School of Electronics and Computer Science
Faculty of Engineering, Science and Mathematics
University of Southampton

Samuel John Odell Miller

May 26, 2009

Researching and Implementing a Computer Agent to Play Arimaa

Supervisor: Dr Alex Rogers
Second Examiner: Prof Mahesan Niranjan

A project report submitted for the award of
Computer Science with Artificial Intelligence Meng

Abstract

This project is concerned with the board game Arimaa and researches ways to implement an agent to play the game. The project agent, named bot_degree is implemented in C/C++ and uses two specific algorithms to construct and traverse the game tree; the UCT algorithm and Monte-Carlo Simulation. UCT is an extension of the mini-max bandit algorithm, called UCB1 (Upper Confidence Bounds), applied to trees and provides a way to manage the trade-off between exploration and exploitation. Monte-Carlo Simulation has been implemented to help evaluate nodes within the game tree and assist in the traversal of the UCT algorithm. Performance analysis was carried out for bot_degree and showed that bot_degree could win against the random bot r0 with a win percentage of as much as 75.4% ± 3.6% with 95% confidence.

Contents

Chapter 1

Acknowledgments

I would like to thank my Parents, for proof reading this report and Dr. Alex Rogers, my project supervisor, for giving me guidance when I asked for it.

Chapter 2

Introduction

Artificial intelligence is fast becoming one of the most interesting areas within the computing industry, with many different possibilities for research. The academic community is still a long way off from strong artificial intelligence, but recently there has been a break through with getting computers to play board games well. Originally it was thought that a computer would never be able to beat a grandmaster in chess, then in 1997 Deep Blue beat the world champion, Garry Kasparov [8]. This opened the flood gates for researchers to try and develop computer agents to play other board games. One of those board games is called Arimaa [15]. Arimaa was specifically designed to be hard for computers to play, it has a high branching factor and many different combinations of starting positions. Thus the standard brute force approach is infeasible. Getting a computer agent to play Arimaa at a strong human level would be a break through in artificial intelligence because computer agents would finally be able to play intelligently and win against humans, without using any brute force techniques.

The high branching factor is common to many board games that already have computer agents that play effectively and section 3 discusses how intelligent search approaches have been applied to the game of Go [1]. The best computer agent to play Go is called MoGo [5] which uses two very interesting approaches to tackle the high branching factor problem; the UCT algorithm [5, 4] and Monte-Carlo Simulation. UCT is an extension of the mini-max bandit algorithm, called UCB1 (Upper Confidence Bounds) [10], applied to trees. It treats each of its turns independently and thus constructs a new game tree each turn. Monte-Carlo Simulation has been used in this context to provide random simulations which help in evaluating the strength of a particular move within the game tree [11]. Both these approaches work very well together and provide a way to construct and traverse the game tree as well as making a trade-off between exploration and exploitation [5].

While these approaches are very effective in the game of Go, they have not been applied to the game of Arimaa. Therefore this project is concerned with researching ways to apply UCT and Monte-Carlo Simulation to the game of Arimaa. In order to do this, some modifications of the algorithm must be made, for instance normally the UCT algorithm will explore each node on a particular level once before descending to lower levels of the game tree. This is fine in Go because even though the branching factor is high, it is still feasible. Arimaa's branching factor can be in excess of 25,000 unique possible moves [18] and thus the trade-off between exploration and exploitation must be addressed. In more detail, this project is concerned with the following:

- Testing the UCT algorithm against the ϵ-Greedy algorithm and comparing the effectiveness of the UCT algorithm with Alpha-Beta search.
- Implementing the UCT algorithm to traverse the game tree and provide a trade-off between exploration and exploitation.
- Implementing Monte-Carlo Simulation to evaluate the effectiveness of each node within the

game tree.

- Encapsulating the algorithms within an agent, named bot_degree, that is implemented in C/C++.
- Testing the effectiveness of bot_degree against other agents.

The rest of this report is organised as follows: Section 3 explores in detail the background to this project and researches the relevant area's associated with Monte-Carlo Simulation, UCT, Arimaa and Go. Section 4 talks about the project goals and section 5 details the design of bot_degree and how it has been implemented. Section 6 describes the types of tests that have been carried out and the associated test plans that have been used to test bot_degree thoroughly. Section 7 provides an evaluation of bot_degree, analysing its performance against other Arimaa playing bots, and finally section 8 evaluates how well the project was managed, improvements that could be made and known issues with bot_degree.

Appendix A is provided for completeness and details specific notation that needs to be conformed to when interfacing with the Arimaa server. Appendix B shows the results of the experiment that compares UCT against ϵ-Greedy. Appendices C and D show the initial and final project Gantt Charts respectively, and appendix E provides the test plans and test results that were used to test bot_degree.

Chapter 3

Background

It is thought that Go was played up to 5,000 years ago by the Japanese, Koreans or Chinese. It is played between two people on an 18 by 18 square board (19 by 19 intersections), one of which has white pieces, the other black. Each turn, a player may place one of their pieces on any intersection that is not occupied by another piece. Once a piece has been placed it may not be moved, unless it is captured and removed from the game. The aim of the game is to occupy the most territory, and lose as few pieces as possible. The game ends when both players pass.

At the start of the game there are 361 unique moves that may be played. This high branching factor is why constructing a tree of all the possible moves in the entire game is infeasible, and is why the A.I. community has found it very hard to construct a computer agent to play this game at a competent level. Recently a new computer agent has emerged, called MoGo [5], which was invented by Sylvain Gelly, Yizao Wang, Remi Munos, Olivier Teytaud and Pierre-Arnaud Coquelin. MoGo is currently the best rated computer agent [7] on the Go server and uses a form of Monte-Carlo Simulation and the UCT [5, 4] algorithm to play the game.

3.1 MoGo

MoGo constructs a tree by using a rollout-based algorithm [10] during its turn, and then from the tree picks the next move based on the statistics associated with the tree. Each node represents a board state and contains how many games have been won or lost, from this current state. At the start of each turn, MoGo constructs a new tree with only the current board state as the root. It will then populate the tree with board states, by iterating through the tree between 70,000 to 200,000 times. Each iteration is called an episode and this episodic approach is the aforementioned rollout-based algorithm. The way MoGo performs each episode is in two parts. The first part is traversing the tree from the root to a leaf node using the UCT algorithm. The second is using Monte-Carlo Simulation to simulate a game from the leaf node, (i.e. a particular board state) to the end and then evaluating it.

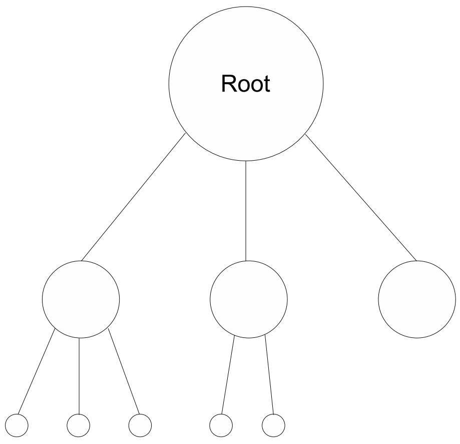
3.1.1 UCT

The UCT algorithm is used to decide how the tree is descended, from the root node to a leaf node using a trade-off between exploration and exploitation [5]. UCT is an extension of the mini-max bandit algorithm, called UCB1 (Upper Confidence Bounds) [10], applied to trees. It treats each board state as a multi-armed bandit problem, with each arm being a legal move to another board state, having unknown reward but of a certain distribution [5]. UCT only considers two types of arms, either winning ones or losing because a draw in Go is very rare. The reason why UCT outperforms other tree searching algorithms, such as Alpha-Beta, as mentioned in [6] is as follows :

1. UCT can be stopped at any time and still produce good results, whereas if Alpha-Beta is stopped prematurely, some child nodes of the root have not even been evaluated yet.
2. UCT handles uncertainty automatically. This means that at each node the value is the mean of each of its children's value, weighted by how many times that child has been visited.
3. It explores more deeply the good moves in an automatic manner. This is achieved by growing the tree asymmetrically

Figure 3.1 is a graphical representation of what the tree may look like during a particular episode. Figure 3.2 shows a possible path of the UCT algorithm to a leaf node.

Figure 3.1: A graphical representation of the tree built during Monte-Carlo Simulation



The root node represents the current board state, all other nodes represent other possible board states. A line drawn between two nodes, A and B, indicates a legal move that transforms the board state represented by node A into the board state represented by node B.

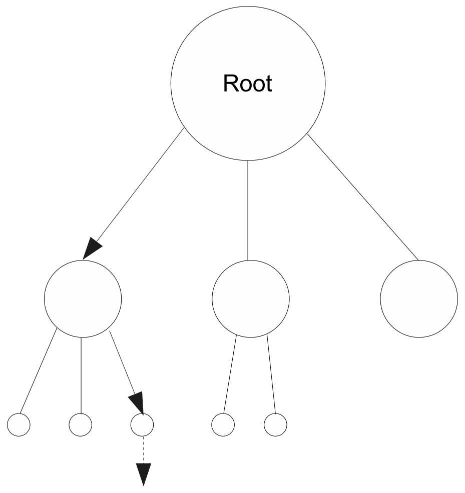
The UCT algorithm, that was first introduced in [10], chooses the best arm to play next based on equation 3.1, where I_t is the index of the selected arm that maximises the equation at episode t, t is the current episode and i is the index of an arm. The function $T_i(t-1)$ is the number of times arm i has been selected up to episode $t-1$. Therefore $\bar{X}_{i,T_i(t-1)}$ represents the average payoff that arm i has produced, and can be found by summing through all the payoffs that arm i has produced up to episode $t-1$ and dividing by $T_i(t-1)$. In practice however, it is computationally expensive to continually iterate through all the payoffs of a certain arm, and therefore the total payoff of a particular arm is held instead.

$$I_t = \underset{i \in \{1,\cdots,K\}}{\text{argmax}} \{\bar{X}_{i,T_i(t-1)} + bias_{t-1,T_i(t-1)}\} \tag{3.1}$$

The bias function, equation 3.2, provides a way of trading between exploration and exploitation. If an arm has not been selected much the bias becomes very big and biases the UCT algorithm towards the exploration strategy. If an arm has been selected many times the bias is small and reduces the chance of it being selected. In order to show the effectiveness of the UCT algorithm, an experiment was constructed in which the UCT algorithm was compared with an ϵ-Greedy algorithm, details of which are located in Appendix B.

$$bias_{t,c} = \sqrt{\frac{2 \ln t}{c}} \tag{3.2}$$

Figure 3.2: A graphical representation of the tree built during Monte-Carlo Simulation and the particular path the UCT algorithm may take



The arrows show the path of the UCT algorithm. The dashed arrow indicates that the Monte-Carlo Simulation will now be performed.

3.1.2 Monte-Carlo Simulation

Once the UCT algorithm reaches a leaf, the Monte-Carlo Simulation takes over. The first step is to choose a new move, m, to play based on an evaluation function from the current board state, c. A new board state, n, which corresponds to applying the move m to the board state c, is then added to the game tree with its total payoff and number of times selected initialised to 0 and 1 respectively. The board state n is then evaluated, so to do this, MoGo plays a random game against itself from n to the end. The Monte-Carlo method tries to simulate the most probable moves that would be played during this random game. When the end of the random game is reached, the payoff is 1, if m resulted in a win, or 0 otherwise. It should be noted that draws are not taken into account as they happen very rarely in Go. Each node that was visited from the root to n is updated by adding the new payoff to its total payoff and incrementing the number of times its been selected by 1. The next episode commences, starting from the root of the tree.

Once many episodes have taken place, the child node which maximises the UCT algorithm is chosen. The UCT algorithm takes care of the exploration-exploitation trade-off and thus if a node has been simulated more, it means that the expected reward from playing a move to reach that node is higher and thus a better move to play. The aim of this project is to apply these techniques of Monte-Carlo Simulation and the UCT algorithm in implementing a computer agent to play the board game Arimaa [14].

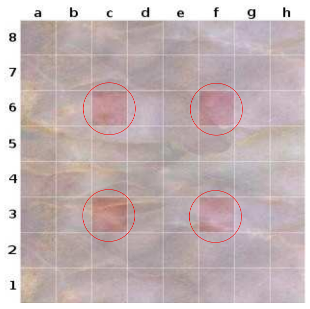
3.2 Arimaa

Arimaa was invented in 1999 by Omar Syed and Aamir Syed. It was invented to be specifically difficult for computers to play and relatively easy for humans to play. The idea was that this game would create a challenge for the artificial intelligence community so that new and exciting algorithms and techniques were invented for playing games. Arimaa is a 2-player, zero-sum, perfect information board game [2]. Zero-sum means that if one player is winning, the other player is equally losing; their scores sum to zero [12]. A perfect game is one where only one person moves at a time and both players know exactly where the other player moved.

3.3 Arimaa Rules

The rules of Arimaa [14] are simple and intuitive to play and this is why humans can play the game so well compared to computers. Arimaa can be played on a standard chess board (8 by 8 squared board) with standard chess pieces. The board is annotated with letters and numbers so that each square can be accessed. Lower case letters a to h indicate the column and numbers 1 to 8 indicate the row. There are four trap squares that are situated on c3, f3, c6 and f6, see figure 3.3. Table 3.1 contains information on the Arimaa pieces and their Chess alternatives.

Figure 3.3: The location of the four trap squares on the Arimaa playing board



The traps are circled in red. Picture adapted from [16]

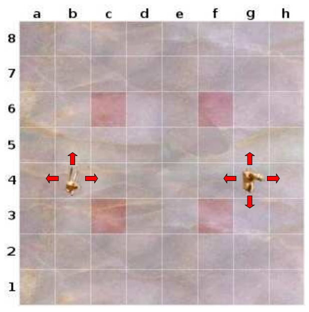
Table 3.1: The pieces used in Arimaa and the chess equivalent

Arimaa Piece	Arimaa Picture	Chess Equivalent	Quantity
Elephant		King	1
Camel		Queen	1
Horse		Rook	2
Dog		Bishop	2
Cat		Knight	2
Rabbit		Pawn	8

Pictures used from [3]

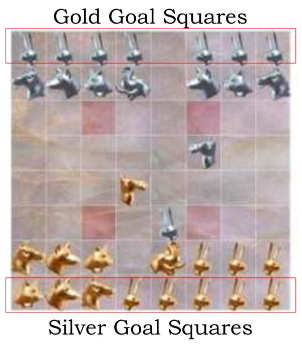
The order of strongest to weakest is as follows : Elephant, Camel, Horse, Dog, Cat and finally Rabbit. An explanation of what it means to be stronger or weaker will be given in due course. Each piece in Arimaa moves in exactly the same way; they can move one square north, south, east or west, however Rabbits may not move south, see figure 3.4. The goal of Arimaa is to get your Rabbit to the other side of the board, see figure 3.5.

Figure 3.4: The method of moving in Arimaa



Picture adapted from [16]

Figure 3.5: The goal squares in Arimaa



Picture adapted from [17]

During a player's turn, they may move up to four steps. One step counts as moving a piece to an adjacent square (Note that diagonals do not count as adjacent squares). Multiple pieces may be moved during a player's turn and the direction of a piece may be changed. The strength of a piece determines what that piece can and cannot do. There are three concepts in Arimaa called *pulling, pushing* and *freezing,* which are dependent on the strengths of the pieces involved.

3.3.1 Pushing And Pulling

Pushing or pulling happens between two pieces, where one of the pieces is stronger then the other. For instance a Dog may push or pull a Cat or Rabbit, but not a Horse, Camel, Elephant or another Dog. To push a weaker enemy piece with your stronger piece, first the stronger piece must be adjacent to the weaker enemy piece. The weaker piece is then moved into one of its unoccupied adjacent squares and the stronger piece is moved into the square where the enemy piece was. To pull a weaker enemy piece with your stronger piece, first the stronger piece must be adjacent to the weaker piece. The stronger piece is then moved into one of its unoccupied adjacent squares and the weaker piece is moved into the square where the stronger piece was, see figure 3.6.

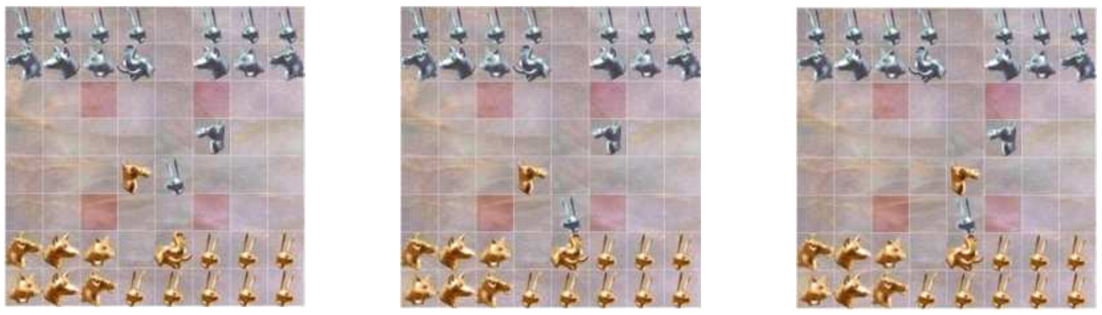
Pushing or pulling requires two steps and must be completed within the same turn. While a stronger piece is pushing it may not pull at the same time and vice versa. Pushing and pulling can be done a total of twice each turn because of the restriction of four steps per turn.

Figure 3.6: Gold performing a push on Silver

(a) The Gold Camel is positioned next to the Silver Rabbit

(b) The Gold Camel pushes the Silver Rabbit south

(c) The Gold Camel moves east into the Silver Rabbit's place



Picture adapted from [17]

3.3.2 Freezing

Freezing also happens between a stronger piece and at least one other weaker piece, (i.e. a stronger piece may freeze multiple weaker pieces). To freeze a weaker enemy piece with your stronger piece, just move your stronger piece next to the enemy's weaker piece. That enemy piece is now frozen and may not be moved until either the stronger piece moves away or another friendly piece is adjacent to it. In figure 3.6a, the Silver Rabbit is frozen and cannot move because the Gold Camel is adjacent to it.

3.3.3 Trapping

If any piece is moved onto one of the trap squares they are immediately removed from the game. Pieces can be pushed or pulled onto trap squares. If a piece is on a trap square but has a friendly piece adjacent to it, then the piece in the trap square is not removed from the game. See Appendix A for further rules that apply to making a computer agent to play Arimaa on the Arimaa server.

3.3.4 Hard For Computers

Arimaa is hard for computers to play because of many different factors that were specifically taken into account when Arimaa was invented. The main problem is the amount of legal moves that can be played from a given board state, otherwise know as the branching factor. A board state can have as many as 300,000 four step move sequences [3] however this contains different permutations of the same move and thus a typical board state can have within the region of 20,000 to 30,000 unique four step moves. The way Deep Blue calculated the next move to play during chess was to construct a tree of all the possible different moves, choosing the best move from the tree, known as the brute force approach.

The number of nodes in the search tree is affected by the size of the branching factor [18], and since this is very high in Arimaa it is infeasible to try to construct a tree of all possible moves. Equation 3.3 represents the size of the search tree at depth d, branching factor b, and grows exponentially. As an example of how infeasible it is to construct a tree of all possible moves in Arimaa, a comparison has been made between the average branching factors of Go and Chess [9], see Table 3.2. In each case it is assumed that a 5 move look ahead is required and that the computer searching the tree can evaluate 1,000 nodes a second.

$$b^d \tag{3.3}$$

Table 3.2: The contrast between the branching factors of Go, Arimaa and Chess

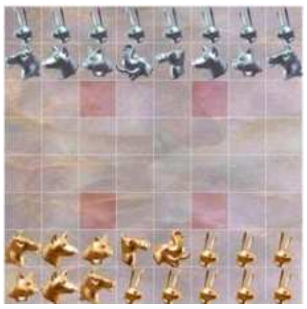
Game	Branching Factor	Time Taken to Calculate (years)
Arimaa	$25,000^5 = 9.765625 \times 10^{15}$	3.1×10^{11}
Go	$200^5 = 3.2 \times 10^{11}$	10.1
Chess	$30^5 = 2.43 \times 10^7$	7.7^{-4}

Chess has the smallest branch factor and is why Deep Blue was so effective against Kasparov [8]. All Deep Blue did was to construct a massive search tree of every possible move and choose the best one based on an evaluation function. However, Arimaa has the greatest branching factor and thus, a 5 move look ahead of every possible move in Arimaa is infeasible. In the example, a computer that can perform 1,000 evaluations per second has been used. Obviously faster computers exist, but for Arimaa even with the fastest computer in the world it would still take hundreds of thousands of years to find the best move in the tree. That is why different tree building and search techniques must be explored.

Arimaa is a game of strategy, not tactics and is why computers find it difficult to play. Since capturing an enemy's piece involves pushing them into a trap, an attack scenario might take three or four moves to set up, thus in order for the computer agent to anticipate this it must look ahead. As the example above showed, this is infeasible for brute force methods. This slow moving game means that a player may escape multiple threats on its pieces and so a strategic avoidance of a capture in one area of the board, may affect the other side of the board 20 moves later.

Starting positions must also be considered as this further complicates computers ability to play effectively. To set up the game, Gold goes first and places their pieces, in any order, on the first two rows closest to them. Silver may then observe where Gold has placed their stronger and weaker pieces and act accordingly. Figure 3.7 shows a possible starting position for Gold and Silver. The fact that no two starting positions are likely to be the same for any two games means that a database of starting positions is infeasible. End games are similar because often there are still many pieces on the board and thus constructing an end game database is also infeasible.

Figure 3.7: Possible starting positions for Gold and Silver



Picture used from [17]

3.3.5 Easy For Humans

Despite there being a very large branching factor for Arimaa, humans are not affected as much because they use knowledge of past games to decide which move should best be played [18]. Arimaa's rules are intuitive and easy to learn and thus humans can play at a reasonable standard quicker then a computer can. There is no need to memorise end games or start games therefore humans can concentrate on playing each game as it comes, instead of knowing predefined moves to be played.

Chapter 4

Project Goals

The goal of this project is to implement the UCT algorithm and Monte-Carlo Simulation in an agent, called bot_degree, which will play Arimaa. The UCT algorithm will be fine tuned to perform more efficiently, once bot_degree is running, by changing the bias. The ultimate goal is to beat bot_bomb at least once. This would be a great achievement as bot_bomb is the best Arimaa playing agent on the Arimaa server [14] at present. The rest of this chapter is concerned with the requirements that this project will fulfil.

4.1 Functional Requirements

The following list describes the main functional requirements of the project, with the most important at the top of the list.

1 Read the current board state using the notation described in Appendix A.1 and the techniques described in Appendix A.4.

2 Output a legal move to be played based on the notation described in Appendix A.1 and the techniques described in Appendix A.4.

3 Choosing a move via the UCT algorithm must consist of building a tree of selected moves that have been evaluated using Monte-Carlo Simulation.

4.2 Non-Functional Requirements

The following list describes the main non functional requirements of the project, with the most important at the top of the list.

5 Must be written in C/C++ using pointers and preallocation of memory where ever possible in order to increase the efficiency of the program.

6 When playing on the Arimaa server, bot_degree must make a move within the timing limits of the game as described in Section A.3. A move must be made in less then T seconds or (M + R) seconds, whichever is less, otherwise will lose automatically.

7 Perform at least 10,000 simulations while still conforming to the specified time constraints.

4.3 The MoSCoW Approach

The following are the requirements of the project in more detail and priority.

4.3.1 Must Have

1 A working move function that produces a sequence of legal steps (i.e. one move) within the specified time constraints.

2 Ability to interface with the Arimaa server so that it can play against other bots.

3 An evaluation function that uses the techniques of MoGo.

4.3.2 Should Have

4 Win consistently against a program that chooses a random move

5 Win against lower ranked bots, that exist on the Arimaa server ranking system, with more then 50% success rate.

4.3.3 Could Have

6 Win against middle ranked bots, that exist on the Arimaa server ranking system, with more then 50% success rate.

7 Win one game against bot_bomb

4.3.4 Would Like

8 Use some form of rule to decide where to place each piece based on the other player (only when Silver).

9 Win against bot_bomb with more then 50% success rate.

Chapter 5

Agent Design

The Arimaa website [14] provides much information as to how an agent can be implemented to play the game of Arimaa. It provides a set of sample bots that can be used as a starting point and a set of scripts that can be used to test the bots offline without connecting to the Arimaa server. The bot r0, created by Don Dailey [14], is a random Arimaa playing bot that uniformly chooses a random move from all the unique moves that exist given a particular board position. This was used as a starting point for bot_degree. To test bot_degree, r0 has been utilised using the offline Perl scripts.

The bot r0 contains a main function which is used by the offline Perl scripts and the Arimaa server to run the bot. The function that produces a move for r0, called “begin_search” was replaced by a function called “move” that contains the Monte-Carlo Simulation and the UCT evaluation that bot_degree uses. In order to implement Monte-Carlo simulation, bot_degree has to calculate random moves from a given board position. This involved using and modifying the existing random move function within r0, called “root”. The function “root”, calculates all possible moves that exist, from a given board position that is passed into the function, and then chooses one of them. If the random move chosen is unique then “root” returns the move, or else it randomly selects another move.

Each time it is bot_degree’s turn to move, it receives a board position from which it will make a move. To do this, bot_degree creates a new tree structure and makes this board position the root of the tree. Subsequently, bot_degree then builds the tree as illustrated in figure 5.1. Lines 3-5 descends through the tree, starting at the root, until the UCT algorithm is satisfied either by reaching a leaf node or by deciding to explore a particular Node within the tree. This decision of whether to descend another level or to explore the current node, was initially random, i.e. there was a 50% chance that the algorithm would descend to another level. However it was decided that a more “intelligent” way of descending the tree was required and thus a decay function was implemented that initially does more exploration then exploitation. See section 7 for an explanation of the decay function as well the tests that were carried out to find the optimal variable values. Once line 3-5 have completed, d contains the node that the UCT algorithm stopped at during the descent of the tree.

Line 7 chooses a random move from d’s board state and then adds it to the tree, line 8. Line 9 calculates the payoff of d by randomly simulating a game from d’s board position to the end. The payoff is either 1, if d’s board state resulted in a win, or 0 otherwise. The calculated payoff is then added to each node’s payoff that was visited during the descent through the tree, and the number of times each node was visited is incremented by 1, line 10. Lines 2-11 are repeated until the number of simulations required has been reached. The child node of the tree’s root that satisfies the UCT algorithm is selected and its move is returned by “move”. To construct the tree that will be used during the “move” function, a class called “core::tree”, which is available from

[13], was used. It uses the notion of iterators to traverse and manipulate the tree, which is highly appropriate for bot_degree as fast and efficient traversal of the game tree is desired.

Figure 5.1: Pseudo code of how bot_degree will build the game tree

```
nEpisodes := # of simulations to perform
for i := 1 to nEpisodes do
        repeat
                d := descendUCT;
        until (UCT algorithm is satisfied)

        m := createRandomMove(d);
        tree.add(m);
        payoff := simulate(n);
        updateVisitedNodes(payoff);
end for

move := child of the root that satisfies the UCT algorithm;
return move;
```

Figure 5.2 describes the detailed design structure of bot_degree in the form of a UML Class diagram. A description of the design for each class along with its functions and variables are as follows:

Node Contains all the data associated with each node of the tree structure. During each episode, the program descends through the tree evaluating the nodes based on the UCT algorithm, section 3.1.1, and eventually will create a new node which is added to the game tree.

boardPosition The variable that describes everything associated with the board position that this node is representing. Amongst other data, it contains a 2-Dimensional array of 14 bit boards which represent all the pieces on the board. The bit boards are indexed by colour (0 for Gold, 1 for Silver) and then by type (0 for empty squares, 1 for rabbits, 2 for Cats, 3 for Dogs, 4 for Horses, 5 for Camels and 6 for Elephants).

main_line The variable that describes the sequence of steps that were applied to the parent of this nodes board position that results in this nodes board position. When the simulations have finished, it is this variable that is returned by the GetMove class.

count The variable that describes how many times this node has been visited during the traversal of the tree structure.

payoff The variable that describes the total payoff that this node has experienced during the simulations.

Tree Contains all the functions and variables that are used to construct a tree structure. The game tree is constructed during the simulations, and is traversed by the UCT algorithm.

nodes_array The array of nodes that is used to construct a tree structure.

insert() The function used to insert nodes into the tree structure.

GetMove Contains all the functions and variables that are used to select a move given a particular board state.

visited_nodes The array of nodes that have been visited through the descent of the tree structure. This is needed because the payoff and count of each node within this array must be updated after each simulation.

tree The tree variable that is used to construct a tree structure of nodes.

main() The function that is used by the offline Perl scripts and the Arimaa server to run bot_degree. Within this function move() is called.

descendUCT() The function that returns the child node of a node within the tree structure that maximises the UCT algorithm. Within this function chooseUCT() is called to calculate the UCT value for each child node.

chooseUCT() The function that actually calculates the UCT value for each child node of a node within the tree structure. This function returns the number of the child that maximises the UCT algorithm, (i.e. if 1 is returned, it indicates that the first child of the node maximises the UCT algorithm)

updateNodes() The function that updates the payoff and count of every node within the visited_nodes array. The payoff is calculated by the simulate function, count is simply incremented by 1.

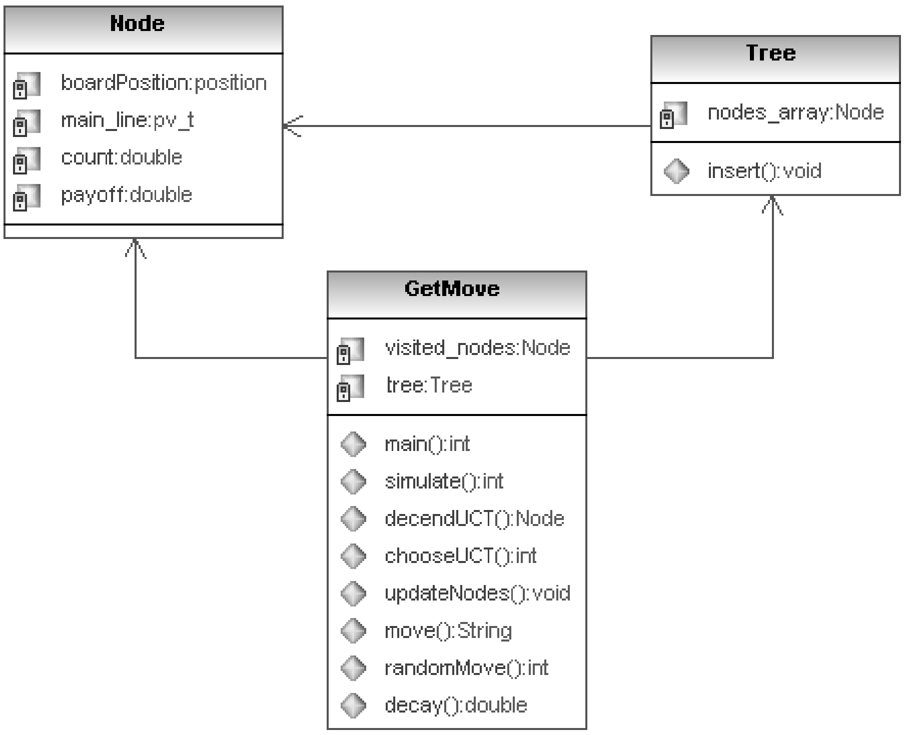
simulate() The function that simulates a random game from the current board state, passed into the function, to the end. The payoff of the random game is then evaluated as either 1 or 0, depending on which side wins, and returned.

move() The function that receives a board position, constructs and builds the tree and then returns the chosen move that maximises the UCT algorithm. Within this function descendUCT(), chooseUCT(), updateNodes() and simulate() are called.

randomMove() The function that receives a board position and returns a random legal move. This function is a modification of the existing random move function that exists within r0.

decay() Determines whether the UCT algorithm explores or exploits based on the current episode.

Figure 5.2: The UML class diagram for the design of bot_degree



Chapter 6

Testing

This chapter details the techniques used in order to test bot_degree.

6.1 Design

A number of test cases were designed in order to test the robustness, functionality and reliability of bot_degree. See Appendix E tables E.1 - E.7 for a detailed list of test cases, and tables E.8 - E.14 for the results. Many different types of test strategies were used in order to test bot_degree thoroughly :

Unit Testing Was used to test the Node class and the getMove class

Stub/Driver Testing Was used to test the individual functions which were written for bot_degree. These include:

- simulate
- descendUCT
- chooseUCT
- updateNodes
- move
- randomMove
- decay

White Box Testing Was used to particularly test the descent from the root of the tree to a leaf, using branch and path testing, making sure that all paths through the code are tested properly.

Performance Testing Was used to test bot_degree over an extended period of time while recording statistical information, such as number of wins, loses, moves per game and how many simulations where performed.

In order to run each test, the Southampton Linux cluster, called Lyceum, was used. This cluster consists of 128 processors split into 16 nodes and provides the ability to run multiple tests in parallel, greatly reducing the length of time necessary to complete all the tests. This was particularly useful when running the performance tests, as a great number of games was required to be played over an extended period of time, with some games taking as much as 3 hours to play.

Chapter 7

Agent Evaluation

An evaluation of bot_degree was carried out which involved various tests being completed. Section 7.1 details the decay function used by the UCT algorithm, while section 7.2 describes the performance tests that were carried out.

7.1 The Decay Function

As described in section 5, bot_degree uses a decaying function that decides the trade-off between exploration and exploitation. The objective of this decay function was to provide more exploration when the number of episodes completed was small, by expanding the game tree near the root, and then exploiting the vast amount of nodes within the game tree when the number of episodes completed was large. The reason for using a decay function is because the UCT algorithm works best when many nodes near the root of the tree have been explored at least once [6], and therefore will converge more effectively towards the "best" move for the given board position.

Equation 7.1 shows the decay function which has been used in bot_degree, where V is equal to 0.7 and d is the decay, in the interval of [0,1], based on the current episode, f, and the total number of simulations that will be used, s. During the UCT algorithm's descent through the game tree, bot_degree will produce a random number and compare with d. If the random number is smaller then d, then the UCT algorithm will descend one level through the tree to the node that maximises the UCT algorithm, otherwise bot_degree will calculate a random move from the current node, add it to the tree and simulate a random game to the end, as explained in section 5. Therefore when the number of episodes completed is very small, d will be small, and thus will drive the UCT algorithm towards exploration, instead of exploitation, of nodes.

$$d = \exp(\frac{-1}{s \times V} \times (s - f)) \tag{7.1}$$

A set of tests were designed in order to find the optimal value for V. Each test involved bot_degree playing a number of games against r0 with 1,000 simulations per move and varying values of V. The standard error[1] of the results was calculated, see table 7.1. The most pessimistic percentage win when V is 0.7 using the 95% confidence interval is 74%. Similarly the most optimistic percentage win when V is 0.2 using the 95% confidence interval is 73.9%. Therefore even with this worst

[1]

$$Standard\ Error = \sqrt{\frac{p \times (1 - p)}{N}} \tag{7.2}$$

Equation 7.2 was used to calculate the standard error where p is the percentage of bot_degree wins and N is the number of repeated tests carried out for the given number of simulations, which did not produce a bad position. The standard error can be used to calculate the 95% confidence interval which gives an estimator of how much the percentage of bot_degree wins fluctuates. The 95% confidence interval is simply twice the standard error.

case scenario, the results show that the optimal value for V is 0.7. When V is nearer to 0.9, the UCT algorithm performs more exploitation by descending more levels in the tree, whereas when V is nearer to 0.2, the UCT algorithm is more likely to explore by adding new random moves to the game tree. Interestingly enough it seems that the UCT algorithm works well if it is either more inclined to explore new moves or exploit existing ones, but behaves very poorly if it tries to have an even balance of both. Therefore using V equal to 0.7 means that initially bot_degree will perform exploration of moves but will have a small bias towards exploiting moves already in the game tree. This bias of exploitation will increase as the number of episodes completed increases and will end with bot_degree almost exclusively exploiting moves within the game tree.

Table 7.1: The standard error of bot_degree wins against the random bot r0 with varying values for V

Value of V	Number Of Games Played	Percentage Of Bot_degree Wins	Standard Percentage Error	95% Confidence Interval
0.9	67	46.3%	6.1%	± 12.2%
0.8	68	57.4%	6.0%	± 12.0%
0.7	75	82.7%	4.4%	± 8.7%
0.6	85	21.2%	4.4%	± 8.9%
0.5	84	23.9%	4.6%	± 9.3%
0.4	85	24.7%	4.7%	± 9.4%
0.3	87	33.3%	5.1%	± 10.1%
0.2	88	63.6%	5.1%	± 10.3%

7.2 Performance Tests

Once the decay function had been implemented, a series of tests were carried out to see how bot_degree performs against different Arimaa playing bots. When each test was carried out, whenever bot_degree produced a "bad step" error, this test was not taken into account. See section 8.3 for an explanation of this error. The majority of the tests involved bot_degree playing against the random bot r0 with varying number of simulations, see table 7.2. The table shows that as the number of simulations that bot_degree uses increases, the percentage that bot_degree wins by increases as well. When simulating just 10, 50 or 100 times, bot_degree behaves randomly because the UCT algorithm does not sample the nodes within the game tree enough and therefore does not have enough statistics to find the optimal move. As bot_degree increases the number of simulations above 500, it can be seen that the percentage of wins rises considerably. This is because many moves have been explored within the game tree and thus the UCT Algorithm has enough statistics to descend the tree effectively and choose the "best" move.

Interestingly, 1,000 simulations per move produced the best results whereas 50,000 simulations produced the worst results for tests that used simulations above 1,000. One reason for this is the results of the tests were effected by bot_degree's known issue of producing a bad position around 21% of the time. The outcome of the 21% of games that resulted in a bad position could have favoured either Arimaa bots. For instance it may be the case that whenever bot_degree produced the error, r0 was about to win, thus biasing the results towards bot_degree. This could have been the case with the test results when 1,000 simulations per move was used, however it is impossible to tell, and only speculations can be made. See section 8.2 for possible ways in which this error could be fixed.

Table 7.2: The standard error of bot_degree wins against r0 with varying simulations

Number Of Simulations Per Move	Number Of Games Played	Percentage Of Bot_degree Wins	Standard Percentage Error	95% Confidence Interval
10	672	51.9%	1.9%	± 3.9%
50	611	47.8%	2.0%	± 4.0%
100	603	47.4%	2.0%	± 4.1%
500	587	64.6%	2.0%	± 3.9%
1,000	570	75.4%	1.8%	± 3.6%
5,000	533	70.9%	2.0%	± 3.9%
10,000	597	73.2%	1.8%	± 3.6%
50,000	233	68.2%	3.1%	± 6.2%

Another reason for 1,000 simulations per move producing the best results could be because an equal number of games for each test was not carried out. The explanation for this is that even though the intentions of the testing was to have bot_degree play 800 games for each different number of simulations, the use of the Linux cluster had timing constraints therefore making it very hard to run tests for long periods of time, especially when some games took longer then 3 hours each. Furthermore since bot_degree produces the bad position error, it was very difficult to get the same number of tests completed for varying number of simulations.

It can be observed that from 50 simulations to 500 simulations, there is a 16.8% increase in bot_degree wins. The most optimistic percentage of wins using the confidence interval for 50 simulations is a win percentage of 51.8%. Similarly the most pessimistic percentage of wins for 500 simulations is 60.7%. However this still produces an increase in win percentage of 8.9% proving that increasing the number of simulations for bot_degree increases the percentage of bot_degree wins. This is not the case for 500 to 5,000 simulations as using the most optimistic percentage win for 500 simulations and the most pessimistic percentage win for 5,000 simulations does not produce an increase in bot_degree wins and actually produces a decrease of 1.5%.

The difference between the percentage wins above 1,000 simulations is too close together to make the assumption that as the number of simulations above 1,000 is increased, the number of bot_degree wins increases. This is evidently due to noise and can be particularly noted in the 50,000 simulations test as the confidence interval is approximately 2% higher then the rest. In order to produce more meaningful results, many more tests would have to be carried out, thus reducing the confidence interval. However even with the current set of tests, it can still be seen that as the number of simulations per move is increased, the number of bot_degree wins increases. This proves that using UCT and Monte-Carlo Simulation is a viable starting point to implementing an Arimaa playing agent that can play the game of Arimaa well.

Other tests were carried out on bot_degree using more “intelligent” bots as its opponent. The bots that bot_degree played against were called “bot_SampleC” and “bot_fairy” which are both available from the Arimaa website [14]. Both these bots use an evaluation technique of playing the game which involves calculating every possible move from a given board position and then returning the “best” move based on some evaluation heuristic. In each test bot_degree used 1,000 simulations per move and the tests were run 400 times. The results of the test showed that bot_degree won once in 400 games, with a percentage win of 0.29%, against bot_SampleC, and did not win at all against bot_fairy. Another test was carried out against bot_SampleC with bot_degree using 20,000 simulations per move. This test was run 364 times and resulted in bot_degree winning none of the games.

The one time when bot_degree won against bot_SampleC was obviously random since all the other tests showed that bot_degree lost. The reason why bot_degree lost almost all the games played against these evaluation bots is most likely to be because bot_degree performs no evaluation of the moves that it calculates. Therefore moves that for instance trap its own pieces or help the opponent are used within the game tree, regardless of how bad the move is. As proved in [4], the UCT algorithm benefits from an increased number of Monte-Carlo Simulations, therefore bot_degree may not be simulating enough games for the UCT algorithm to choose the “best” move. See section 8.2 for possible ways in which bot_degree’s performance can be improved. It was decided that bot_degree would not be tested against the best Arimaa playing agent bot_bomb, contrary to the project requirements, because bot_degree performed so poorly against other agents that evaluate a position and thus would have been pointless.

As expected, when the number of simulations per move increases, the time taken for bot_degree to complete a move increases as well. This is not only because bot_degree has to simulate more games per move, but also because it has to build and traverse a bigger game tree each episode. Therefore the greater the number of simulations per move, the more time bot_degree must spend calculating the UCT algorithm for each node at each level. See table 7.3 for the results of running bot_degree against the bot r0 with varying simulations per move. It should be noted that games that produced a bad position were also factored into this table.

Table 7.3: The time taken for bot_degree to play a game against the random bot r0 with varying simulations

Number Of Simulations Per Move	Number Of Games Played	Time Taken (h:m:s)
10	768	0:03:40
50	799	0:01:58
100	783	0:02:18
500	800	0:02:52
1,000	800	0:04:28
5,000	726	0:17:16
10,000	799	0:34:31
50,000	321	3:35:17

The time taken for r0 to produce a move does have a slight effect on the game length, but this is very small and is considered insignificant towards the overall result. The general trend of the results do show that as the number of simulations per move increases, the time taken for bot_degree to play a whole game is increased. However there does appear to be an anomaly within the results, for the tests when the number of simulations was 10. The explanation for this could be that during that time, the Linux cluster, on which the tests were run, was slower then usual because of a high number of jobs being computed and thus processed the tests slightly slower then normal. As bot_degree is very slow at playing a game above 10,000 simulations, section 8.2 describes ways in which bot_degree could be optimised to run faster and perform more simulations per move.

Chapter 8

Conclusions

In this chapter, the project as a whole is evaluated and improvements suggested along with the errors that exist within bot_degree and possible ways to fix them.

8.1 Project Evaluation

A commentary was kept by the author so that the evaluation of the project could be easily made. The following sections describe how the project was managed as well as the project goals that were fulfilled.

8.1.1 Project Management

Initially, a lot of research was made in the area's of Chess, Go and Arimaa, see section 3, in order to prepare for the design, implementation and testing of bot_degree. Right from the beginning it was decided that C/C++ should be used instead of Java because of its low level capabilities and efficiency when using pointers. This was a good choice because the code needed to be as fast as possible due to the high amount of simulations required by Monte-Carlo Simulation and the UCT algorithm. Regular meetings were held with the project supervisor, Dr. Alex Rogers, to discuss the direction of the project and any major issues that were happening. The UCT algorithm was implemented in Matlab first, see Appendix B, to test the effectiveness of it against another tree traversal algorithm. The reason it was tested first was because it would then be easier to implement in C/C++ having explored its possible problems and successes.

A Gantt chart was drawn up, see Appendix C, which was the predicted work load for the entire project. This was stuck to as much as possible but after continued work on the project it became clear that some areas would take longer then others. For instance even though in the initial Gantt chart the authors exam commitments had been factored in, it was believed that the project could still be implemented and tested during this time. This was however not the case because all the author's time and effort went into revising for exams. This did not affect the project at all because most of the project had already been completed; much of the implementation had been completed, many sections of this report had been written already and thorough continuous debugging of bot_degree's code had been employed. This is reflected in the final Gantt chart, see Appendix D, which shows a gap in all aspects of the project during exam time. Implementation of bot_degree took longer then expected because of unexpected errors within the code. One such error was that during the execution of the code it would randomly terminate sometimes when using the game tree.

After many days of checking board states, variables and functions that all used the game tree it was discovered that at the start of each episode, the game tree iterator was not being initialised to the root of the tree and thus during the descent through the tree, the iterator would sometimes iterate past the last child on the tree and would thus terminate bot_degree prematurely. This was fixed by initialising the tree iterator to the root of the tree at the start of each episode and was tested by checking the board states, variables and functions that used the game tree iterator. Since implementation was taking longer then expected, the project goals had to be prioritised. Therefore it was decided to focus the attention of the project on getting bot_degree to produce a move, and not try to conform to the Arimaa timing constraints imposed on the Arimaa server, (see non-functional requirement 6 in section 4.2). However there is still the possibility of modifying bot_degree to be able to play on the Arimaa server and this is discussed in section 8.2. Initially it was intended that bot_degree would try to use a database of good starting positions when setting up its pieces for the start of a game. If it played as Gold, (i.e. bot_degree would set up after Silver had positioned all its pieces), then it would try to use the position of Silvers pieces to set up a strong starting position.

However after much research using the Arimaa website [14], and papers [18, 3] it was concluded that using a starting position database is almost impossible since there are over 16 million different possible starting positions to choose from. See equation 8.1 which shows how many unique starting positions are possible, where 16 is the number of squares to set up on, and 6 is the number of different types of pieces that exist in Arimaa. Thus given the timing constraints posed by this project and the fact that implementation of bot_degree was taking longer then expected, no further research was considered for a starting positions database.

$$16^6 = 16,777,216 \tag{8.1}$$

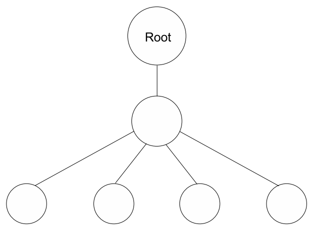
Initial testing of bot_degree with varying number of simulations produced some poor results. With as little as 100 simulations per move, it was taking bot_degree approximately 45 minutes to complete a game. This was unacceptable as the UCT algorithm needed many more simulations per move in order to be effective at choosing the "best" move. Therefore the random move function was modified and optimised so that it executed faster. This was achieved by calculating a select number of random moves and choosing one at random instead of calculating all possible random moves and selecting uniformly from them. Additionally, the random move function was modified not to check whether a chosen move was unique, as this slowed the original random move function immensely. These modifications have sacrificed true unified randomness for speed, however it was deemed unlikely that this would make a significant difference. After the modifications, bot_degree could complete a 100 simulations per move game in as little as 1 minute.

Up until this point it was assumed that bot_degree was building the game tree properly. However after initial testing of bot_degree, against the random bot r0, with increasing number of simulations, the number of times that bot_degree won did not increase, which was contrary to what was expected. This was investigated by checking the number of children of the root, just before bot_degree returned a move. It was found that in fact only one child of the root existed in all cases, see figure 8.1, and thus the UCT algorithm was always choosing the only child of the root. This problem had something to do with the way the tree class was being used. If a node was being added as a child of the root, using the tree iterator to do this produced the aforementioned error, thus a conditional statement was added which made sure that if a node was being added as the child of the root, the actual tree object was used instead of the tree iterator.

This fixed the problem and the tree grew in the appropriate manner from then on, however bot_degree now had 2 more problems associated with producing a move, they were:

1. Sometimes bot_degree would choose a move that did not change the board state, and thus lost the game by default. For example the following move, `Ha2n Ha3s Rf3w Re3e`, leaves the board in the same state as before this move was made, it simply moves the Horse to an adjacent square, moves it back again and then does a similar thing with the Rabbit.

Figure 8.1: The state of the game tree being built incorrectly



For the UCT algorithm to work, it needs to have many children of the root to evaluate. With only one child of the root, the UCT algorithm was always choosing this node and thus was not making any difference to the outcome of the game, making bot_degree act random.

2. Sometimes bot_degree would choose a move that was wrong. For example it would try to move pieces from squares that they did not occupy or introduce entirely new pieces so that an illegal amount of the same type of piece now existed on the board. These "bad moves" would result in bot_degree losing by default.

Both these errors were not found earlier because of the problem with the way the tree was being built. Since the root only had one child, and thus was the move chosen by the UCT algorithm, it meant that it was highly unlikely that the one child of the root would be either a move that did not change the position or a bad move and therefore went undetected. The reason bot_degree would sometimes produce a move that did not change the state of the board was because the random move function did nothing to discriminate the select number of moves it produced. This error was overcome by testing for a move that did not change position during the random move function. If the position did not change, bot_degree would reduce the move to one step and then calculate a select number of random steps to produce a 4 step move. It would then attempt to pick a move at random until the move changed the state of the board. The bad move error is still present in bot_degree, however it has been reduced in frequency, see section 8.3 for more details.

A lot of software testing was performed on bot_degree to ensure that there were as few errors as possible, see appendix E. Performance testing was also carried out in order to evaluate how good bot_degree is at playing Arimaa, see section 7 which discusses the results of bot_degree's evaluation against other Arimaa playing Agents.

8.1.2 Project Goals

During the project the main goal was to get bot_degree to be able to play a game of Arimaa against another player using the UCT algorithm and Monte-Carlo Simulation. This main objective was achieved along with many of the other project requirements. All of the functional requirements, see section 4.1, and non-functional requirement 5, see section 4.2, were achieved. As explained in section 8.1.1, non-functional requirement 6, which involved conforming to the Arimaa server timing constraints, was not implemented because it was decided not to be feasible in the amount of allocated time for the project. However non-functional requirement 7, was partially completed as bot_degree can perform 10,000 simulations per move, with each game taking on average 35 minutes to complete.

The requirements of the project were prioritised in section 4.3 using the Moscow approach. Requirement 1 was almost completely satisfied, however occasionally the random move generator produces a move that does not exist and thus loses the game. This is a known issue, see section 8.3 for an explanation of the issue along with possible ways to resolve it. Requirement 2 has been achieved because bot_degree has been constructed from an existing Arimaa playing bot and thus conforms to the interface requirements of the Arimaa server, see section A.4. However, bot_degree does not consider timing constraints imposed by the Arimaa server, as explained in section 8.1.1, and thus will lose automatically if it runs out of time making a move.

Requirement 3 has been completely satisfied because both Monte-Carlo Simulation and the UCT algorithm have been implemented in bot_degree. Requirements 4-7 and 9 have been tested and evaluated, see section 7 for an evaluation of bot_degree's performance against other Arimaa playing bots. Requirement 8 was not implemented because it was decided that trying to construct a way to choose the best starting position based on the other player was infeasible given the amount of time allocated to this project, see section 8.1.1 for further explanation.

8.2 Improvement

A substantial improvement to bot_degree would be achieved by optimising the random move generating code. This is because the random move function is used the most during bot_degree's execution. For instance, if bot_degree were simulating 10,000 random games per move, with each random game on average having 40 moves per player, this would amount to 800,000 function calls to the random move function, (i.e. $10,000 \times 2 \times 40 = 800,000$). If this function call took 0.0003 seconds to complete, this would still amount to over 4 minutes per move and thus over two hours to complete a whole game. Obviously there are machines that will be able to run bot_degree's code faster, however to increase the efficiency of the UCT algorithm and Monte-Carlo Simulation, many more simulations need to be completed in order to guarantee that the UCT algorithm converges to the best move. Consequently if this factor is not taken into account, bot_degree will not scale at all.

Another area of bot_degree's code that could be optimised to run faster would be the way the UCT algorithm descends the game tree. As described in [6], MoGo has already benefited from using multi threads therefore extending bot_degree to use multi threading is definitely plausible. The modifications that would have to be made would be to use mutual exclusion techniques to lock the tree for each thread. Each thread would then be able to descend a different route down the tree concurrently and therefore bot_degree would be able to use more then one processor to its advantage by simulating multiple random games simultaneously. This technique would greatly reduce the time taken per move and thus would allow bot_degree to increase the number of random simulations it could achieve, enhancing the effectiveness of the UCT algorithm.

Currently, during Monte-Carlo Simulation, a random game is played to the end and evaluated as either being a 1 or 0. However as described previously, this can take an extended period of time if the number of simulations per move is large. Therefore an evaluation function could be implemented which evaluates a board position and decides which side would win, if the game were to carry on to the end. The number of random moves that are simulated before the evaluation function is used would have to be tested, however using this method would decrease the number of function calls to the random move function per Monte-Carlo Simulation and thus would speed up the execution time of bot_degree. This technique could be further improved by using a weighted bias so that instead of using either 1 for winning and 0 for losing, the payoff of a particular board state could be in the interval of [0,1] based on some heuristic evaluation. As long as the heuristic evaluation was good at distinguishing between excellent winning moves and adequate winning moves, this would bias the exploitation of the UCT algorithm towards better winning moves.

When bot_degree adds a node to the game tree, it chooses a move at random, creates a Node object and adds it to the tree. As explained in section 7, this is a possible reason why bot_degree performs very poorly against other agents that actually evaluate a position because the moves that it chooses are not evaluated in any way. Therefore a possible improvement would be to calculate random moves and then evaluate a number of them based on some evaluation function. The random move with the best evaluation would then be added to the tree, and thus only "better" random moves would populate the tree. Consequently this would aid the UCT function in choosing better nodes to exploit and explore and would improve the performance of bot_degree.

One way to evaluate board positions would be to look for patterns. MoGo already uses patterns to its advantage and is definitely a plausible improvement to bot_degree. As explained in [6], using patterns greatly improves the effectiveness of Monte-Carlo Simulation and the UCT algorithm because it drives the simulation towards better moves and thus builds the game tree with more meaningful moves as opposed to just random ones. For instance, after experimentation it has become apparent that bot_degree will move its own pieces into the trap squares, or move a piece between two squares repeatedly over many moves making no progress. With pattern matching this could be reduced.

As suggested in [11], another way to optimise the UCT algorithms trade-off between exploration and exploitation is to use a weighted bias within equation 3.1, section 3.1.1. This has been implemented in MoGo and has been verified to improve the UCT algorithm, therefore there is potential to implement this technique in bot_degree as well. It suggests that equation 8.2 should be used, where t is the current episode, s is the number of times arm i has been played up to time t, D is the estimated game length starting from the current node and d is the distance from the root. Therefore if the UCT algorithm is currently visiting nodes far away from the root, the bias will be small and will thus bias the UCT algorithm less.

$$bias = \ln(\frac{t}{s})^{(\frac{D+d}{2D+d})} \tag{8.2}$$

As explained in section 8.3, bot_degree produces an error around 21% of the time and thus prematurely terminates the game. Bot_degree uses a transposition table to keep track of moves that it has produced and also uses it when calculating possible legal moves from a given board position. Since bot_degree uses the underlying code from the random bot r0, which was not implemented with bot_degree in mind, it could be that bot_degree is using this code in slightly the wrong way and could be a reason for the error. For instance, it may be the case that the state of the transposition table needs to be saved before a simulation is carried out, and then restored after the simulation has finished, therefore achieving the effect that the simulation never took place as far as the transposition table is concerned.

Another way to possibly reduce the error would be to save the state of the transposition table within a node. Doing this would mean that when a new random move was to be created from a particular node, the transposition table could be initialised to the transposition table within the node and thus the random move function would hopefully not get confused and produce bad moves. However the drawback of saving the transposition table within each node would be that the total memory required by bot_degree would be greatly increased.

8.3 Known Issues

One of the known issues with bot_degree is that it will sometimes produce a move that has nothing to do with the previous board position. Once this move has been played the offline Perl script called “match”, that plays two agents against each other, will detect this bad move and produce the following error, “bad step” followed by the move that produced the error. The Perl script will then terminate, prematurely ending the game between the two agents. When this problem was first discovered, bot_degree produced a bad step approximately 50% of the time. This was obviously unacceptable and meant that it was impossible to evaluate bot_degree effectively without this problem biasing the outcome.

Tests were carried out to determine the nature of the error. These involved checking the board states before, during and after the random move function was executed. It was discovered that if the simulation part of bot_degree was not executed, then the error was reduced to occurring around 5% of the time. Obviously the simulation part of bot_degree is needed to evaluate a board position, therefore a second random move function was created, which closely resembled the first apart from a few lines of code. This new random move function was used during the simulation part of bot_degree and helped reduce the bad moves to happening around 21% of the time. Since this is a much smaller percentage, it was decided that given timing constraints, it would not be feasible to try and reduce the percentage error further. See section 8.2 for possible ways to fix the error.

During the tests it became evident that the more simulations bot_degree carried out, the more likely bot_degree was to produce a bad move see table 8.1. The reason for this trend in the increase of bad moves is because if the number of simulations is increasing, bot_degree will call the random move function more frequently. Therefore if the random move function is called more frequently, there is more of a chance that bot_degree will generate a bad move.

Table 8.1: The percentage of bad moves with varying simulations

Number Of Simulations Per Move	Number Of Games Played	Percentage Of Bad Moves
10	768	12.4%
50	799	23.5%
100	783	22.9%
500	800	26.6%
1,000	800	28.8%
5,000	726	26.7%
10,000	799	25.3%

Appendix A

Arimaa Server Rules

This chapter describes the various constraints that are imposed on the Arimaa server and further rules that apply to Arimaa.

A.1 Notation

Every game that is played on the Arimaa server is recorded for future reference, therefore there are a number of different notations that must be taken into account [14]. This notation is also important because bot_degree must receive a position file, as input, containing the current board state, and produce a string, as output, containing the move that it would like to make. This notation uses upper and lower case letters for Gold and Silver respectively. Indication of each piece is as follows: Elephant(e,E) Camel(m,M) Horse(h,H) Dog(d,D) Cat(c,C) Rabbit(r,R). See figure A.1 for an example board state. In the example it shows the location of the traps, marked by an x, but this is not a necessity. The first line of the position file, indicates whose turn it is, in this example it is Silver's 8th turn.

Figure A.1: An example input board state

```
8s
 +-----------------+
8|   m   r r   r   |
7| r         e     |
6|   r h r   x r   |
5| h   d     c d   |
4|     H         M |
3|     x   R   R   |
2| D   C R   C   D |
1|   R   R R   R   |
 +-----------------+
   a b c d e f g h
```

At the start of the game each player must specify where they want their pieces to be placed, this is always recorded as being the first move of each player. This is done by indicating the piece using a symbol, as above, and then the square that it is to be placed on. For instance ea2 means the Silver Elephant starts on square a2. Moves are recorded in groups of fours. Each group contains the piece to be moved, the square it is currently occupying and the direction it is moving in. The direction can either be n, s, e, w for North, South, East and West respectively. Therefore a possible output giving a move could be `cd3n cd4e ce4s ce3e` which indicates that the Silver Cat moves from square d3 to square f3 in four moves.

A.2 Special Situations

There are a few special rules that exist in Arimaa. A player loses the game if they lose all their Rabbits, or unable to move any of their pieces. An enemy rabbit may be pushed or pulled into the goal squares, but if after the move is complete it remains there, the enemy Rabbit's player wins. If both players lose their Rabbits in the same turn, then the player making the move wins the game. Repetition is not allowed in Arimaa, therefore if the same board position and move is made three times, that move is made illegal and a player must make a different move. If there are no other moves that the player can make then they lose the game because they cannot make a move.

A.3 Timing Constraints

Time constraints are used in official match games of Arimaa. This means that in order to use the server to test bot_degree, it will need to conform to these extra rules. They can be found on the Arimaa website [14] and the following is a summary of the main points. If a game is stopped because of any of the time constraints, the player with the most pieces wins, however if no pieces have been removed from the game then the player who moved second wins. At the start of the game, time constraints must be specified. They are in the form of M/R/P/L/G/T, see Table A.1 for details.

Table A.1: The description of M/R/P/L/G/T, the time control format

	Description	Format	example
M	The amount of time per move	minutes:seconds	0:30
R	The amount of time a player has in reserve	minutes:seconds	2:00
T	The amount of time that a player must make a move in	minutes:seconds	5:21
P	The percentage of unused M that gets added to R	number <= 100, number >= 0	100
L	The limit the reserve can reach	minutes:seconds	5:32
G	The amount of time that's allowed for the game, may also be specified in the number of turns, t, each player has before the game ends	hours:minutes or number t	24:59, 70t

A.4 Interfacing With The Arimaa Server

In order for bot_degree to be able to play Arimaa games against other agents, it will need to interface with the Arimaa server as described in [14]. Bot_degree should accept a file that contains the current board state and the move number, it should then compute a legal move from this board state and output the move to standard output, `printf`, using the notation described in Section A.1.

Appendix B

The Effectiveness Of The UCT Algorithm

An experiment was carried out involving 5 “arms of a bandit”, where each arms payoff consisted of a fixed amount plus some random noise, see table B.1.

Table B.1: The payoff of each arm

Arm	Fixed payoff	Random Noise, x
1	0.87	$-0.05 \leq x \leq 0.05$
2	0.62	$-0.20 \leq x \leq 0.20$
3	0.75	$-0.20 \leq x \leq 0.20$
4	0.39	$-0.10 \leq x \leq 0.10$
5	0.69	$-0.17 \leq x \leq 0.17$

The experiment simulated 10,000 times of picking an arm using both algorithms, and was run 10,000 times in order to thoroughly test both algorithms and minimise error. The ϵ-Greedy algorithm that was used had ϵ probability of choosing a random arm, and $1 - \epsilon$ probability of choosing the arm that had the highest payoff so far. Meaning if the ϵ-Greedy algorithm gets lucky early on and chooses arm 1, it may end up performing very well. However if it is unlucky and receives a one off high payoff from say arm 3 it will always think that arm 3 is the best arm to choose and will under perform greatly later on in the experiment. Equation 3.1, section 3.1.1, is the UCT algorithm which was tested against the ϵ-Greedy algorithm with ϵ set to 0.5. The mean payoff and the mean payoff^2 were calculated per iteration, along with the standard deviation, see table B.2.

Table B.2: The results of the experiment between the UCT and ϵ-Greedy algorithms

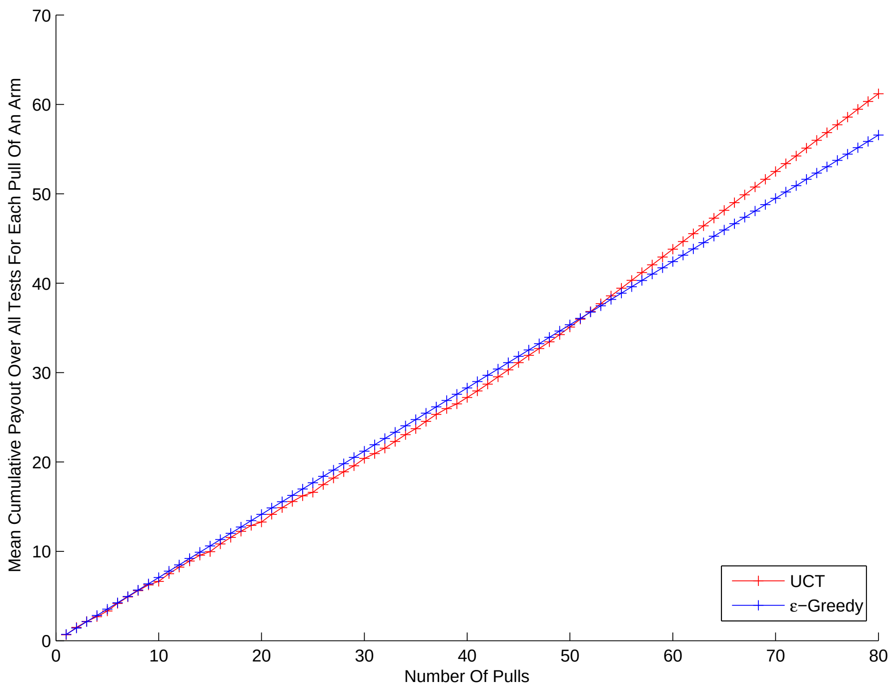
Algorithm	Mean payoff	Mean payoff^2	Standard Deviation
ϵ-Greedy	0.71	0.53	0.16
UCT	0.87	0.76	0.03

The results show that UCT has a 21% higher mean payoff then ϵ-Greedy, and a standard deviation four times lower proving that UCT chooses the best arm, in this case arm one, more often then

ϵ-Greedy, and thus is less prone to choosing an arm that produces a lower payoff. A graph was plotted to show that unlike ϵ-Greedy, UCT goes through initial stages of learning, see figure B.1. The cumulative mean payoff that each arm receives on each pull of the test was calculated, (i.e. The cumulative mean payoff over all tests for the first pull of each test was calculated. The cumulative mean payoff over all tests for the second pull of each test was calculated etcetera). The cumulative mean payoff was plotted against the number of pulls within each test, however the graph shows the first 80 pulls of the test because after this point the UCT has finished learning.

It can be seen initially that the UCT algorithm performs randomly for the first 5 pulls of the test. This is because it must try each arm once before it can compute the arm which maximises the UCT algorithm effectively. The learning stage of the UCT algorithm can be seen between pulls 1 and 50; it behaves worse then the ϵ-Greedy algorithm because it has not tried enough arms yet in order for the it to choose the best arm. After 50 pulls the UCT has sufficiently explored all the arms and has enough data in order to choose the arm with the highest payoff every time. This is clearly seen because the UCT data points now reside above the ϵ-Greedy data points and continue to do so for the remainder of the experiment. The ϵ-Greedy algorithm has no learning as it performs quite randomly, this is the reason why the ϵ-Greedy data is assembled in a straight line.

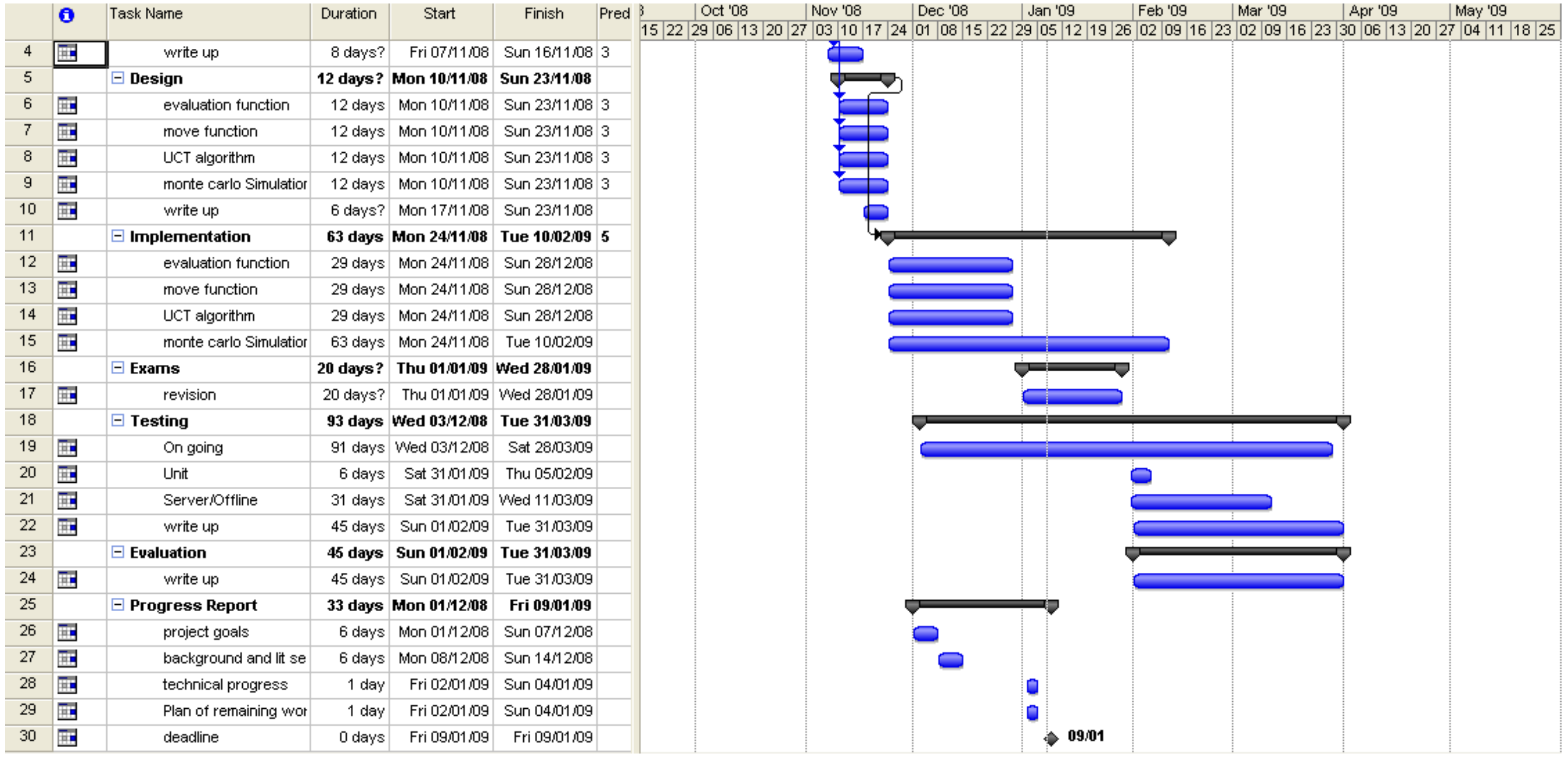
Figure B.1: The cumulative mean payoff of each pull of the experiment



Appendix C

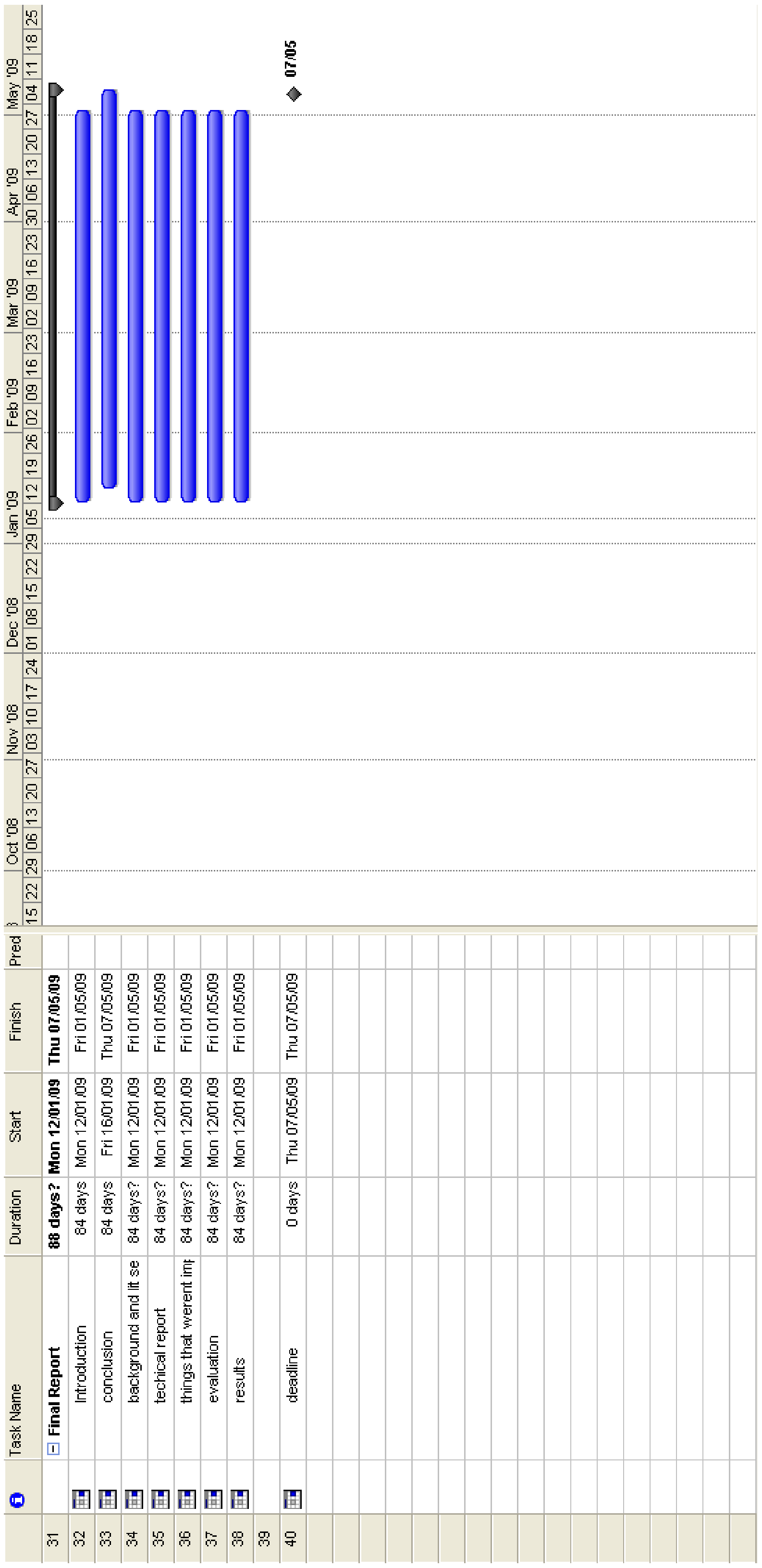
Initial Gantt Chart Of Project Work

Figure C.1: Page 1 of the initial project Gantt Chart



		Task Name	Duration	Start	Finish	Pred
4		write up	8 days?	Fri 07/11/08	Sun 16/11/08	3
5		**Design**	**12 days?**	**Mon 10/11/08**	**Sun 23/11/08**	
6		evaluation function	12 days	Mon 10/11/08	Sun 23/11/08	3
7		move function	12 days	Mon 10/11/08	Sun 23/11/08	3
8		UCT algorithm	12 days	Mon 10/11/08	Sun 23/11/08	3
9		monte carlo Simulatior	12 days	Mon 10/11/08	Sun 23/11/08	3
10		write up	6 days?	Mon 17/11/08	Sun 23/11/08	
11		**Implementation**	**63 days**	**Mon 24/11/08**	**Tue 10/02/09**	**5**
12		evaluation function	29 days	Mon 24/11/08	Sun 28/12/08	
13		move function	29 days	Mon 24/11/08	Sun 28/12/08	
14		UCT algorithm	29 days	Mon 24/11/08	Sun 28/12/08	
15		monte carlo Simulatior	63 days	Mon 24/11/08	Tue 10/02/09	
16		**Exams**	**20 days?**	**Thu 01/01/09**	**Wed 28/01/09**	
17		revision	20 days?	Thu 01/01/09	Wed 28/01/09	
18		**Testing**	**93 days**	**Wed 03/12/08**	**Tue 31/03/09**	
19		On going	91 days	Wed 03/12/08	Sat 28/03/09	
20		Unit	6 days	Sat 31/01/09	Thu 05/02/09	
21		Server/Offline	31 days	Sat 31/01/09	Wed 11/03/09	
22		write up	45 days	Sun 01/02/09	Tue 31/03/09	
23		**Evaluation**	**45 days**	**Sun 01/02/09**	**Tue 31/03/09**	
24		write up	45 days	Sun 01/02/09	Tue 31/03/09	
25		**Progress Report**	**33 days**	**Mon 01/12/08**	**Fri 09/01/09**	
26		project goals	6 days	Mon 01/12/08	Sun 07/12/08	
27		background and lit se	6 days	Mon 08/12/08	Sun 14/12/08	
28		technical progress	1 day	Fri 02/01/09	Sun 04/01/09	
29		Plan of remaining wor	1 day	Fri 02/01/09	Sun 04/01/09	
30		deadline	0 days	Fri 09/01/09	Fri 09/01/09	

Figure C.2: Page 2 of the initial project Gantt Chart

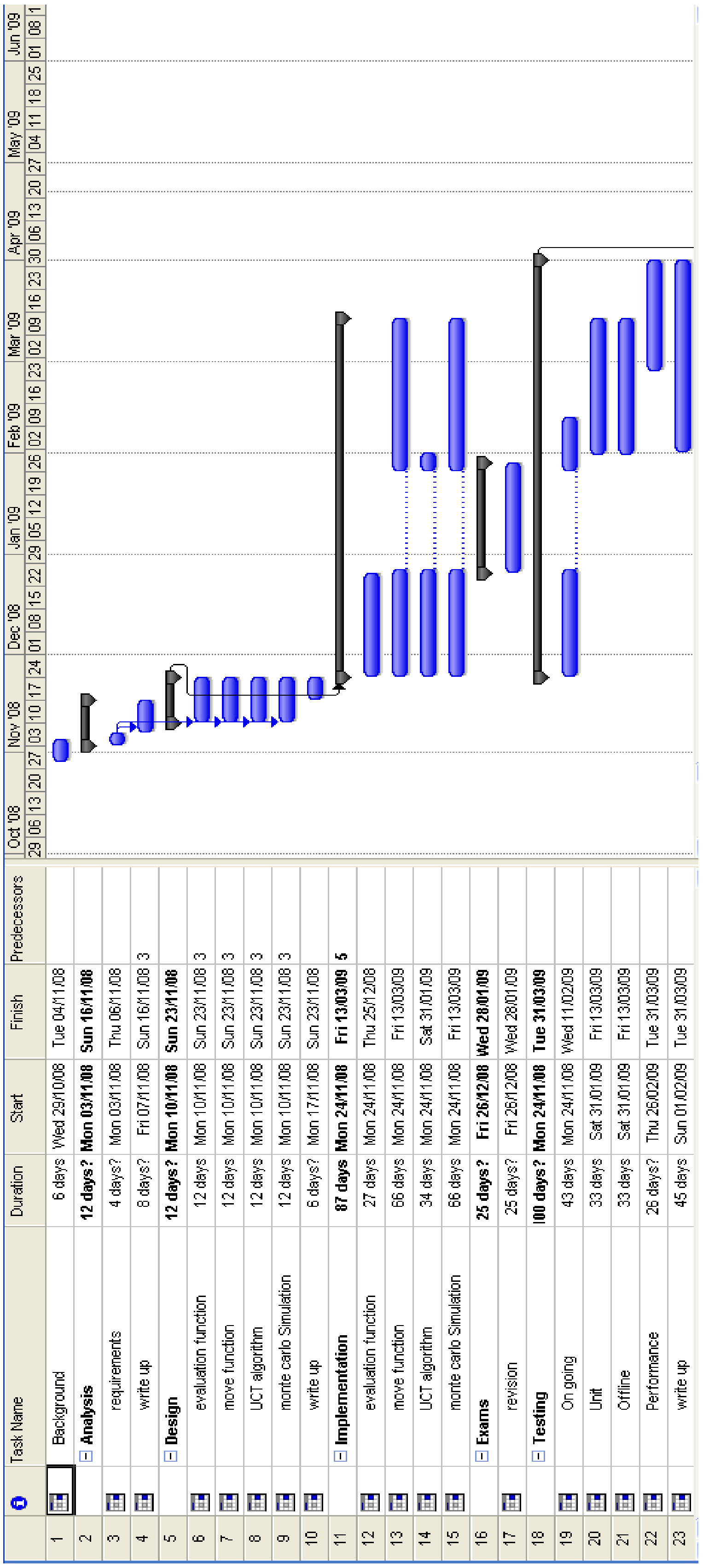


	Task Name	Duration	Start	Finish	Pred
31	**Final Report**	**88 days?**	**Mon 12/01/09**	**Thu 07/05/09**	
32	Introduction	84 days	Mon 12/01/09	Fri 01/05/09	
33	conclusion	84 days	Fri 16/01/09	Thu 07/05/09	
34	background and lit se	84 days?	Mon 12/01/09	Fri 01/05/09	
35	techical report	84 days?	Mon 12/01/09	Fri 01/05/09	
36	things that werent imp	84 days?	Mon 12/01/09	Fri 01/05/09	
37	evaluation	84 days?	Mon 12/01/09	Fri 01/05/09	
38	results	84 days?	Mon 12/01/09	Fri 01/05/09	
39					
40	deadline	0 days	Thu 07/05/09	Thu 07/05/09	

Appendix D

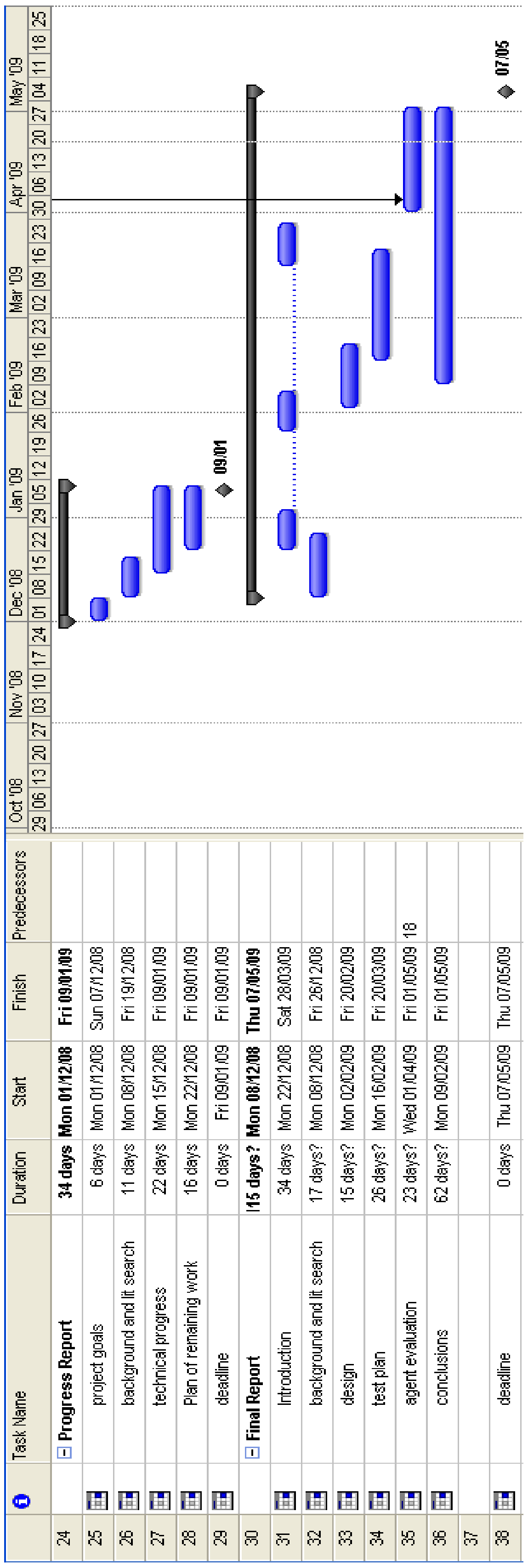
Final Gantt Chart Of Project Work

Figure D.1: Page 1 of the final project Gantt Chart



	Task Name	Duration	Start	Finish	Predecessors
1	Background	6 days	Wed 29/10/08	Tue 04/11/08	
2	**Analysis**	**12 days?**	**Mon 03/11/08**	**Sun 16/11/08**	
3	requirements	4 days?	Mon 03/11/08	Thu 06/11/08	
4	write up	8 days?	Fri 07/11/08	Sun 16/11/08	3
5	**Design**	**12 days?**	**Mon 10/11/08**	**Sun 23/11/08**	
6	evaluation function	12 days	Mon 10/11/08	Sun 23/11/08	3
7	move function	12 days	Mon 10/11/08	Sun 23/11/08	3
8	UCT algorithm	12 days	Mon 10/11/08	Sun 23/11/08	3
9	monte carlo Simulation	12 days	Mon 10/11/08	Sun 23/11/08	3
10	write up	6 days?	Mon 17/11/08	Sun 23/11/08	
11	**Implementation**	**87 days**	**Mon 24/11/08**	**Fri 13/03/09**	**5**
12	evaluation function	27 days	Mon 24/11/08	Thu 25/12/08	
13	move function	66 days	Mon 24/11/08	Fri 13/03/09	
14	UCT algorithm	34 days	Mon 24/11/08	Sat 31/01/09	
15	monte carlo Simulation	66 days	Mon 24/11/08	Fri 13/03/09	
16	**Exams**	**25 days?**	**Fri 26/12/08**	**Wed 28/01/09**	
17	revision	25 days?	Fri 26/12/08	Wed 28/01/09	
18	**Testing**	**100 days?**	**Mon 24/11/08**	**Tue 31/03/09**	
19	On going	43 days	Mon 24/11/08	Wed 11/02/09	
20	Unit	33 days	Sat 31/01/09	Fri 13/03/09	
21	Offline	33 days	Sat 31/01/09	Fri 13/03/09	
22	Performance	26 days?	Thu 26/02/09	Tue 31/03/09	
23	write up	45 days	Sun 01/02/09	Tue 31/03/09	

Figure D.2: Page 2 of the final project Gantt Chart



	Task Name	Duration	Start	Finish	Predecessors
24	**Progress Report**	**34 days**	**Mon 01/12/08**	**Fri 09/01/09**	
25	project goals	6 days	Mon 01/12/08	Sun 07/12/08	
26	background and lit search	11 days	Mon 08/12/08	Fri 19/12/08	
27	technical progress	22 days	Mon 15/12/08	Fri 09/01/09	
28	Plan of remaining work	16 days	Mon 22/12/08	Fri 09/01/09	
29	deadline	0 days	Fri 09/01/09	Fri 09/01/09	
30	**Final Report**	**115 days?**	**Mon 08/12/08**	**Thu 07/05/09**	
31	Introduction	34 days	Mon 22/12/08	Sat 28/03/09	
32	background and lit search	17 days?	Mon 08/12/08	Fri 26/12/08	
33	design	15 days?	Mon 02/02/09	Fri 20/02/09	
34	test plan	26 days?	Mon 16/02/09	Fri 20/03/09	
35	agent evaluation	23 days?	Wed 01/04/09	Fri 01/05/09	18
36	conclusions	62 days?	Mon 09/02/09	Fri 01/05/09	
37					
38	deadline	0 days	Thu 07/05/09	Thu 07/05/09	

Appendix E

Test Cases

Table E.1: Design : The test cases that were used to test bot_degree

Test Number	Test Description	Expected Outcome	Requirements Satisfied
1	Test the functionality of the tree class and how it can be used. Create a new tree object of type int, insert 81 and 27 into the tree as the children of the root using the tree iterator.	Both children are added without error and can be accessed by using the tree iterator.	4.1 (3), 4.2 (5)
2	Setup the test as in 1, but instead insert 4 and 9 into the tree as the children of the root using the tree iterator, then insert 97 and 128 as the children of 4.	All four children are added without error and can be accessed by using the tree iterator.	4.1 (3), 4.2 (5)
3	Setup the test as in 1, but instead do not insert any children, then try to access the children of the root by using the tree iterator.	An error is produced and/or the program terminates because there are no children of the root.	4.1 (3), 4.2 (5)
4	Setup the test as in 1, but instead insert 2, 7, 9 and 129 into the tree as the children of the root. Then using the tree iterator access each child of the root, but upon reaching the last child, i.e. 129, increment the iterator one more time.	An error is produced and/or the program terminates because the iterator has gone past the number of children at that level in the tree.	4.1 (3), 4.2 (5)
5	Test the functionality of the tree class and how it can be used with the Node class. Create a new tree object of type Node, create a new Node, a, with its variable count equal to 5 and insert into the tree as children of the root. Try to access Node a's count using the tree iterator.	Node a's count is accessed and equals 5.	4.1 (3), 4.2 (5)
6	Test the functionality of the tree class and how it can be used with the Node class, as well as passing by reference. Create a new tree object of type Node, create new Nodes, a, b and c and insert them into the tree as children of the root. Create a new array of type node pointers and copy the address of a, b and c, using the tree iterator, to the array. Update each node's count in the array to equal 3 and totalpayoff equal 97.	When the nodes a, b and c are accessed from the tree using the tree iterator, the count and totalpayoff will be updated with the new values.	4.1 (3), 4.2 (5)
7	Test the functionality of the Node class. Create a new node, b. Access b's variables, count, boardState, main_line and totalPayoff.	All variables can be successfully accessed.	4.1 (3), 4.2 (5)

Table E.2: Design : The test cases that were used to test bot_degree

Test Number	Test Description	Expected Outcome	Requirements Satisfied
8	Test the functionality of the Node class using pointers. Create a new node, b and a new node pointer, c. Initialise b's count to 5 and totalpayoff to 245. Initialise c to the address of b. Change c's count to 27.	b's count will equal 27, but the total-payoff will remain the same.	4.1 (3), 4.2 (5)
9	Test the functionality of the Node classes overloaded equality operator. Create nodes a and b. Initialise a's count to 498 and b's count to 12. Check whether equality operator is satisfied, (i.e. a == b returns true).	Equality operator will return false as the two nodes are not equal.	4.1 (3), 4.2 (5)
10	Test the functionality of the Node classes overloaded equality operator. Create nodes a and b. Initialise a's count to 498. Then initialise b to a (i.e. b = a) Check whether equality operator is satisfied, (i.e. a == b returns true).	Equality operator will return true as the two nodes are equal.	4.1 (3), 4.2 (5)
11	Test the functionality of the getMove classes updateNodes function. Create nodes a, b, c, d and e. Create a node pointer array, nodes, of size 5. Create a node pointer pointer, arrayP and initialise to the first element of nodes (i.e. arrayP = nodes). Add the address of a, b, c, d and e to nodes using arrayP. Call function updateNode(arrayP = nodes, 5, 20) where 5 is the number of elements in nodes and 20 is the payoff.	The count of each node in nodes is incremented by 1 and the payoff of each node in nodes is incremented by 20.	4.1 (3), 4.2 (5), 4.3.1 (3)
12	Setup the test as in 11, but instead call function updateNode(arrayP = nodes, 7, 20).	An error is produced and/or the program terminates because the iterator arrayP will try and access an address in memory that is not associated with nodes.	4.1 (3), 4.2 (5), 4.3.1 (3)
13	Setup the test as in 11, but instead nodes is of size 10. Call function updateNode(arrayP = nodes, 7, 20).	An error is produced and/or the program terminates because the 6th and 7th elements in nodes do not point to a node.	4.1 (3), 4.2 (5), 4.3.1 (3)

Table E.3: Design : The test cases that were used to test bot_degree

Test Number	Test Description	Expected Outcome	Requirements Satisfied
14	Test the functionality of the getMove classes chooseUCT function. Create a new tree object of type Node, create nodes a, b and c, with their count equal to 1 and a's total payoff equal to 150. Insert a, b, c into tree as children of the root. Create tree iterator, it, and initialise to point to the first child of the root, a. Initialise episode, as used by chooseUCT, to 3. Call function chooseUCT(it,3) where 3 is the number of children of the root.	The chooseUCT function will return 1 indicating that the first child of the root, i.e. a, has highest UCT value.	4.1 (3), 4.2 (5), 4.3.1 (3)
15	Setup the test as in 14, but instead initialise c's total payoff to 151.	The chooseUCT function will return 3 indicating that the third child of the root, i.e. c, has highest UCT value.	4.1 (3), 4.2 (5), 4.3.1 (3)
16	Setup the test as in 14, but instead call function chooseUCT(it,0).	The chooseUCT function will return -1 and produce an error message indicating that no children where passed into the function and thus cannot choose a child. Therefore must make sure the correct number of children are passed into the function.	4.1 (3), 4.2 (5), 4.3.1 (3)
17	Setup the test as in 14, but instead initialise tree iterator, it, to point to the children of a, i.e. no children. Call function chooseUCT(it,3).	An error is produced and/or the program terminates because there are no children of a, and thus the iterator will attempt to access a memory location that is not associated with the tree.	4.1 (3), 4.2 (5), 4.3.1 (3)
18	Test the uct value within chooseUCT. Setup the test as in 14.	UCT value for a will equal 151.4823, UCT value for b and c will equal 1.4823.	4.1 (3), 4.2 (5), 4.3.1 (3)
19	Test the uct value within chooseUCT. Setup the test as in 14, but have count for a equal to 10, count for b equal to 2 and payoff for b equal to 75.	UCT value for a will equal 15.4687, UCT value for b will equal 38.5481 and UCT value for c will equal 1.4823. Thus the chooseUCT function will return 2 indicating that the second child of the root, i.e. b, has highest UCT value.	4.1 (3), 4.2 (5), 4.3.1 (3)

Table E.4: Design : The test cases that were used to test bot_degree

Test Number	Test Description	Expected Outcome	Requirements Satisfied
20	Test the functionality of the getMove classes simulate function. Create a new position, p, that is empty. Call function simulate(p).	An error is produced and/or the program terminates because no pieces exist on the board and therefore, cannot choose a random move. May also hang and/or get stuck in an infinite loop.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)
21	Test the functionality of the getMove classes simulate function. Create a new position, p, initialise to Gold having already won (i.e. Golds rabbit has reached the goal). Checking that no simulation is carried out because Gold has already won. Call function simulate(p).	No simulation is carried out because Gold has already won, correctly identifies that Gold has won.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)
22	Setup the test as in 21, but initialise p to Silver having already won (i.e. Silvers rabbit has reached the goal).	No simulation is carried out because Silver has already won, correctly identifies that Gold has won.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)
23	Setup the test as in 21, but initialise p so that Gold has no rabbits left (i.e. Gold has lost).	No simulation is carried out because Gold has already lost, correctly identifies that Silver has won.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)
24	Setup the test as in 21, but initialise p so that Silver has no rabbits left (i.e. Silver has lost).	No simulation is carried out because Silver has already lost, correctly identifies that Gold has won.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)
25	Setup the test as in 21, but initialise p so that Gold has no unfrozen pieces (i.e. Gold has lost).	No simulation is carried out because Gold has already lost, correctly identifies that Silver has won.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)
26	Setup the test as in 21, but initialise p so that Silver has no unfrozen pieces (i.e. Silver has lost).	No simulation is carried out because Silver has already lost, correctly identifies that Gold has won.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)

Table E.5: Design : The test cases that were used to test bot_degree

Test Number	Test Description	Expected Outcome	Requirements Satisfied
27	Setup the test as in table E.4 test 21, but initialise p to a legal board state. Test 10 times, each time initialising p to a different legal board state.	A simulation from the current board state p until either Gold or Silver wins is carried out.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)
28	Test the functionality of the getMove classes descendUCT function. Create a new tree object of type Node, create nodes a, b and c, with their count equal to 3 and a's total payoff equal to 150. Insert a, b, c into tree as children of the root. Create tree iterator, it, and initialise to point to the root of the tree. Initialise episode, as used by chooseUCT, to 3. Call function descendUCT(tree,3), where 3 is the number of children of the root.	The descendUCT function will return the tree iterator, it, which will point to node a within the tree indicating that a maximises the UCT algorithm	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)
29	Setup the test as in 28, but instead call function descendUCT(it,7), where 7 is the number of children of the root	An error is produced and/or the program terminates because the chooseUCT function located within descendUCT will attempt to increment the tree iterator, it, past the last child node, c, and thus will point to an area of memory that is not associated with the tree.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)
30	Setup the test as in 28, but instead call function descendUCT(it,tree.size), where tree.size returns the number of children of the root, (i.e. 3)	The descendUCT function will return the tree iterator, it, which will point to node a within the tree indicating that a maximises the UCT algorithm	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)
31	Setup the test as in 28, but instead initialise node c's total payoff to 390. Call function descendUCT(it,tree.size), where tree.size returns the number of children of the root, (i.e. 3)	The descendUCT function will return the tree iterator, it, which will point to node c within the tree indicating that c maximises the UCT algorithm	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)

Table E.6: Design : The test cases that were used to test bot_degree

Test Number	Test Description	Expected Outcome	Requirements Satisfied
32	Test the functionality of the getMove classes randomMove function. Create a new position, p, initialise to a random starting board state. Indicate that it is Golds turn to move using one of the variables inside position. Create a new position, n, and create a new main_line, m. Call function randomMove(p,n,m). Test 10 times, each time initialising p to a different random starting position	A random Gold move is calculated from the board state p, placed in m and executed on n. The board state n, will be a result of applying the move in m to p.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)
33	Setup the test as in 32, but instead indicate that it is Silvers turn to move using one of the variables inside position. Call function randomMove(p,n,m). Test 10 times, each time initialising p to a different random starting position	A random Silver move is calculated from the board state p, placed in m and executed on n. The board state n, will be a result of applying the move in m to p	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)
34	Setup the test as in 32, but instead initialise p to a random board state. Indicate that it is Golds turn to move using one of the variables inside position. Call function randomMove(p,n,m). Test 10 times, each time initialising p to a different random position	A random Gold move is calculated from the board state p, placed in m and executed on n. The board state n, will be a result of applying the move in m to p	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)
35	Setup the test as in 32, but instead initialise p to a random board state. Indicate that it is Golds turn to move using one of the variables inside position. Call function randomMove(p,n,m). Test 10 times, each time initialising p to a different random position	A random Gold move is calculated from the board state p, placed in m and executed on n. The board state n, will be a result of applying the move in m to p	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (3)
36	Test the functionality of the getMove classes decay function. Set number of simulations equal to 1,000 and the current episode to 1. Call function decay().	The decay function should return 0.2399.	4.2 (5)
37	Setup the test as in 36 but instead set number of simulations equal to 10,000 and the current episode to 5,999. Call function decay().	The decay function should return 0.5646.	4.2 (5)

Table E.7: Design : The test cases that were used to test bot_degree

Test Number	Test Description	Expected Outcome	Requirements Satisfied
38	Test the getMove class as a whole, using black box testing. Set number of simulations to 10. Use the Perl script "match" to play bot_degree against random bot "r0", Test 10 times.	The two bots, r0 and bot_degree, play 10 games against each other.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (1), 4.3.1 (3)
39	Setup the test as in 38 but instead set number of simulations to 100. Use the Perl script "match" to play bot_degree against random bot "r0", Test 10 times.	The two bots, r0 and bot_degree, play 10 games against each other.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (1), 4.3.1 (3)
40	Setup the test as in 38 but instead set number of simulations to 1,000. Use the Perl script "match" to play bot_degree against random bot "r0", Test 10 times.	The two bots, r0 and bot_degree, play 10 games against each other.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (1), 4.3.1 (3)
41	Performance test the getMove class. Set number of simulations to 1,000. Use the Perl script "match" to play bot_degree against random bot "r0", Test 100 times.	The two bots, r0 and bot_degree, play 100 games against each other.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (1), 4.3.1 (3)
42	Performance test the getMove class. Set number of simulations to 10,000. Use the Perl script "match" to play bot_degree against random bot "r0", Test 100 times.	The two bots, r0 and bot_degree, play 100 games against each other.	4.1 (1), 4.1 (3), 4.2 (5), 4.3.1 (1), 4.3.1 (3)

Table E.8: Results : The results of the tests that were used to test bot_degree

Test Number	Actual Outcome	pass/fail	comment
1	Both children were added without error and were accessed by using the tree iterator.	pass	no further comments.
2	All four children were added without error and were accessed by using the tree iterator.	pass	no further comments.
3	An error was produced and the program terminated	pass	no further comments.
4	An error was produced and the program terminated	pass	no further comments.
5	Node a's count was accessed and it did equal 5	pass	no further comments.
6	When nodes a, b and c are accessed, their count is equal to 3 and their totalpayoff is equal to 97	pass	need to make sure passing by reference and passing by value are not mixed up, otherwise will give undesired results.
7	Node b's variables are all accessed successfully and give default values, i.e. 0	pass	no further comments.

Table E.9: Results : The results of the tests that were used to test bot_degree

Test Number	Actual Outcome	pass/fail	comment
8	b's count was equal to 27 and the total payoff stayed the same	pass	no further comments.
9	Equality operator returned false	pass	no further comments.
10	Equality operator returned false	fail	This failed because the objects where not identical, (i.e. even though they had the same parameters, they are not the same), therefore an alternative test was carried out: create a Node pointer, c. Initialise c to the memory address of a. Check whether equality operator is satisfied, (i.e. a == *c returns true). This returned true, and is sufficient for bot_degree as the equality operator is only used when inserting nodes into the tree and only needs to determine if the node being inserted is already in the tree.
11	The count of each node was incremented by 1 and the payoff of each node within the nodes array was incremented by 20.	pass	no further comments.
12	An error was produced and the program terminated.	pass	no further comments.
13	An error was produced and the program terminated.	pass	no further comments.

Table E.10: Results : The results of the tests that were used to test bot_degree

Test Number	Actual Outcome	pass/fail	comment
14	The chooseUCT function did return 1 indicating that the first child of the root, a, has highest UCT value.	pass	no further comments.
15	The chooseUCT function did return 3 indicating that the third child of the root, c, has highest UCT value.	pass	no further comments.
16	The chooseUCT function did return -1, and produced the error message to standard error output	pass	no further comments.
17	An error was produced and the program terminated.	pass	no further comments.
18	The UCT value for a was equal to 151.4823 and UCT value for b and c was equal to 1.4823.	pass	no further comments.
19	The UCT value for a was equal to 15.4687, the UCT value for b was equal 38.5481 and the UCT value for c was equal 1.4823. The chooseUCT function returned 2 indicating that the second child of the root, i.e. b, had the highest UCT value.	pass	no further comments.

Table E.11: Results : The results of the tests that were used to test bot_degree

Test Number	Actual Outcome	pass/fail	comment
20	The function simulate returns 1, indicating that Gold won from this position. An error was produced when p was printed to screen, indicating that p was not initialised	fail	The function simulate returned 1 because Silver had no rabbits left, (i.e. an empty board state). However an empty board state is never produced by randomMove because it would produce an error.
21	No simulations were carried out because Gold had already won.	pass	no further comments.
22	No simulations were carried out because Silver had already won.	pass	no further comments.
23	No simulations were carried out because Gold had already lost	pass	no further comments.
24	No simulations were carried out because Silver had already lost	pass	no further comments.
25	No simulations were carried out because Gold had already lost	pass	no further comments.
26	No simulations were carried out because Silver had already lost	pass	no further comments.

Table E.12: Results : The results of the tests that were used to test bot_degree

Test Number	Actual Outcome	pass/fail	comment
27	10 simulations were carried out from p, until either Gold or Silver won, without error.	pass	no further comments.
28	The descendUCT function returned the tree iterator, it, pointing to node a within the tree. This was verified by checking the total payoff of the tree iterator was equal to 150.	pass	no further comments.
29	An error was produced and the program terminated.	pass	no further comments.
30	The descendUCT function returned the tree iterator, it, pointing to node a within the tree. This was verified by checking the total payoff of the tree iterator was equal to 150.	pass	no further comments.
31	The descendUCT function returned the tree iterator, it, pointing to node c within the tree. This was verified by checking the total payoff of the tree iterator was equal to 390.	pass	no further comments.

Table E.13: Results : The results of the tests that were used to test bot_degree

Test Number	Actual Outcome	pass/fail	comment
32	A random Gold move was calculated from the board state p, placed in m and executed on n. This was repeated 10 times	pass	no further comments.
33	A random Silver move was calculated from the board state p, placed in m and executed on n. This was repeated 10 times	pass	no further comments.
34	A random Gold move was calculated from the board state p, placed in m and executed on n. This was repeated 10 times	pass	no further comments.
35	A random Silver move was calculated from the board state p, placed in m and executed on n. This was repeated 10 times	pass	no further comments.
36	The decay function returns 0.2399.	pass	no further comments.
37	The decay function returns 0.5646.	pass	no further comments.

Table E.14: Results : The results of the tests that were used to test bot_degree

Test Number	Actual Outcome	pass/fail	comment
38	The two bots, r0 and bot_degree, play 10 games against each other. However around 50% of the time, bot_degree produces a move that is not legal and therefore the Perl script prematurely terminates the game.	fail	See section 8.3 for an explanation of this error as well as how it was reduced in frequency of occurrence.
39	The two bots, r0 and bot_degree, play 10 games against each other. However around 50% of the time, bot_degree produces a move that is not legal and therefore the Perl script prematurely terminates the game.	fail	See section 8.3 for an explanation of this error as well as how it was reduced in frequency of occurrence.
40	The two bots, r0 and bot_degree, play 10 games against each other. However around 50% of the time, bot_degree produces a move that is not legal and therefore the Perl script prematurely terminates the game.	fail	See section 8.3 for an explanation of this error as well as how it was reduced in frequency of occurrence.
41	The two bots, r0 and bot_degree, play 100 games against each other. However around 50% of the time, bot_degree produces a move that is not legal and therefore the Perl script prematurely terminates the game.	fail	See section 8.3 for an explanation of this error as well as how it was reduced in frequency of occurrence.
42	The two bots, r0 and bot_degree, play 100 games against each other. However around 50% of the time, bot_degree produces a move that is not legal and therefore the Perl script prematurely terminates the game.	fail	See section 8.3 for an explanation of this error as well as how it was reduced in frequency of occurrence.

Bibliography

[1] BGA. How to play go. [Online], 2008. Available at: http://www.britgo.org/intro/intro2.html [Accessed 16 October 2008].

[2] C. Cox. *Analysis and Implementation of the Game Arimaa.* PhD thesis, Universiteit Maastricht, Faculty of General Sciences, March 2006. Available at: http://arimaa.com/arimaa/ [Accessed 31 October 2008].

[3] D. Fotland. Building a world champion arimaa program. Canada, 2004. Smart Games. Available at: http://arimaa.com/arimaa/ [Accessed 13 October 2008].

[4] S. Gelly and D. Silver. Combining online and offline knowledge in uct. In *The International Conference on Machine Learning 2007*, Corvallis Oregon, USA, 20th - 24th June 2007. ICML.

[5] S. Gelly and Y. Wang. Exploration exploitation in go: Uct for monte-carlo go. In *Online trading between exploration and exploitation*, Whistler, Canada, 8th December 2006. NIPS.

[6] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of uct with patterns in monte-carlo go. [Online], 2006. Available at: http://hal.inria.fr/inria-00117266 [Accessed 9 November 2008].

[7] A. Hollosi and M. Pahle. Mogo. [Online], 2008. Available at: http://senseis.xmp.net/?MoGo [Accessed 24 October 2008].

[8] IBM. Kasparov vs. deep blue, the rematch. [Online], 2008. Available at: http://www.research.ibm.com/deepblue/ [Accessed 24 October 2008].

[9] A. Junghanns, J. Schaeer, M. Brockington, Y. Bjornsson, and T. Marsland. Diminishing returns for additional search in chess. Technical report, University of Alberta, Dept. of Computing Science, Edmonton, Alberta, Canada, T6G 2H1, 1997.

[10] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. Technical report, Computer and Automation Research Institute of the Hungarian Academy of Sciences, Budapest, Hungary, 2007.

[11] L. Kocsis, C. Szepesvári, and J. Willemson. Improved monte-carlo search. Technical report, Computer and Automation Research Institute of the Hungarian Academy of Sciences, Budapest, Hungary, 2006.

[12] Z. Michalewicz and D. Fogel. *How To Solve It: Modern Heuristics.* Springer, second edition, 2004.

[13] Snippler. core::tree is an stl-like template to implement, for example, a tree of cstrings. [Online], 2008. Available at: http://snipplr.com/view/9719/coretree-is-an-stllike-template-to-implement-for-example-a-coretree/ [Accessed 02 January 2009].

[14] O. Syed and A. Syed. Arimaa: The next challenge. [Online], 1999. Available at: http://arimaa.com/arimaa/ [Accessed 13 October 2008].

[15] O. Syed and A. Syed. Arimaa - a new game designed to be difficult for computers. In *International Computer Games Association*, 2004. Available at: http://arimaa.com/arimaa/ [Accessed 13 October 2008].

[16] Wikipedia. Vorlage: Arimaa diagram. [Online], 2008. Available at: http://upload.wikimedia.org/wikipedia/commons/3/37/37px-Arimaa_board.jpg [Accessed 20 April 2009].

[17] Wikipedia. Vorlage: Arimaa diagram. [Online], 2008. Available at: http://upload.wikimedia.org/wikipedia/commons/thumb/e/e6/Arimaa.JPG/111px-Arimaa.JPG [Accessed 20 April 2009].

[18] H. Zhong. *Building a Strong Arimaa-Playing Program.* PhD thesis, University of Alberta, Department of Computer Science, September 2005. Available at: http://arimaa.com/arimaa/ [Accessed 29 October 2008].

List of Figures

List of Tables