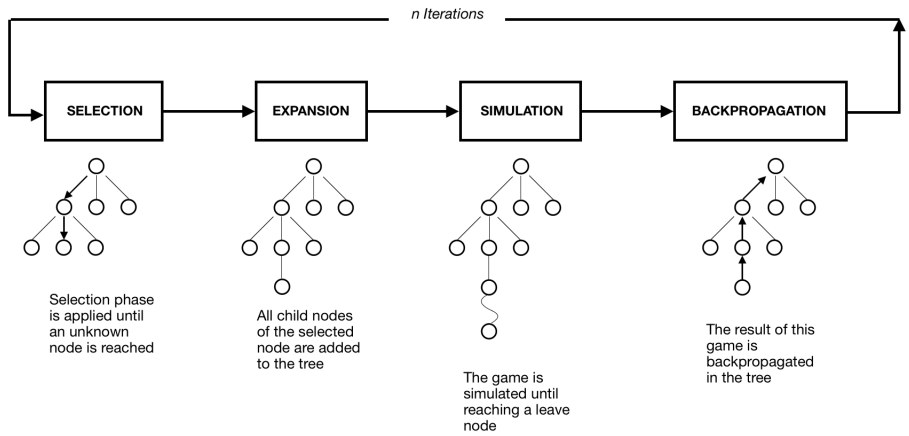

Figure 2.7: Outline of Monte Carlo Tree Search

For building a game tree this algorithm follows this four phases:

1. *Selection*: In this initial phase, the algorithm starts with a root node and selects a child node such that it picks the node with maximum win rate. In order to make sure that each node is given a fair chance to be selected and to balance the situation between exploration and exploitation, we use *UCT* [16].

$$\frac{w_i}{n_i} + c * \sqrt{\frac{ln(t)}{n_i}} \tag{2.4}$$

   Where

   - w_i = number of wins after the i-th move
   - n_i = number of simulations after the i-th move
   - c = exploration parameter (theoretically equal to $\sqrt{2}$ [14])
   - t = total number of simulations for the parent node

   This formula ensures that agent will play promising branches more often than their counterparts, but will also sometimes explore new options to find a better node, if exists.

   It selects the best node of the entire generated tree, traverses down the tree and selects a leaf node.

2. *Expansion*: When it can no longer apply UCT to find the successor node, it expands the game tree by generating all possible states from the leaf node.

3. *Simulation*: After expansion phase, the algorithm picks a node randomly and it simulates the game until the very end, randomly for both players.

4. *Backpropagation*: This last phase consists of updating the nodes according to the result of the simulation. It evaluates the state to figure out which player has won and traverses upwards to the root incrementing visit counts and win scores, i.e. a count of if the player of that position has won, of each node visited.

   The algorithm keeps looping these four phases until some fixed number of iterations. Higher the number of iterations, more reliable the estimate becomes.

2.7 Genetic Algorithm

Genetic algorithm (GA) is a method inspired in natural selection [3]. Basically it encodes a candidate solution to a specific problem on a simple data structure called chromosome. All implementations of this algorithms start with creating a population, in other words, a group of chromosomes, typically random. The size depends on the problem. During each successive generation, a part of the population is selected to breed the next generation. Solutions are selected according to a fitness evaluation, that tells how good is that solution for our problem.

Also this algorithms use mutation and crossover operators to generate new sample points in a search space. At each generation usually the best chromosomes reach the next generation. The key point on this algorithms is how to evaluate the fitness of a chromosome, because this will determine how good is your solution. For computing the fitness of each chromosome it is applied a fitness evaluation function.

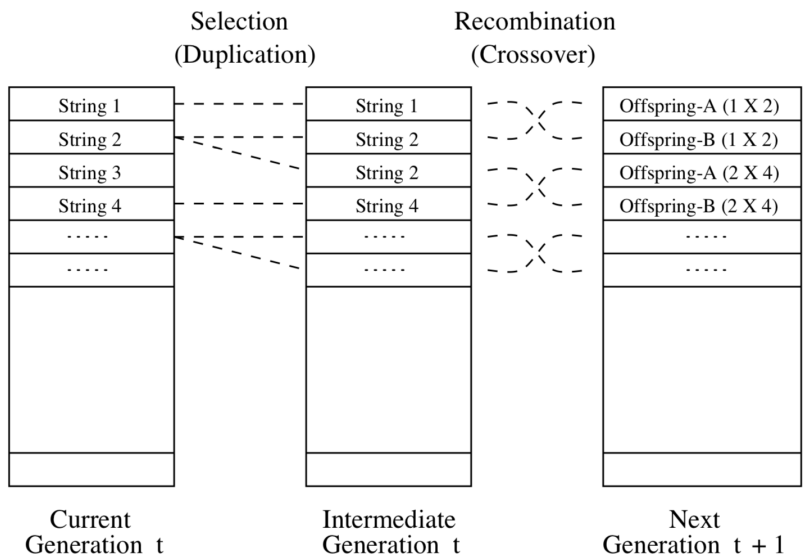

Figure 2.8: One generation is broken onto a selection phase and recombination phase. This figure shows strings being assigned into adjacent slots during selection. In fact, they can be assigned slots randomly in order to shuffle the intermediate population. Mutation (not shown) can be applied after crossover. [3]

After this phase, selection is the next step. There are many ways for implementing this [38]:

- Roulette Wheel Selection (RWS): This method says that the probability for a chromosome to. be selected is proportional to the fitness of itself. It follows this formula [39]:

$$p_i = \frac{w_i}{\sum_{i=1}^{N} w_i} (i = 1, 2, ..., N)$$

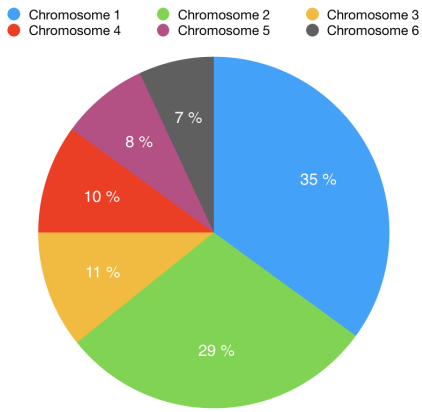

Figure 2.9: Example of Roulette-wheel where chromosome 1 has the highest probability to be selected

- Stochastic Universal Sampling (SUS): This technique uses a single random value to sample all the solutions by choosing them at evenly spaced intervals. This gives weaker members of the population, according to their fitness, a chance to be chosen and thus reduces the unfair nature of fitness-proportional selection methods. It starts from a small random number, and chooses the next candidates from the rest of population remaining, not allowing the fittest members to saturate the candidate space.

- Linear Rank Selection (LRS): It defines the target sampling rate (TSR) of an individual x as:

$$TSR(x) = Min + (Max - Min)\frac{rank(x)}{N-1}$$

  where *rank(x)* is the index of x when the population is sorted in increasing order based on fitness, and N is the size of the population. The TSR is the number of times an individual should be chosen as a parent for every N sampling operations [40].

- Tournament Selection (TOS): The idea is choose some number of individuals randomly from a population, select the best individual from this griup fro further genetic processing and repeat as often as desired. [41]

Chapter 3

Methodology

Considering all the research presented in earlier sections, we decided to use Monte Carlo Tree Search algorithm for building the AI agent for Quoridor, as it appears to be an efficient algorithm for this type of board game and game tree size.

The proposed search algorithm works as follows: We have built a game tree with a root node, then it is expanded with random simulations of the game. In the process, we maintain the number of times we have visited a specific node and a win score, used to evaluate the state of the board. The game is simulated until the end 120000 times per each decision that the agent has to do. In the end, we select the node with higher win score. In case of having more than one node with maximum win score, we take a random one from all best candidates.

When we did some experiments with the system, we realized that it is not working as we wanted. After this amount of simulations we were expecting good decisions from our agent as theory shows, but due to the amount of possible places that a fence could be placed in the board, the chances of moving the pawn were so low. This implies that all fences were placed in the first turns of the game almost randomly because both pawns are still in the initial position. This makes the agent so easy to beat, just waiting that the agent has no more fences, for example moving the pawn between two positions all the time, and afterwards start moving forward to the goal and placing fences to the agent's path for reaching first the goal.

For improving our decisions, we should make MCTS less random giving more information about the status of the board. Simulation phase should be changed. Instead of just taking a random child every time, what we do for solving the problem is simulating with an heuristic function our game until the very end for getting results more significant and valid for our agent. An heuristic function is a way of ranking all possible moves and select an approximate one to the optimal solution. For a better performance the heuristic function should be easy to compute, taking reasonable time for finding a solution and solving the problem. This solution as stated before may not be the best of all possible solutions, or it may just approximate the exact solution. However it is relevant

because it does not require a huge amount of time to find it.

It is clear that the key of this game is maximizing the opponent's shortest path while minimizing yours. So the heuristic developed will work for this purpose and also for balancing pawn moves and placement of fences.

The main feature used in the heuristic decision is shortest path to the goal of current player and opponent player. The algorithm used is Breadth First Search (BFS). BFS is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph) and explores the neighbour nodes first, before moving to the next level neighbours. In our case, the tree root is the position of the player and the algorithm stops as soon as it reaches the goal. It is guaranteed that as soon as we reach some goal square of the board, it will be the shortest path. That's the reason why we can stop the algorithm before traversing the whole board.

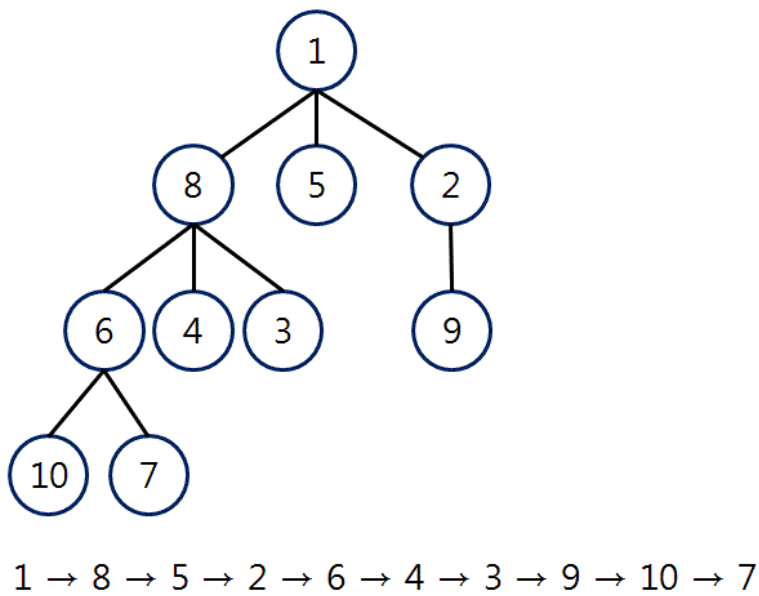

Figure 3.1: Breadth First Search example

Chapter 4

Implementation

This project is implemented in Java programming language, using standard libraries. This section explains main classes and data structures used. We start describing the classes related to the game and its playability and we finish with the algorithm for our agent.

The class *Square*, represents a single square inside a real board. It has the coordinates in *String* way (i.e. E5, B7, A2) and also numerical ones, replacing the letters for numbers. The reason is because it makes things easier when it is inside a matrix, using indexes rather than letters.

For representing a player in our game, the class *Player* stores the id for identifying which player it is, the current position of the player, the number of fences and a *Set* with all neighbour squares, in other words, the squares that the player is able to move the pawn.

Now we are able to build the board of the game using the two classes explained before, *Board* class. For representing the board we create an *ArrayList* of one dimension with all possible *squares*, and two players since this thesis just analyses two players version. The idea is to represent the board with an undirected graph. An example is shown in Figure 4.1.

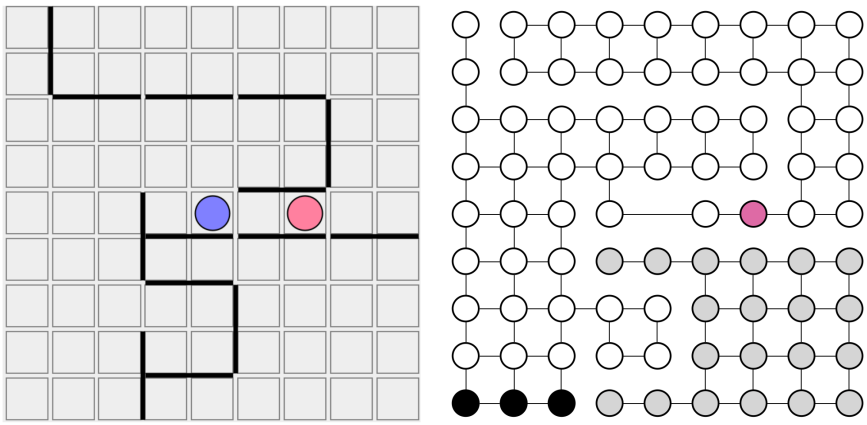

Figure 4.1: Example of graph representation of a board from pink player point of view

For managing the neighbours of each square we use a *Map* that given a coordinate (i.e. E5) returns the current neighbours of that square, in other words, returns the adjacent nodes in the graph for a given coordinate. Finally for making things faster, we have a *Set* containing all possible positions that a fence could be placed. This class also contains all functions related to perform moves, check that rules are followed and giving information to other classes about the state of the board.

We have two possible moves that a player can do at each turn, moving the pawn or placing a fence. For moving a pawn the following function is used 4.2 where *checkMove()* function checks if *new position* is inside the list of neighbours of that player and *updatePlayerNeighbours()* updates the neighbours from *new position* square.

```
function movePlayer(new_position) {
    if checkMove(new_position) {
        player.setPosition(new_position)
        updatePlayerNeighbours()
    }
}
```

Figure 4.2: Moving a pawn

For placing a fence, first we need to check that the position where we want to place the fence is inside our *Set* of fences (i.e. if we have a fence placed in E5h, our *Set* would not contain E5v because a fence cannot be intersected by another). If this is fulfilled we need to check that there exists a path for each player until the correspondent goal. This is done applying BFS algorithm

inside *checkPath()*. If there exists a path for each player, then the neighbours of the players will be updated, and the adjacent nodes to the affected squares too. *updateFenceList()* removes all positions that are no longer available and *removeFence()* it undoes the fence placed for testing 4.3.

```
function addFence(position) {
if fences.contains(position) {
     placeFence(position)
        if checkPath() {
         updateFencesList()
         updatePlayerNeighbours()
        }
        else {
            removeFence(position)
        }
    }
}
```

Figure 4.3: Adding a fence

The system explained before for storing all possible positions that fence could be placed is also useful for returning all possible movements available to the player.

From here, we have a system for playing Quoridor with the standard rules. Now we will add and agent able to play this game in our system.

The class *Monte Carlo Tree Search*, contains the brain for our agent. It builds the tree and decides which move to perform next. The first attempt to implement the tree was including in each node the board of the game. The problem was since it contains a huge amount of information and data structures, the program ran out of memory just with 5000 simulations, because MCTS was generating too many nodes. The solution is, instead of saving the board in each node generated, we store the move that we should perform and the scores of the node. Then each time when we visit a node we need to compute the move in a temporal board. The score added to each winning node is 10.

```
function findNextMove() {
    while simulations < max_simulations {
        node = selectPromisingNode()
        nodeToExplore = expandNode(node)
        playoutResult = simulateRandomPlayouts(nodeToExplore)
        backPropagation(playoutResult)
        ++simulations
    }
    return root.getChildrenWithMaxScore()
}
```

Figure 4.4: Monte Carlo tree search for deciding which move to perform next

As shown in Figure 4.4, four phases are executed *max simulation* times. First the promising node is selected 4.5 using UCT method, then this node is expanded 4.6, it means that all possible children are generated. From all children, one is selected randomly for simulating 4.8 the game until the very end, with *simulateRandomPlayouts*. Finally the result of the simulation is back propagated 4.7 until the root. When the maximum number of simulations is reached, it returns the child of the root with maximum win score.

```
function selectPromisingNode() {
    while node.children not null {
        node = findBestNodeWithUCT()
    }
    return node
}
```

Figure 4.5: Selection phase

```
function expandNode() {
    node.addChildren(getAllPossibleMoves)
}
```

Figure 4.6: Expansion phase

```
function backPropagation() {
    while node != null {
        node.incrementVisit()
        if (node.player == winner) {
            node.addScore(WINSCORE)
        }
        node = node.parent
    }
}
```

Figure 4.7: Backpropagation phase

```
function simulateRandomPlayout() {
    while gameStatus != END {
        node.togglePlayer()
        node.heuristicDecision()
    }
}
```

Figure 4.8: Simulation phase

Finally, instead of using random decisions in the simulation phase of Monte Carlo Tree Search, we improved our system by adding a heuristic. The heuristic helps us to balance the placement of fences and the moves of the pawn. Running MCTS with random simulation shows that our agent spends all fences at the beginning of the game and on average is better to save them until the middle of the game. The heuristic decision used in the simulation phase is basically based on comparing if the shortest path until the goal of the current player is less than the opponent's one. If yes, just pawn moves are available, for making the game faster on finishing, else all possible moves are candidates 4.9.

```
function heuristicDecision() {
    if distanceToGoalPlayer1 <= distanceToGoalPlayer2 || player1.fences == 0 {
        followShortestPath()
    }
    else {
        Random(allPossibleMoves)
    }
}
```

Figure 4.9: Heuristic function used for simulating the game

To keep track on the game situation, a board is printed each round using the following example of representation:

```
9 . .|. . .|.|. . .
  ---|---  | |
8 . .|. . .|.|. . .
        --- --- ---
7 . . . . . . . . .

6 . . .|. . . . . .
  ---  |
5 .|. 2|. .|. . . .
   |       |
4 .|. . .|.|. . . .
         |
3 . 1 . .|. . . . .
    --- --- --- ---
2 . . . . .|. . . .
  ---   ---|
1 . . . . .|. . . .
  A B C D E F G H I
```

Figure 4.10: Internal representation of the board

The numbers 1 and 2 inside the board represent each player. Player 1 starts always at position *E1* and Player 2 at *E9*. Each dot represents a square of the board and the fences are represented as dashes between the dots.

Chapter 5

Evaluation and Discussion

5.1 Experimental Settings

Due to limited research on the game of Quoridor and no human Elo rankings, we cannot measure the level of the agent globally. However, we have tried to evaluate our player against self play with more simulation steps and against other player agent types to have an idea of how efficient our agent is.

5.1.1 120k Simulations Agent

The default agent for running the experiments uses MCTS with the heuristic described before. It performs 120000 simulations of the game per decision.

5.1.2 60k Simulations Agent

This agent was created to see the influence in the number of simulations of the game per decision. The only difference between this agent and the one stated in subsection 5.1.1 is that this agent is doing 60000 simulations per decision.

5.1.3 Alternative Agent

To further evaluate the MCTS method, an alternative Quoridor agent base was found with four different agent levels *(Brain1, Brain2, Brain3 and Brain4)* [42]. *Brain1* simply moves the pawn at every turn without placing any fence. *Brain2*, places all fences at the beginning of the game, wasting all resources. *Brain3* places fences more strategically, but still the problem of placing all fences at the beginning of the game. That gives a lot of advantage to the opponent then. *Brain4* is the smartest agent, it focuses on reaching the goal and it uses fences during the middle of the game. It is sadly impossible to describe the algorithm that is using, because of the lack of information about the API from the developers.

5.1.4 Genetic Algorithm Agent

This agent is based on a previous work, mentioned in Section 2.4, Mastering Quoridor of Lisa Glendenning [28]. For developing such an agent we used Minimax algorithm [43] with some modifications to improve its performance. It is possible to modify the game tree values to use just maximization operations, negating the returned values from the recursion. This approach is called *Negamax algorithm* [33]. However the problem with Minimax search [43] is that the number of states it has to examine is exponential in the number of moves. For reducing this amount of moves it is applied *alpha-beta pruning* [11] [34] technique, that basically prunes away branches of the tree that cannot influence the final decision. Last modification applied was iterative-deepening due to the agent plays with time limit decision for every move. This allows us to return the best value computed until that point [44]. After each execution of alpha beta, it is required an evaluation function for selecting the best state of the board. Eight features are proposed for Quoridor:

- Shortest Path Player (SPP), length of Breadth First Search path for the player
- Shortest Path Opponent (SPO), length of Breadth First Search path for the opponent
- Manhattan Distance Player (MDP), Manhattan distance for the player (straight distance from the player to the goal)
- Manhattan Distance Opponent (MDO), Manhattan distance for the opponent (straight distance from the opponent to the goal)
- Pawn Distance (PD), the distance between pawns using Breadth First Search
- Goal Side Player (GSP), boolean that tells if the player is between the midpoint of the board and the goal
- Goal Side Opponent (GSO), boolean that tells if the opponent is between the midpoint of the board and the goal
- Number Fences Player (NFP), number of fences of the player

Finally a genetic algorithm (GA) [3] [45] [46] is used for weighting each feature described before. A chromosome is represented as a vector of weights and the fitness of each one is determined by the number of games that a chromosome wins against the other chromosome of the population. To create a population it is initialized with random float point values. It is established with a probability of 0.3 that a chromosome has a non-zero weight in some feature.

The experiments performed are as follows: 120k agent vs 60k agent, 120k agent vs Brain1, 120k agent vs Brain2, 120k agent vs Brain3, 120k agent vs Brain4, 120k agent vs GA and 120k agent vs Psi1 which is the best chromosome extracted from the results of *Mastering Quoridor* [28].

5.2 Results

For selecting the Genetic Algorithm between all chromosomes created inside a population, we performed a tournament consisting about creating a population of 10 chromosomes and each one plays against each other. This will give us the fitness of the chromosomes. The maximum number of moves to avoid infinite loops was set to 120, and the decision time to 10 seconds per move. For playing against our 120k agent we took the two best chromosomes of the tournament, as shown in Table II, chromosome 2 and 5 with fitness 11 and 10 respectively.

Chromosome	SPP	SPO	MDP	MDO	PD	GSP	GSO	FSP	Fitness
0	0.0	0.0	0.0	0.0	0.412	0.0	0.0	0.0	2
1	0.0	0.0	0.0	0.0	0.412	0.0	0.649	0.0	8
2	0.0	0.0	0.0	-0.249	0.0	0.0	0.649	0.0	11
3	0.244	0.0	0.0	0.0	0.0	0.0	-0.073	0.0	3
4	0.0	0.0	0.0	-0.527	0.0	0.0	0.0	0.0	1
5	0.91	0.0	0.0	-0.249	0.0	0.0	-0.319	0.0	10
6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.078	1
7	0.0	-0.339	-0.954	0.0	0.0	0.0	-0.073	0.0	5
8	0.0	0.0	0.0	0.0	0.412	0.0	0.649	0.0	5
9	0.0	-0.339	0.0	0.0	0.0	0.0	0.0	0.0	2

Table 5.1: Population used for the tournament with weights of each feature and fitness

In all experiments performed, our player was the default, 120k Agent. The experiment with 60k Agent was done automatically for 250 games since it was easy to adapt the code to play against itself. However, as we do not have the source code for the alternative agents, the experiments with all of them were done manually, taking the moves from our agent and playing them against the four brains. In the evaluative process, first ten games were played against each of the four brain types, when there was no an obvious mercy situation, i.e. in the case of *Brain4*, then we extended this to 100 games in order to allow for a statistical evaluation.

Opponent	Number of Games Played	Percentage (Count) of Wins for 120k
60k Agent	210[1]	46% (97 games)
Brain1	10	100%
Brain2	10	100%
Brain3	10	100%
Brain4	100	66% (66 games)
Chromosome2	10	100%
Chromosome5	10	100%
Psi1 [28]	10	100%
Opponent	Lower Confidence Bound (95%)	Upper Confidence Bound (95%)
60k Agent	39.3%	53.2%
Brain1	69.2%	100%
Brain2	69.2%	100%
Brain3	69.2%	100%
Brain4	55.9%	75.2%
Chromosome2	69.2%	100%
Chromosome5	69.2%	100%
Psi1 [28]	69.2%	100%

Table 5.2: Comparison of the 120k Agent to Other Agent Types (Significant at 95% Confidence in Bold using a Binomial exact test)

Table 5.2 shows that reducing the amount of simulations to 60000, our agent performs a little bit better than 120k agent, and much faster to decide each movement. Though this result is not significant at $p < 0.05$. Note that in playing these games, in 40 instances the MCTS players began to alternate moving a pawn back and forth, not seeing an obvious solution to winning the game. We therefore can infer there is a situation in the game which such a delaying tactic has some amount of value, or for which there is not an obvious good strategy. This only happened in self play, and more analysis is required to understand what developed this situation, as there is no obvious tie state.

The first three brains of the alternative agent and Chromosome 2 and 5 of the GA agent were easy to beat, with significant results over the ten evaluations each the MCTS would win 100% of the time. However, with *Brain4*, it is not possible for our agent to win all the time, but still performed better that the opponent, significant at $p < 0.05$. There is no information about which method is used by this agent so we are unable to make a deep evaluation as to the play method which is able to at least give some challenge to MCTS. Finally following the results described in [28], we created the best chromosome achieved, Psi1, and the MCTS was also able to defeat it in all of the ten games played.

[1] 250 games in total were played with 40 being undecided, we only examine the difference of wins in decided games

Chapter 6

Conclusion

We have created an MCTS agent for the board game Quoridor and compared it to a number of previous agent types, including reimplementation of a GA. This research work completed thus far focuses only in the two players version, Quoridor can be played with four players, each with the goal of taking their pawn the opposite end of the board, the two other players take their pawns horizontally from the side baselines.

We have used Monte Carlo tree search as the main algorithm. It is a probabilistic search algorithm, and a unique decision making because of its efficiency in open-ended environments with an enormous amount of possibilities. Also, we have added a heuristic to balance the placement of fences and the moves of the pawn, in the simulation phase of the MCTS algorithm. The results obtained from the experiments are not sufficient to determine precisely the level of our agent, but it gives us an estimation of how it will preform in play against humans.

The 60k agent as shown in Table 5.2 appears to preform better than the 120k agent and has a shorter runtime. Though this study will need to extend in order to prove this trend to hold in larger cases, and then evaluate why more simulations would show the result to be.

The future work should take into account the improvement in the heuristic, to decide a better quality movement and consider more features of the game such as a number of fences of each player. After applying GA in this game using the features described in the thesis work [28], it gives an idea of which kind of features are relevant for improving our agent. More research on features should be done. This will build a solid strategy for an agent. Moreover, as used in AlphaGo, deep learning can also be used [8]. Finally, it is the goal of the authors to show that this system is competitive against the ranked human play. There are a number of human play strategies which are used commonly in competitive play, much along the same lines as chess openings, and perhaps it would be best to add an evaluated game tree as an initialization step to ensure competitive play.

Bibliography

[1] Quoridor Strats, "Notation," September 2014, https://quoridorstrats.wordpress.com/notation/.

[2] L. Li, H. Liu, H. Wang, T. Liu, and W. Li, "A parallel algorithm for game tree search using gpgpu," vol. 26, 07 2014.

[3] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[4] I. Millington and J. Funge, *Artificial intelligence for games*. CRC Press, 2016.

[5] M. Campbell, A. J. Hoane, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[6] F.-H. Hsu, *Behind Deep Blue: Building the computer that defeated the world chess champion*. Princeton University Press, 2004.

[7] A. Newell, J. C. Shaw, and H. A. Simon, "Chess-playing programs and the problem of complexity," *IBM Journal of Research and Development*, vol. 2, no. 4, pp. 320–335, 1958.

[8] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016. [Online]. Available: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html

[9] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[10] M. Müller, "Computer go," *Artificial Intelligence*, vol. 134, no. 1-2, pp. 145–179, 2002.

[11] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial intelligence*, vol. 6, no. 4, pp. 293–326, 1975.

[12] J. Schaeffer, "The history heuristic and alpha-beta search enhancements in practice," *IEEE transactions on pattern analysis and machine intelligence*, vol. 11, no. 11, pp. 1203–1212, 1989.

[13] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *International conference on computers and games*. Springer, 2006, pp. 72–83.

[14] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*. Springer, 2006, pp. 282–293.

[15] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[16] S. Gelly, Y. Wang, O. Teytaud, M. U. Patterns, and P. Tao, "Modification of uct with patterns in monte-carlo go," 2006.

[17] G. Tesauro and G. R. Galperin, "On-line policy improvement using monte-carlo search," in *Advances in Neural Information Processing Systems*, 1997, pp. 1068–1074.

[18] B. Sheppard, "World-championship-caliber scrabble," *Artificial Intelligence*, vol. 134, no. 1-2, pp. 241–275, 2002.

[19] M. Marchesi, *Quoridor*. Family Games, Inc., 1997.

[20] L. V. Allis *et al.*, *Searching for solutions in games and artificial intelligence*. Rijksuniversiteit Limburg, 1994.

[21] P. J. Mertens, "A quoridor-playing agent," *Bachelor Thesis, Department of Knowledge Engineering, Maastricht University*, 2006.

[22] V. Allis, *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen, 1994.

[23] J. M. Robson, "N by n checkers is exptime complete." *SIAM J. Comput.*, vol. 13, no. 2, pp. 252–267, 1984. [Online]. Available: http://dblp.uni-trier.de/db/journals/siamcomp/siamcomp13.html#Robson84

[24] J. Tromp, "John's connect four playground," 2010.

[25] C. E. Shannon, *Programming a Computer for Playing Chess*. New York, NY: Springer New York, 1988, pp. 2–13. [Online]. Available: https://doi.org/10.1007/978-1-4757-1968-0_1

[26] C.-M. Xu, Z. M. Ma, J.-J. Tao, and X.-H. Xu, "Enhancements of proof number search in connect6," in *Proceedings of the 21st Annual International Conference on Chinese Control and Decision Conference*, ser. CCDC'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 4561–4565. [Online]. Available: http://dl.acm.org/citation.cfm?id=1714810.1715001

[27] G. Tesauro, "Practical issues in temporal difference learning," *Machine Learning*, vol. 8, no. 3, pp. 257–277, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992697

[28] L. Glendenning *et al.*, "Mastering quoridor," *Bachelor Thesis, Department of Computer Science, The University of New Mexico*, 2005.

[29] C. Heyden, "Implementing a computer player for carcassonne," *Master's thesis, Department of Knowledge Engineering, Maastricht University*, 2009.

[30] J. Tromp and G. Farnebäck, "Combinatorics of go," in *International Conference on Computers and Games*. Springer, 2006, pp. 84–99.

[31] J. Tromp, "The number of legal go positions," in *International Conference on Computers and Games*. Springer, 2016, pp. 183–190.

[32] C. Heyden, "Implementing a computer player for carcassonne," *Master's thesis, Department of Knowledge Engineering, Maastricht University*, 2009.

[33] M. S. Campbell and T. A. Marsland, "A comparison of minimax tree search algorithms," *Artificial Intelligence*, vol. 20, no. 4, pp. 347–367, 1983.

[34] J. Pearl, "The solution for the branching factor of the alpha-beta pruning algorithm and its optimality," *Communications of the ACM*, vol. 25, no. 8, pp. 559–564, 1982.

[35] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, "Checkers is solved," *Science*, vol. 317, no. 5844, pp. 1518–1522, 2007.

[36] F. Van Lishout, G. Chaslot, and J. W. Uiterwijk, "Monte-carlo tree search in backgammon," 2007.

[37] N. Metropolis and S. Ulam, "The monte carlo method," *Journal of the American statistical association*, vol. 44, no. 247, pp. 335–341, 1949.

[38] K. Jebari and M. Madiafi, "Selection methods for genetic algorithms," *International Journal of Emerging Sciences*, vol. 3, no. 4, pp. 333–344, 2013.

[39] A. Lipowski and D. Lipowska, "Roulette-wheel selection via stochastic acceptance," *Physica A: Statistical Mechanics and its Applications*, vol. 391, no. 6, pp. 2193–2196, 2012.

[40] R. J. Collins and D. R. Jefferson, *Selection in massively parallel genetic algorithms*. University of California (Los Angeles). Computer Science Department, 1991.

[41] B. L. Miller, D. E. Goldberg *et al.*, "Genetic algorithms, tournament selection, and the effects of noise," *Complex systems*, vol. 9, no. 3, pp. 193–212, 1995.

[42] M. van Steenbergen, "Quoridor," online, 2006, http://martijn.van.steenbergen.nl/projects/quoridor/.

[43] G. C. Stockman, "A minimax algorithm better than alpha-beta?" *Artificial Intelligence*, vol. 12, no. 2, pp. 179–196, 1979.

[44] N. J. Nilsson, "Artificial intelligence: A modern approach: Stuart russell and peter norvig,(prentice hall, englewood cliffs, nj, 1995); xxviii+ 932 pages," 1996.

[45] L. Davis, "Handbook of genetic algorithms," 1991.

[46] C. M. Anderson-Cook, "Practical genetic algorithms," 2005.