Dots and Boxes – Final Report

Author: Matthew Rea

Supervisor: Dr Martin Caminada

Moderator: Dr Carolina Fuentes Toro

Module: CM2303 – One Semester Individual Project – 40 Credits

School of Computer Science and Informatics, Cardiff University

Abstract

The first focus of this project is creating an implementation of the game Dots and Boxes. This implementation is then used as a testbed for multiple advanced computer players. Some complex computer players have been created, including a Minimax player and a Monte Carlo Tree Search player.

The computer players created were made to compete against each other in tournaments on different sizes of game board, in order to obtain a ranking of the players. The game has also been analysed using these players to determine some important properties of the game.

A ranking of the players shows their relative strengths and weaknesses as well as highlighting key areas for improvement.

The game and all of the players are implemented with Python, using the PyQt5 module for graphics.

Acknowledgements

I would like to acknowledge Dr Martin Caminada, my supervisor, for his support and assistance throughout the duration of this project.

I would also like to acknowledge my friends who helped by playing the game, reading the report, and offering constructive feedback.

Table of Contents

Introduction

This introduction to the report will briefly describe the aims and scope of the project, the approach used to tackle the problem and a summary of important outcomes.

Project Aims and Scope

The overarching aims of this project are to analyse the game of Dots and Boxes and to analyse various game-playing strategies made by game playing agents. The final goal is to produce a ranking of the players that have been implemented and determine what steps need to be taken to improve these players and make them stronger.

The first aim of this project is to create a working and reliable implementation of the game Dots and Boxes that can be played by two players, human or AI.
The second aim of this project is to analyse both the game itself and various strategies used to play the game. This will be achieved using the implementation of the game as a testbed.
The specific 'game playing strategies' to be implemented are: making moves randomly, making moves in a predetermined order, making moves based on a Minimax evaluation of the game tree, and making moves based on a Monte Carlo Tree Search (MCTS) evaluation of the game tree.

The limited scope of this project means that there will be no further attempt to 'solve' the game of Dots and Boxes, as has been done for some board sizes (1). Boards up to sizes of 4x5 have previously been solved; attempting to solve the game for larger board sizes would require a considerable amount more time and resources than are available for this project. It would also require a lot of mathematical analysis, whereas this project will focus on a more practical approach.

Approach

The approach being taken to tackle this problem is a practical one. A significant proportion of the work being done for this project is in the implementation. Creating the game and creating the players for the game will take some time; the rest of the time will be spent analysing the performance of the players and writing the report.

Results will be gathered through repeated testing of each game playing strategy. Each player type will have a strategy they use. To compare them they will be matched against every other player as well as themselves, in order to produce a ranking from best to worst and quantify their relative strength.
All of the planned players, with the exception of the random player, will have parameters that can be tweaked to improve their performance. There will also be repeated testing of these players with modified parameters to determine what the best configuration is for each player. Once the best parameters for each player are discovered, the players can be trialled against each other.
The players will be trialled on multiple sizes of game board. Standard trials will take place on 3x3 and 4x4 boards as these boards have previously been solved, allowing comparison between the results achieved and known results. Trials will also take place on a 5x5 board, as this board has not yet been solved and it will be interesting to analyse the results on this board.

Facing every player against every other is a good way to analyse the relative strengths of each player and is also a good way to analyse the game itself. The data obtained from players playing against copies of themselves can show if a particular board starting position is stronger. If the player who starts the match wins significantly more games than the player who goes second, it will show that player 1 is in an intrinsically stronger position than player 2.

Important Outcomes

There are a number of important specific outcomes for this project based on the aims and objectives.

- Create a working implementation of the game Dots and Boxes.
    - The game should have a title and options screen for choosing game and player settings.
    - The game should be consistent and reliable.
    - This should be playable by human and/or computer players.
- Create working computer players for the game.
    - These should range from basic to complex.
    - There should be a basic player that plays randomly.
    - There should be an advanced player that uses the Minimax algorithm.
    - There should be an advanced player that uses Monte Carlo Tree Search.
    - One of these players should be able to reasonably compete with a human.
- Analyse the players and determine which one performs the best.
    - Determine a ranking of the players.
- Analyse the game and learn about game playing strategies.
    - Determine which strategies for playing are better than others.
    - Determine if player 1 or player 2 is in a stronger position on any game board.
    - Determine what can be done in future to improve the players.

Background

Dots and Boxes

Dots and Boxes, first published in 1895 as 'La Pipopipette' by Édouard Lucas (2), is a pencil and paper game for two players.

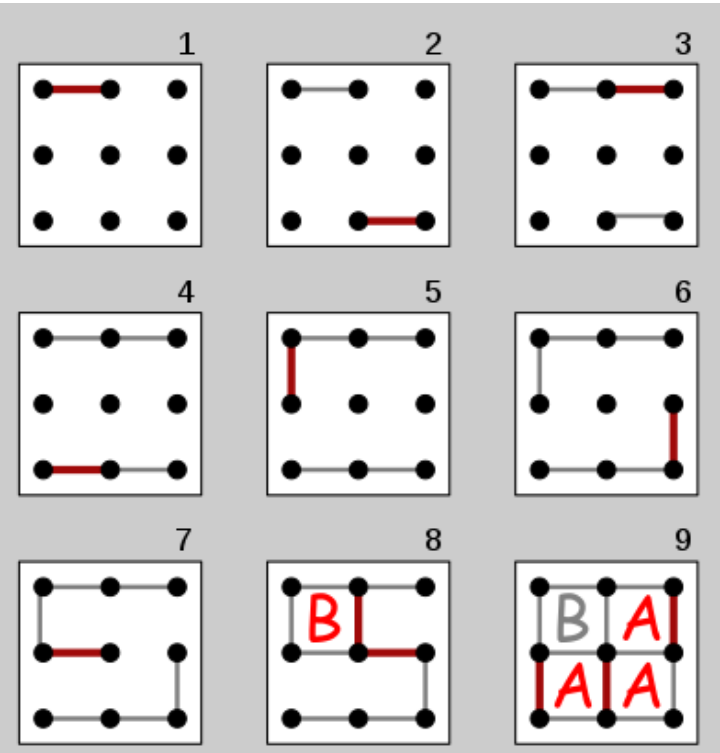

Figure 1 - Full game example of Dots and Boxes on a 3x3 grid. (25)

The game is played on a grid of dots, commonly ranging from 3x3 to 5x5 grids. Two players take turns drawing horizontal or vertical lines to connect adjacent dots. When a player completes the fourth side of a square and encloses the space inside, they 'claim' that box for themselves, usually marking the box with their initial or their player number. An additional rule that is crucial to the complexity of the game is that when a player draws a line to complete a box, they must take another turn afterwards. Players cannot skip turns. The game is finished when all of the lines have been drawn. The winner is the player who claimed the most boxes. (3)

Figure 1 shows a full game of dots and boxes on a 3x3 grid, completed in 9 turns.
Player B is initially mirroring player A's moves, until turn 8 where they claim a box and so get an extra move. With this extra move they play another line which opens the board for player A. Player A then claims the remaining three boxes and wins the game.

Dots and Boxes is a good game to analyse from an Artificial Intelligence standpoint as it is a 'combinatorial' game, which means that is has the following properties: (4)

- Two players. There are typically only two players.
- Zero-sum. The gain or loss of a player is exactly balanced by the loss or gain of the other player.
- Perfect information. The state of the game is fully observable to all players.
- Deterministic. There is no randomness in the development of future states.
- Finite. The number of movements must always be finite.

This makes the game of Dots and Boxes similar to games like Draughts or Chess from a game theory perspective.

Dots and Boxes has already been previously solved for some smaller board sizes, as shown in (1). (5) also presents a winning strategy for player 2 on a 4x4 board. Boards that are larger than 5x5 have not yet been solved, so it is not known if either player can be guaranteed a win on these boards and there is no known strategy for 'perfect play'.

Game Variants

Dots and Boxes is classically played on a grid with all of the lines unfilled, to be played by the player. This is known by some as an 'American board'. There is another popular variant known as a 'Swedish board', in which the lines that make up the edge of the board are filled in from the start. This can lead to more interesting play and quicker games. (6)
As well as these two versions of the game, the author proposes a third variant, a 'random board'. This version of the game will start with a number of the lines filled in from the start, chosen at random. This variant is inconsequential to the analysis of the game – it's simply fun and interesting.

Algorithms

The algorithms chosen and implemented have been picked to help analyse the game itself, and also to demonstrate how effective different strategies can be in the game of dots and boxes.

Minimax

The minimax theorem is a mathematical theorem in the area of game theory, first proven and published in 1928 by John von Neumann. The minimax theorem provides conditions that guarantee that the max–min inequality is also an equality (7).

The minimax algorithm is a game playing algorithm for n-player games that is based on this theorem. The minimax algorithm is designed to minimise the potential loss in any scenario, whilst maximising potential gain. Minimax was originally designed for n-player zero sum games with perfect information and has more recently been extended to complex games and general decision making in the presence of uncertainty. As Dots and Boxes is an n-player zero sum game with perfect information, minimax is a very good fit for this scenario.

```
MinMax (GamePosition game) {
 return MaxMove (game);
}

MaxMove (GamePosition game) {
 if (GameEnded(game)) {
  return EvalGameState(game);
 }
 else {
  best_move < - {};
  moves <- GenerateMoves(game);
  ForEach moves {
    move <- MinMove(ApplyMove(game));
    if (Value(move) > Value(best_move)) {
      best_move < - move;
    }
  }
  return best_move;
 }
}

MinMove (GamePosition game) {
 best_move <- {};
 moves <- GenerateMoves(game);
 ForEach moves {
   move <- MaxMove(ApplyMove(game));
   if (Value(move) > Value(best_move)) {
     best_move < - move;
   }
 }

 return best_move;
}
```

Figure 2 - Minimax algorithm pseudocode (23)

Minimax is typically implemented using main three functions; Maximise, Minimise and Evaluate. In the example in Figure 1 these are MaxMove, MinMove and EvalGameState (not written). The MaxMove function receives a game state. If the game has terminated in this game state it will return a static evaluation of the game using the EvalGameState function. This returns a score for the state the game is in, using specific game knowledge. If the game has not terminated then all possible moves from this position are generated and iterated through. For each move that can be made a new game state is generated, simulating the player making this move. This new game state is then passed to the MinMove function, which will perform the same operations with one key difference. When the bottom of the tree is reached and the game states are evaluated, the scores are returned and passed back to these functions. The MaxMove function will save the move that returned the highest score, whereas the MinMove function will save the move that returned the lowest score. These represent opposing players making moves. The MaxMove function represents the player making the best move they can possibly make and the MinMove function represents their opponent making the best move they could possibly make. In the case of a zero-sum game the best move for an opponent will correspond to the worst move for the player.

Figure 3 shows the 'state tree' that is being traversed by this recursive algorithm. At the first step, the player is analysing their own moves, and intends to maximise their potential score. These moves are represented by green triangles pointing upwards. At the second step, the player is analysing their opponents' potential moves, and assuming that they want to maximise their own potential score. These moves are represented by red triangles pointing downwards. This tree expands downwards as more moves are predicted, with each level of the tree corresponding to a move made by either the player or the opponent, and a minimising or maximising step.

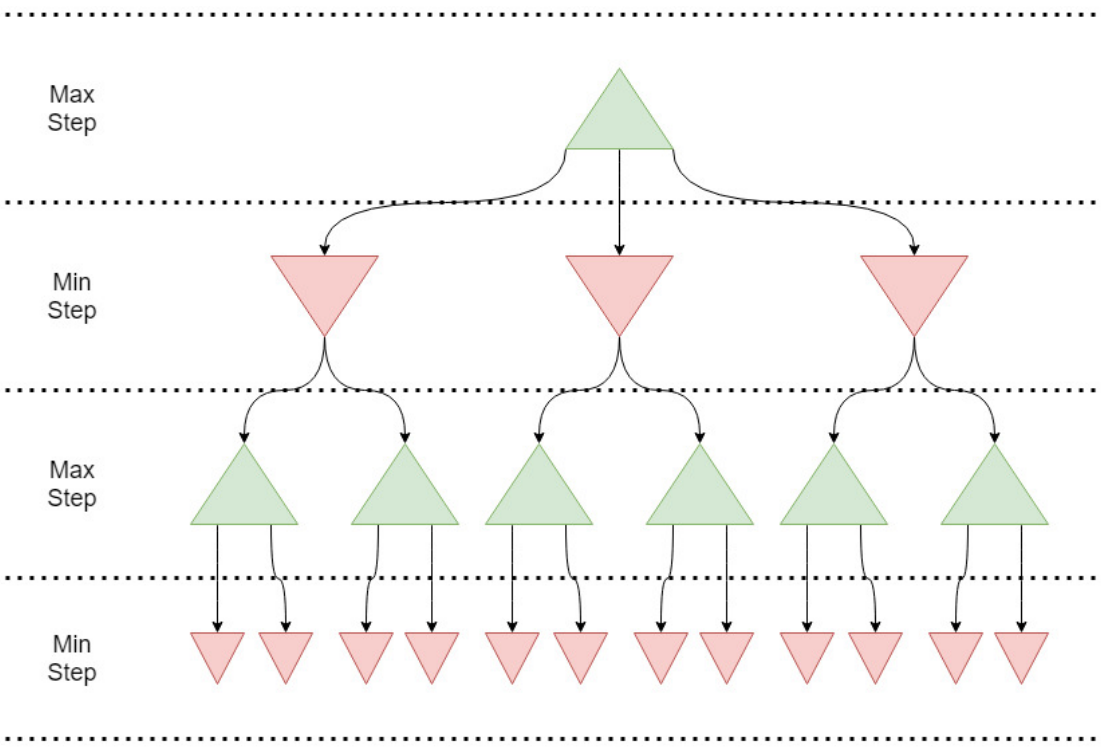

Figure 3 - Minimax state tree diagram

The Minimax algorithm described in Figure 2 is a very basic version of the game playing minimax algorithm.

```
// Search game tree to given depth, and return evaluation of
// root node.
int NegaMax(gamePosition, depth)
{
        if (depth=0 || game is over)
                // evaluate leaf gamePositionition from
                // current player's standpoint
                return Eval (gamePosition);
         // present return value
        score = - INFINITY;
        // generate successor moves
        moves = Generate(gamePosition);
        // look over all moves
        for i =1 to sizeof(moves) do
        {
                // execute current move
                Make(moves[i]);
                // call other player, and switch sign of
                // returned value
                cur = -NegaMax(gamePosition, depth-1);
                 // compare returned value and score
                // value, update it if necessary
                if (cur > score) score = cur;
                // retract current move
                Undo(moves[i]);
        }
        return score;
}
```

Figure 4 - 'NegaMax' algorithm pseudocode

Figure 4 shows a simplified version of Minimax that exploits the property

$$Max(a, b) \equiv -Min(-a, -b)$$

This leads to the NegaMax algorithm, which simply negates the value returned from the recursive call to NegaMax. In the case of a game in which players strictly make alternate moves this produces the exact same result as Minimax. This simplifies the algorithm as it removes the need for separate MinMove and MaxMove functions.

This algorithm also introduces the concept of a depth variable. This allows the calling function to control the depth of the search in the game tree. When NegaMax is called it will be passed a positive integer, when NegaMax is called again, the depth is lowered by 1. When the depth value reaches 0 an evaluation of the game state is made and returned. This is the version of the algorithm that will be adapted for use in this project.

There are further optimisations that can be made to the Minimax algorithm. Two of these will be discussed later in the 'implementation' section.

Monte Carlo Tree Search

Monte Carlo Tree Search is a heuristic search algorithm for decision processes, most notably employed in game playing software. Monte Carlo Tree Search has been used in the famous 'AlphaGo' program for playing the board game Go (8), other board games like Chess and Shogi (9) and in turn based strategy games such as Total War: Rome II (10).

The Monte Carlo Method is a method which dates back to the 1940s that uses randomness to help solve difficult deterministic problems. The method relies on repeatedly using random sampling to find approximate results in problem spaces; as such, the accuracy often increases with increased numbers of samples. (11)

In 1987, Bruce Abramson modified the Minimax search algorithm to use an expected outcome model based on random game playouts – a Monte Carlo method – instead of the usual static evaluation function. (12) This method of searching was developed further to include recursive rolling out and backtracking and later introduced the Upper Confidence Bound (UCB) heuristic for constructing trees. (13) In 2006, Rémi Coulom coined the term Monte Carlo Tree Search (MCTS) when he applied Monte Carlo search to game trees. (14)

The principle of operation of MCTS is to analyse the best moves that are available to the player. A state tree is constructed with nodes corresponding to potential moves that can be made. These moves are discovered by the selection process and analysed by random playout. The selection of moves is made using the UCB formula for exploitation and exploration. The formula for calculating UCB for each individual move is;

$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$ where;

$w_i = win\ score\ for\ this\ node,$

$n_i = number\ of\ times\ this\ node\ has\ been\ visited$

$N_i = number\ of\ times\ the\ parent\ of\ this\ node\ has\ been\ visited$

$c = exploration\ coefficient. Tuned\ experimentally.$

This calculates a value for UCB with two parameters. The first parameter is the exploitation parameter, which is simply a ratio of how many winning states have been discovered after this node and how many times this node has been visited. The second parameter is the exploration parameter, which factors in how many times this node has been visited in total compared to its parent node. The exploration parameter is scaled by c, which allows the algorithm to be tuned to explore more or explore less depending on the use case. (15)

One iteration of the MCTS algorithm includes four steps;

1. Selection: The selection will start at the root node and select the child of this node with the highest UCB value. This process continues, selecting the child of the current node with the highest value until a leaf node is reached. A leaf node is a node which has no children.
2. Expansion: Once a leaf node has been found and selected, create children from this node corresponding to all of the possible moves that could be made from this game state.

3. Simulation: Choose one of these new child nodes and perform a ‘rollout’. This is a simulation of the game playing out until the end. The basic implementation of this is a random playout.
4. Backpropagation: Take the result of the simulation and backpropagate the score up the tree, all the way to the root node. (16)

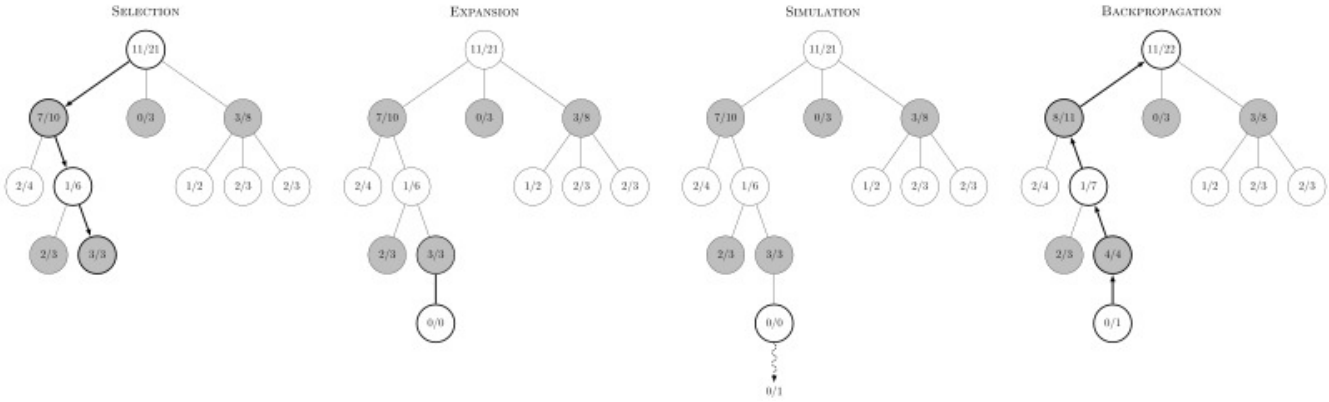

Figure 5 - Four steps of the MCTS algorithm (24)

This is the principle of MCTS. Other implementations that incorporate advanced features like domain specific knowledge & heuristics and advanced techniques like reinforcement learning and deep learning have become some of the world’s leading game playing programs.

Approach

The approach taken to the problem of analysing Dots and Boxes is a practical, experimental one. Firstly, an implementation of Dots and Boxes is needed. Next, a system with which any type of computer player can understand and play the game. Then, a variety of computer players are needed for benchmarking, testing and analysis. The game will also have a Graphical User Interface (GUI), so that humans can play the game and so that the games can be visualised.

To create the game, player system and players, Python will be used. There are many merits of using Python for this project; The author is familiar and comfortable with programming in python, vastly increasing the speed of development. Python can be used to neatly combine multiple programming methodologies in the same program, allowing the use of both procedural and Object Oriented (OO) paradigms. There is also an abundance of libraries that can be imported and used to solve general issues so it is not necessary to solve every problem from scratch.
The PyQt5 module will be used to handle the graphics for the game. (17)

Whilst creating a system that includes the game, the GUI and the players, the main intention is to create this system to be as loosely coupled and as modular as possible. This means that the interactions between the game, the GUI and the players will be as limited as possible. This is so that each part of the program is separate from one another, which will simplify each implementation and also make creating different player types easier.
For this reason, using an Object-Oriented approach is the most sensible choice. The OO concepts of encapsulation, inheritance and polymorphism make creating a loosely coupled system like this far more feasible.
Encapsulation is the concept of ‘wrapping up’ code and data in the same objects and making this private, so that it cannot be modified by other objects external to this one, except through custom public methods designed for access. What this means in terms of this project is that the object that represents the game cannot be modified by the players or by the GUI, it can only be modified via

request. When a player takes a turn in dots and boxes it will not modify the game directly, it will request to make a move and the game will either make the move if it can and modify itself or reject the move if it is illegal.
Inheritance in OOP is the ability to define new classes of objects based on other classes of objects. When one object inherits from another, it takes all of the data and code from the parent object and can then expand on it. This leads to hierarchies of objects going from 'general' at the top of the hierarchy to 'specific' at the bottom. In the case of this project, a 'general' object will be an abstract 'player' object. This will represent any type of player and will define the interface that the game and GUI can have with a player object. More specific objects that inherit from 'player' will be 'Minimax Player' and 'Monte Carlo Player'. This leads on to polymorphism.
Polymorphism is a property that objects can have that means one part of a program does not need to know what it is interacting with, simply how to interact with it. This means that the program can be interacting with any type of player, it does not know which, and it will still be able to interact with it effectively. In the case of this program, each player can be asked to produce a move to be made in the game and the methods to decide which move is selected can be vastly different from one another, yet the game does not need to know how this move is produced.

Another important concept used is the 'factory pattern'. This method simplifies the creation of objects and promotes polymorphism. To use this creation pattern, an interface must be defined that a part of the program knows how to interact with. This interface will be common to all of the objects created by the factory. The factory can then provide concrete objects for the program, without the program knowing the specific implementation of the object or how to create the object itself. (18) In this project a factory pattern will be used to create different types of player. When instantiating the game and the players, the controlling code will not know how to create each different type of player. The controlling code will be provided with a factory which knows how to create players and the controlling code can then 'ask' the factory for a type of player. When asked, the factory will create the player and return it to the program under a common player interface. This will reduce program complexity and code duplication, as the code for creating specific player types is only in one place, in the factory. The code will not need to be duplicated in every place that players need to be created.

Implementation

The general structure of the program is described in this UML class diagram:

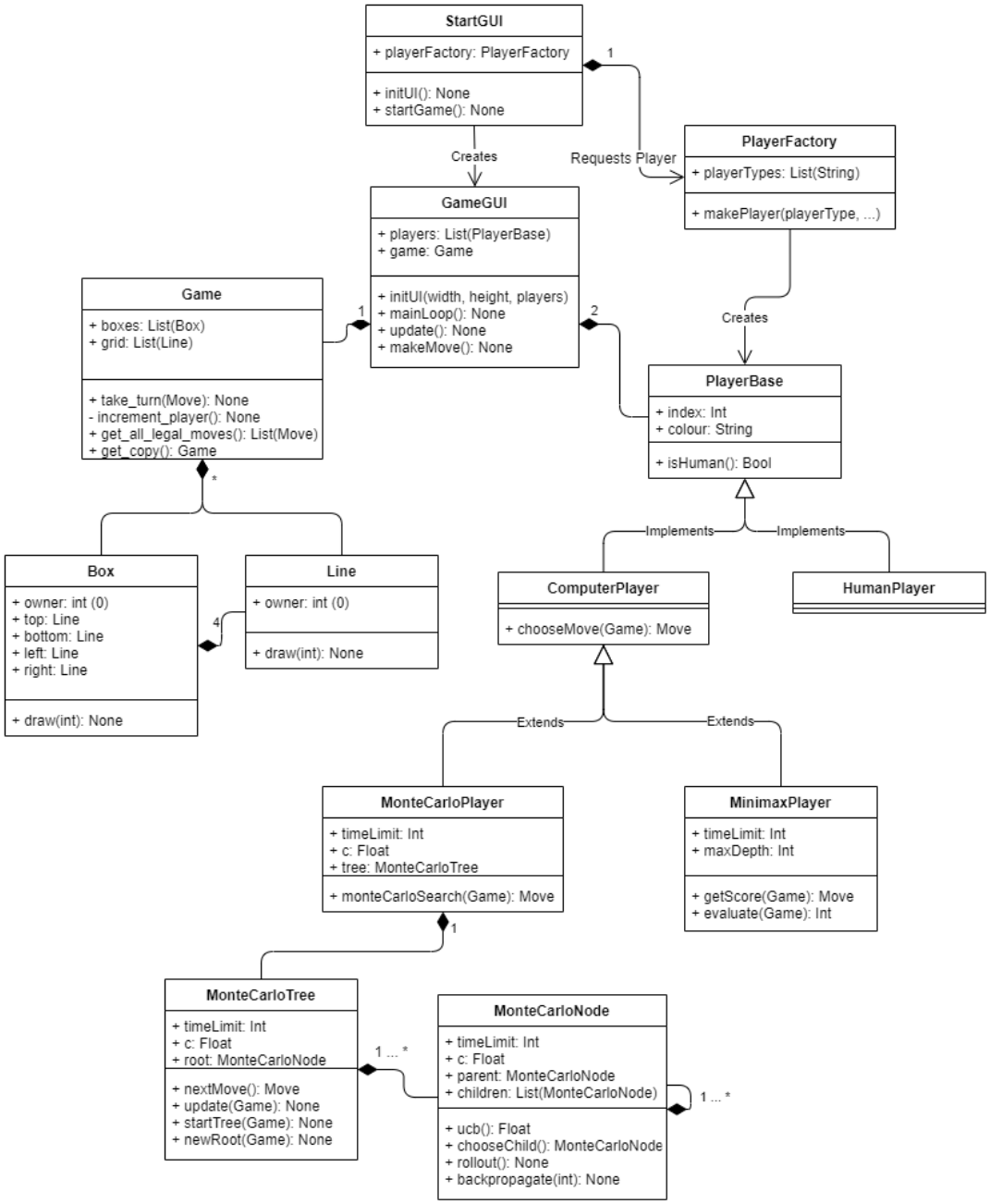

Figure 6 - UML Class Diagram for Dots and Boxes

This diagram describes the structure and interactions between each class in the program. The basic operation of the program is as follows:

- StartGUI is created with an instance of PlayerFactory. Here the user can enter game options such as size and variant and player options such as type and parameters.
- On game creation, StartGUI will ask PlayerFactory for two players which are chosen by the user. The PlayerFactory will return these to StartGUI. StartGUI will then create a Game instance. Finally, StartGUI will create a GameGUI instance, and will pass these player and game instances to it as start parameters.

- The GameGUI instance can read the state of the Game instance and draw this to the screen. It will then enter a main game loop to control move making. The game loop is as follows:
    - The GameGUI instance reads from the Game instance which players' turn it is. Then, depending on what is returned from this players' isHuman() method, it will do one of two things.
        - If this player is a human, the GUI will make the buttons that correspond to legal moves clickable. When a button is clicked, this will correspond to the move the human player wants to make.
        - If this player is not a human, the GUI will call the players' chooseMove() method. The GUI will pass a deep copy of the game to the player so it can make a choice. The player will then return a move to the game.
    - The GameGUI instance then has the move that the player wants to make. It will send this move to the Game instance via its take_turn() method. If this is a legal move then the Game instance will update.
    - The GUI will then read the game instance again and update its display. The loop then returns to the start and will ask the next player for their move.
- Once the Game instance has no moves left to be made, the GUI will read the scores from the Game instance and will then declare the winner.

This method of control and display creates a loosely coupled relationship between the Game and the Players. The Players are only ever given a copy of the game so they can never influence or modify the real version of the game. The players will send a move back to the GameGUI which will pass it on to the game. The Game itself is never aware of the players who are playing the game, it is only aware of the moves being made.

A large advantage of this design is the ease with which the program can be decomposed. Each section of the program was created independently, in isolation of the others. This allowed an agile method of development to be used with each section being developed, tested and verified before moving on to the next.

Game

The first challenge to overcome when creating the game itself was how to represent the game. The game is called 'Dots and Boxes', however when the game is played the most important elements are actually the lines, with the boxes coming after. Thinking about the game in this way created a simple dependency. Users play the game with lines, and boxes are dependent on the lines that have been played. This can be seen in Figure 6; the Game object consists of Line objects and Box objects and the Box objects consist of 4 Line objects each.

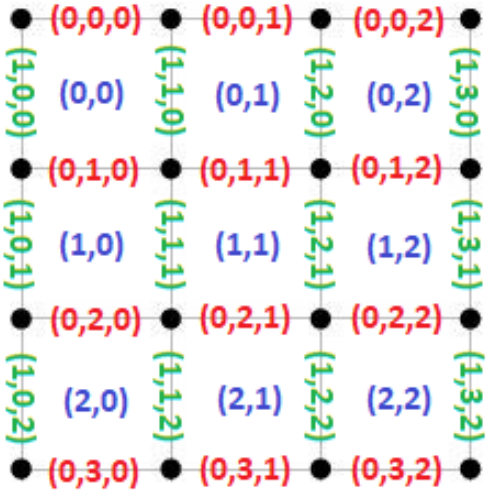

Figure 7 - Line and Box indexing scheme

The next challenge was how to represent this information. The obvious choice was to place all of the Line objects in an array, but the best way to structure this array was not obvious. When creating the game the intention was to make it possible to create non-square grids; this eventually led to the decision to split the lines up into two main arrays: horizontal and vertical. These were then split up further into two dimensional arrays such that adjacent lines – lines that share a dot – are in the same array, and opposite lines – lines that share a

box – are at the same index but are in the next array. This led to the structure in figure 7. The red horizontal lines are indexed first left to right and then top to bottom. The green vertical lines are indexed top to bottom and then left to right. These are placed in a three-dimensional array, with the first index representing the horizontal or vertical collections of lines.

It was decided to represent the boxes using a simple two-dimensional array starting from the top left. This means that when accessing the boxes it is easy to find the lines that the box is dependent on. The line to the top of any box (i, j) will be at index (0, i, j), the bottom at (0, i+1, j), the left at (1, j, i) and the right at (1, j+1, i).

To play the game the players must make moves and these moves equate to the lines in the game. This means making a move is the same as choosing a line. The simple solution to making moves was to simply send the index of the line to play to the game. This means moves are made by sending tuples to the game in the form (o, i, j).
When the game receives a move it first checks if the move is legal. If it is a legal move then the line that has been selected will have its 'owner' property set to the number of the player taking the move. After this, the game will check the boxes associated with this line to see if they have been fully enclosed by lines. This is done by looking at the index of the move and calling the check_completed method of the one or two boxes that are connected to this line. As each Box object stores a reference to the four lines it is comprised of it can check all four of them to see if they have all been claimed. If no box has been completed this turn then the internal player counter will change to the next player. If a box has been completed then the rules state that the same player gets another turn, so the internal player counter will not increment.
This process of making moves, checking boxes and incrementing the player counter encompasses all of the rules of the game. Players can make moves if they are legal, the boxes are claimed as points for each player, and players can take turns in the order they should.

The process of checking if a submitted move is legal or not is simple; the game has an internal list of all of the possible legal moves that can be made and if a move is in this list, it is legal. This list of legal moves is generated at the start of the game when the grid is created. When moves are successfully made they are removed from the list. This method of saving all of the legal moves at the start of the game and then removing them from the list as they are made is a time saving effort. If it were necessary to check all of the moves to see if the current move is legal every time then the entire grid of lines would need to be checked every turn.

```
>>> g1.print_grid()
*_ _*_ _*_ _*
|   |   |   |
*_ _*_ _*_ _*
|   |   |   |
*_ _*_ _*_ _*
|   |   |   |
*_ _*_ _*_ _*
>>> g2.print_grid()
*___*_ _*_ _*
| 2 |   |   |
*___*_ _*_ _*
|   |   |   |
*_ _*_ _*_ _*
|   |   |   |
*_ _*_ _*_ _*
>>>
```

Figure 8 - Example of game print_grid method

To make the representation of the game as lightweight as possible, the game itself does not have anything to do with the display of the game or the GUI. This initially made development difficult as the GUI was created after the game was finished. To aid development and debugging the game includes a method print_grid, which prints a simple ascii representation of the game grid. This can be seen in Figure 9, where the first grid is an empty starting grid and the second grid has had 4 moves made, (0, 0, 0), (0, 1, 0), (1, 0, 0) and (1, 1, 0). These moves claim the top-left box in index (0, 0) for player two.

As the game was being developed for complex computer players to play, it required a method of returning a deep copy of itself. This is a copy of the game in exactly the same state, but completely separate from the original so that modifying the copy does not affect the original. As this method may be called many times whilst a computer player is calculating their move,

this method needed to be as fast as possible. For this reason a game can be constructed initially with only a height and a width input and can also be constructed using a number of input parameters. When a copy is made it will send the current player value, list of legal moves and the arrays of lines and boxes to the copy. The copy will then construct itself using a different constructor method that copies the owner values of the lines and boxes, rather than using the same arrays.

GUI

The GUI was developed after the game had already been implemented. This means the GUI could be made to be dependent on the game whilst the game is not dependent on the GUI at all. The GUI consists of two main components; the Start Frame and the Game Frame. The Start Frame is the first window that comes up when the game is launched and allows the user to pick all of the options that are available for the game.

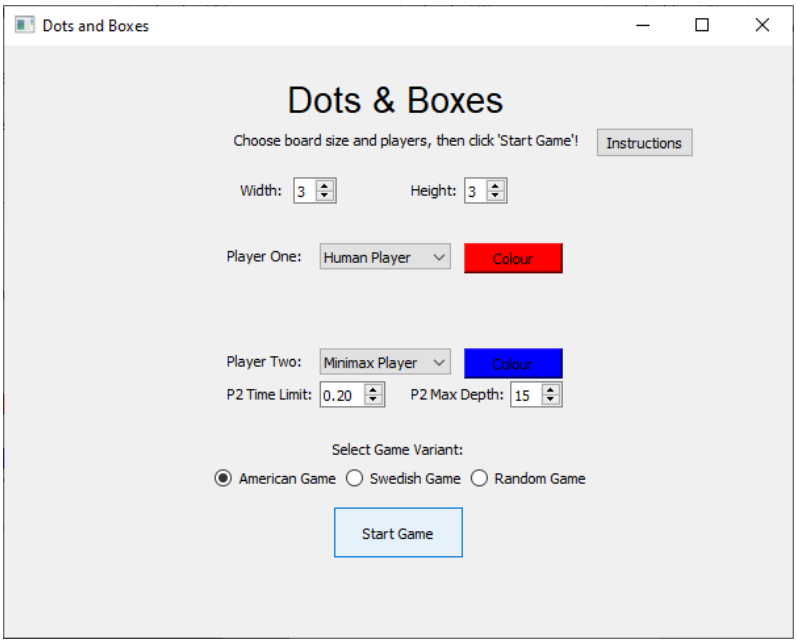

Figure 9 - Start Frame GUI

The specific inputs have been chosen so as to eliminate the possibility of a user entering bad data. The inputs that control grid size are integer only and can only be between 3 and 10. The player choice inputs are dropdowns that are populated by stored values in the player factory. When selecting an advanced computer player – either minimax or monte carlo – inputs appear to change the parameters for these specific types of player. There are also colour pickers which are completely cosmetic choices that simply make the game feel nicer, and a variant radio box group, which can switch between the different variants of the game.

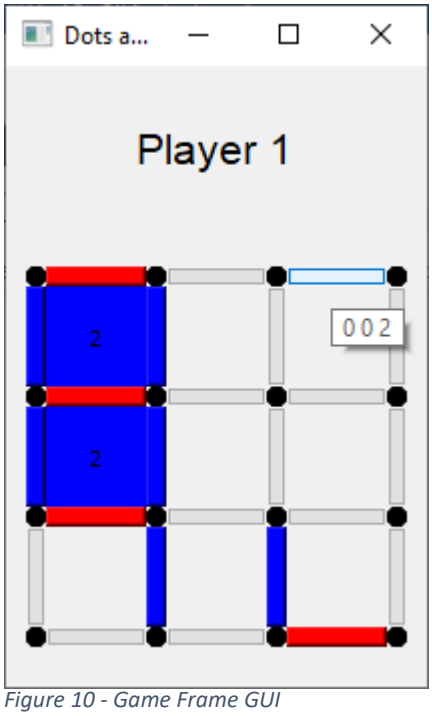

Figure 10 - Game Frame GUI

When the 'Start Game' button is clicked in the start frame, the game frame part of the GUI is created in a separate window. This frame is sent the player and the game objects from the start frame and displays the game grid as a series of buttons. These buttons are dynamically made and placed on initialisation after reading the grid from the game object. After this point, the player or computer player can choose their moves and they are visualised as the buttons being filled in with their colour, as well as becoming permanently unclickable. The buttons are unclickable by default, but when it is a human players' turn the GUI will read the state of the game to find the legal moves and it will make the corresponding buttons clickable. This limits the possible inputs that can be made by the player strictly to the legal moves available. As can be seen in Figure 10, each button can display its grid index when hovering over it, as a tooltip.
The boxes are filled in with both colour and the player number when a player captures them.

One big advantage of having a GUI dependent on the game that has not yet been discussed is the ease of changing player types. With this implementation it is possible to play a game with two human players that take turn clicking buttons, one human player who will click and one computer player who will decide their own moves, or two computer players who both play independently.

Game Variants

As previously discussed, there are two 'official' versions of the game – 'American' and 'Swedish'. There is also the third variant proposed by the author – 'random'. These variants have all been implemented using inheritance. Both the Swedish and Random variants are subclasses of the American variant, only with a modified constructor that fills in the required lines as if they are owned by a non-player. These lines are filled in with the colour black.

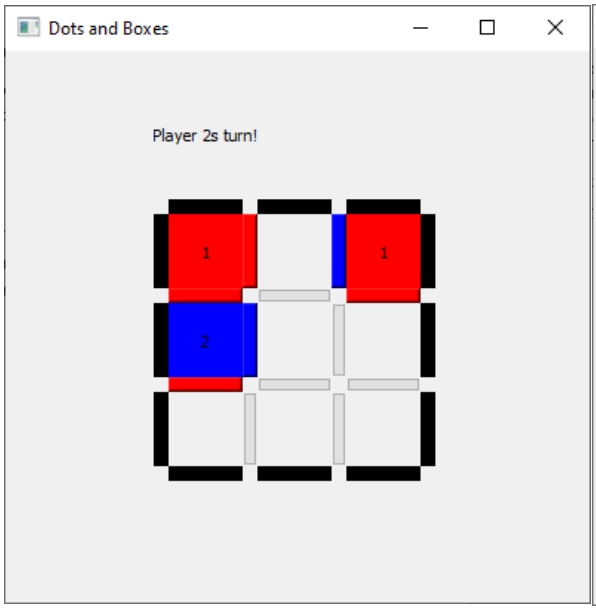

Figure 11 - 'Swedish board' variant

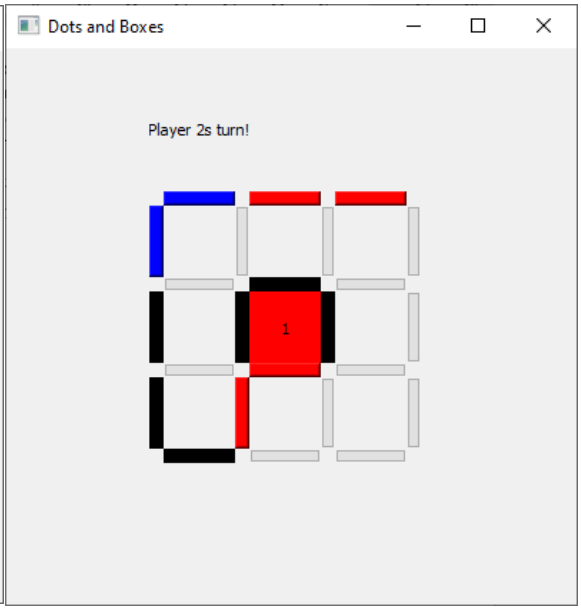

Figure 12 - 'Random board' variant

Player System

The player system is based on the hierarchy of players that can be seen in figure 6. Every type of player inherits from the generic class PlayerBase, meaning they can share a common interface. The HumanPlayer class is an empty implementation, with the only difference being that its 'isHuman' method returns True. The ComputerPlayer class acts as a parent class for all of the computer players, which return False from their isHuman method.

The code for the Player Factory – shown in Figure 13 – mostly consists of a simple if-elif-else statement that can switch between the different string representations of each player. If the 'playerType' variable passed into makePlayer is not in the internal list of players the factory simply returns a default Human Player. This prevents the program from crashing if the factory receives a bad input. The factory can construct each different type of player along with the parameters they require and returns this from the makePlayer method. Because of this, any type of player can be made while only requiring the string name and an index. The factory can fill in any missing parameters if they are needed.
As the factory also holds the list of player types hard coded, the dropdown selection boxes in the StartFrame GUI can be populated directly with these. This allows new player types to be added and removed only by updating the factory, without needing to change the GUI.

```
class PlayerFactory:
    """
    Basic player factory that stores a list of all player types and can return
    corresponding player instances.
    """
    def __init__(self):
        self.playerTypes = ["Human Player", "Random Player", "In order player", "Minimax Player", "Monte Carlo Player"]

    def makePlayer(self, playerType, index, colour="red", timeLimit=None, maxDepth=20, c=1.4):
        if playerType == "Human Player":
            return HumanPlayer(index, colour)
        elif playerType == "Random Player":
            return RandomPlayer(index, colour)
        elif playerType == "In order player":
            return MovesInOrder(index, colour)
        elif playerType == "Minimax Player":
            if timeLimit is None:
                # switch so that different players have different default time limits.
                timeLimit = 1
            return MinimaxPlayer(index, colour, timeLimit, maxDepth)
        elif playerType == "Monte Carlo Player":
            if timeLimit is None:
                # switch so that different players have different default time limits.
                timeLimit = 5
            return MonteCarloPlayer(index, colour, timeLimit, c)
        else:
            return HumanPlayer(index, colour)
```

Figure 13 - Code for Player Factory

Players

There are 5 player types in the game. These are Human, Random, In-order, Minimax and Monte Carlo.
The Human player class is an empty class. It needs no implementation as the GUI simply lets the human click the button they want.
The Random player class is a basic computer player. When its chooseMove method is called and it is passed a copy of the game, it uses Python's built-in random module to return a random choice from the list of legal moves.

The In-order player is also a basic computer player. It simply returns a move from the list of legal moves, from a predetermined index. This is the most basic strategy but can be modified by changing the order in which moves are returned.
The Random and In-order players serve as basic first steps in creating the more advanced players, and also serve as benchmarks for testing and comparing the more advanced players.

Minimax Player

The Minimax player was made by learning from and modifying an implementation demonstrated in a lecture series for an Artificial Intelligence module. The original implementation that was used to teach was for the game Pacman, written in Java. (19)

This player was a basic Minimax player, consisting of chooseMove, MinMove, MaxMove and Evaluation methods. However, it was quickly understood that this approach would not work for Dots and Boxes due to the irregular turn order that the rules create. Figure 3 shows an 'ordinary' Minimax game tree, in which the players take alternate turns. The game tree for a particular game of Dots and Boxes however looks more like Figure 14. Rather than having strict alternating turns, some moves are capturing moves. These moves, marked with an asterisk in the diagram, are moves where the player captures a square, meaning the capturing player gets another turn. This makes the minimax implementation more complex as the same depth of game tree can contain moves from either player.

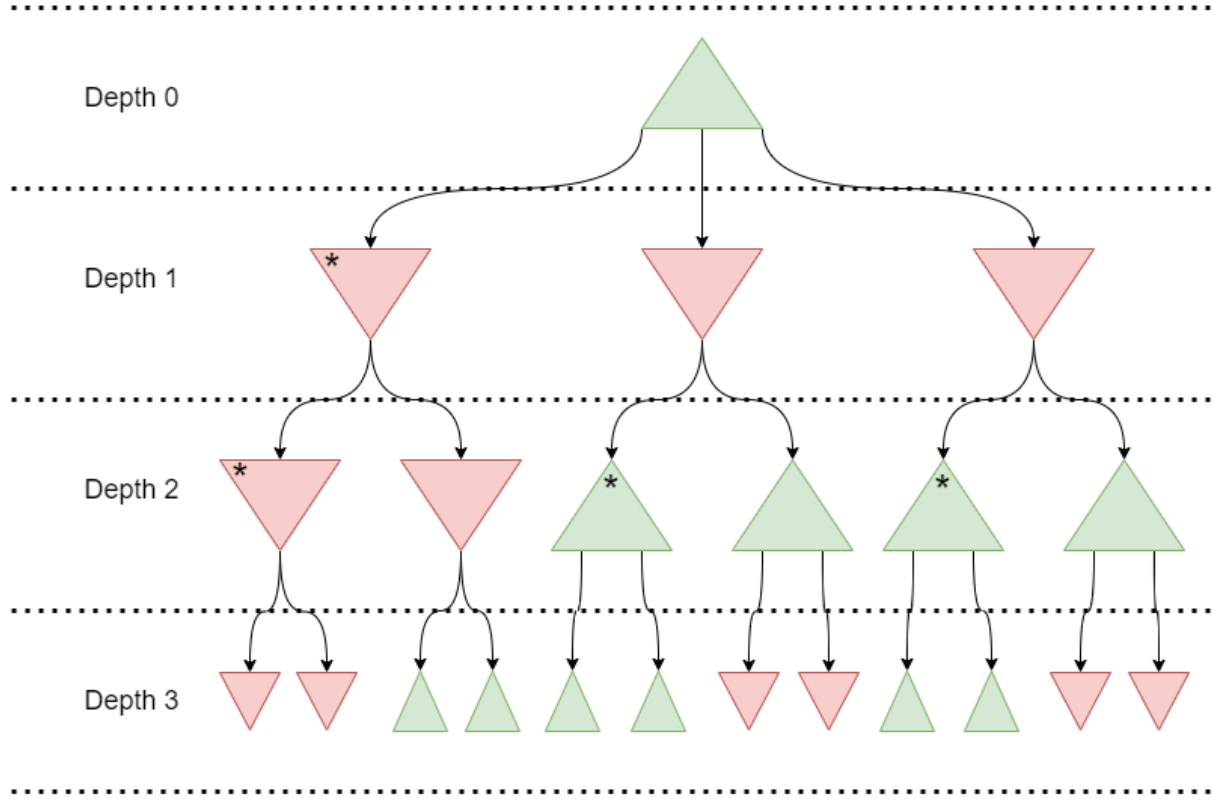

Figure 14 - Minimax state tree for Dots and Boxes

This problem of inconsistent turn order was solved by using and adapting the NegaMax algorithm shown in Figure 4. The NegaMax algorithm uses one recursive method to explore the game tree, rather than the two methods used in the basic Minimax algorithm. The NegaMax algorithm will simply negate the value returned from its recursive calls, causing the Min and Max steps to alternate. The solution used in the Dots and Boxes implementation takes this and introduces a conditional step in order to determine whether or not the returned value should be minimised or

maximised. This conditional step, highlighted in Figure 15, determines whose turn it is in any particular game state. If the Minimax player is taking a turn, then the returned values are maximised; if the player taking the turn is not the minimax player, the returned values are minimised. This conditional step also affects another optimisation included in the algorithm, Alpha-Beta pruning.

```
# When we're at the bottom of the tree, return static evaluation
if depth <= 0 or game.is_finished():
    return self.evaluate(game)

moves = game.get_all_legal_moves()
# Store the current player
currentPlayer = game.currentPlayer
# set bestScore to either high or low value depending on whose turn it is
# also set maximise to True or False
if currentPlayer == self.index:
    bestScore = -10000
    maximise = True
else:
    bestScore = 10000
    maximise = False
for move in moves:
    # Make the move and get the next state of the game.
    copyGame = self.makeMove(game, move)
    # recursive call
    score = self.getScore(copyGame, depth-1, alpha, beta)
    # Different actions depending on wether this is a min node or max node
    if maximise:
        bestScore = max(score, bestScore)
        alpha = max(alpha, bestScore)
    else:
        bestScore = min(score, bestScore)
        beta = min(beta, bestScore)
    # Alpha - beta pruning.
    if beta <= alpha:
        break
return bestScore
```

Figure 15 - Minimax implementation. Conditional elements highlighted.

Alpha-Beta pruning is a method used to 'prune' the branches of the game tree under certain conditions (20). The algorithm keeps track of the best score so far, just as Minimax does, and also keeps track of two values, alpha and beta.
The alpha value is the best value that the *maximising* steps can guarantee for the player so far.
The beta value is the best value that the *minimising* steps can guarantee for the player so far.
The alpha value starts at negative infinity and the beta value starts at positive infinity. These values are updated with the best score returned so far from their respective steps. This means that alpha keeps a record of the highest scoring move found and beta keeps a record of the lowest scoring counter move found so far. If the value for beta ever falls below the value for alpha, it indicates that this branch of the tree cannot possibly find a better value than it has already. This is true as it is assumed the minimising player will always pick the lower value, meaning any higher value found in this branch will be unobtainable. This means the branch can be pruned and the algorithm will not waste time exploring this part of the tree.

Another optimisation that has been applied to Minimax is called Iterative Deepening. This concept is usually applied to Depth First Search to improve its runtime, but here has been applied to Minimax. Iterative Deepening is the concept of iteratively increasing the depth of the search. This means that initially the search starts with a max depth of 1, meaning it will search 1 move ahead and then return the best move. Once this is completed, the max depth is incremented by 1 and the search will be performed again to a max depth of 2.
The main reason for this optimisation being applied is so that Minimax can adhere to a time limit. Without this extra consideration Minimax would only be able to search to a fixed depth and would take a variable amount of time to complete this search. Using iterative deepening means that the search can be performed to the max depth attainable in the time allowed. This turns Minimax into an 'anytime algorithm'. The best score returned from any of the calls to getScore will be returned as the best move.
This optimisation also makes Minimax adaptable. If the search space is large then Minimax will not be able to search to a great depth, however if the search space is small then Minimax can search to a greater depth in the same amount of time. Implementing iterative deepening means this adaptivity

does not need to be defined, and the algorithm will always do the best it can do in the time it has available.

The evaluation function is an important part of the Minimax algorithm. An evaluation function must be tailored to the game being played if it is to be effective, it is not always enough to simply count the scores in the state being evaluated.
The evaluation function created for this Minimax implementation is variable, meaning that it can produce a different result from the same board depending on whose turn it is. This is important for Dots and Boxes as the same state on a different turn can be very desirable or very undesirable. Generally, if there is a box with three sides completed then this will lead to one of the players scoring on the next turn. In the evaluation function, if it is the Minimax player's turn then we will increase the heuristic score for a board with boxes that have three sides, as this means the Minimax player can capture this box and increase their score. If we are evaluating a game state in which it is the other player's turn then we want to avoid leaving three sided boxes as the other player could capture the box and increase their score.

Monte Carlo Player

The implementation of Monte Carlo Tree Search (MCTS) was created following guides found online. The algorithm itself was learned in (21) and the class structure was inspired by (22). This class structure can also be seen in Figure 6. The Player class MonteCarloPlayer contains an instance of MonteCarloTree. This instance of MonteCarloTree is populated with multiple MonteCarloNode instances. The methods that make up the MCTS algorithm are contained in MonteCarloTree and in MonteCarloNode.

The implementation created for this project works on the same principle as MCTS described in the guides, however the methods and control logic operate slightly differently.

```
while timeTaken <= self.timeLimit:
    if current.game.is_finished() or current.n == 0:
        # the rollout method also handles the backpropagation step.
        current.rollout()
        # after rollout reset to root.
        current = self.root
        no_iterations += 1
        # recalculating here saves a little bit of time.
        timeTaken = time.time() - startTime
    # the next node is the best child of the current node.
    current = current.chooseChild()
    # that's it that's the algorithm
```

Figure 16 - MCTS Main loop implementation

This is the main loop for MCTS. This controls the 'Selection' step of the algorithm and can initiate the 'Simulation' and 'Backpropagation' steps.

Each pass of the loop will first check if the current node has not been visited, or if the game in that state is finished.
If these conditions are not true then the best child of this node will become the new current node. This method, chooseChild, also controls the 'Expansion' step of the algorithm. If the node has no children when its chooseChild method is called then it will create children for itself and return one of these.
If the current node has not been visited before (n == 0), then the algorithm will perform a rollout from this node. A rollout consists of making a copy of the game, getting the list of all moves left to be made, shuffling this list, and then playing all of the moves. At the end of the rollout method, the node will call its own backpropagate method. This will increment its visit counter (n) and will update its win counter (t). The backpropagate method then calls the backpropagate method of its parent node. This will ensure the values are backpropagated up the entire tree all the way to the root. Once this is complete, the main loop will set the current node back to the root node and start again. This is

performed until the time limit is reached, at which point the child of the root node with the highest UCB value is returned.

All of the guides used to create this implementation of MCTS have one drawback; they all are limited to making an algorithm that can choose the best move when given any state. None of the implementations or any of the literature found online considered creating a consistent game player that would play an entire game start to finish.
A novel optimisation of an MCTS player was developed with this consideration in mind. This optimisation was not found anywhere online or in research papers and so may be a new technique. The optimisation works as follows;

- When starting a new game and creating a new player an empty instance of MonteCarloTree will be created.
- On the first turn of this player, the tree will be given a copy of the game state. This will become the root node of the tree.
- On every subsequent turn of this player, rather than discarding the old tree and creating a new one from scratch, the player will traverse the existing tree and locate the node corresponding to the new state of the game. This node will become the new root of the tree.

This optimisation can save computation time throughout a game, and it means that a player using this strategy will be slightly stronger towards the end of the game. As the tree is explored, each node gathers data about the number of visits and wins from this position. As the existing tree already contains some data for each node it is efficient not to discard this data where it is relevant.

```
newRoot = self.root
# this finds which moves have been made between the root and the new state.
movesMade = game.movesMade[len(self.root.game.movesMade):len(game.movesMade)]
# go through each move in order
for move in movesMade:
    if newRoot.children:
        # if this node has children, find the one that corresponds to the move made
        for child in newRoot.children:
            if child.move == move:
                # then make this the new root node
                newRoot = child
                break
    else:
        #print("Building new root")
        # if the node doesn't have children then make a fresh new root node
        newRoot = MonteCarloNode(self.index, game, (0,0,0), "NewRoot", self.c)
        newRoot.makeChildren()
        break
```

Figure 17 - Code to replace root node.

The code that traverses the tree to find the new root node is in Figure 17. The game object internally saves a list of all moves that have been made in the game so far. This code uses the lists in the current root and the new game state to determine which moves have been made between the root state and the new state. It then goes through each node in turn, finding the child nodes that correspond to the moves that have been made. Once there are no more moves, the new root node has been found and this can now be used to perform MCTS. If any of the nodes do not have any child nodes, a new root node will be built and used.

Testing

To ensure that the game is reliable and consistent, even after changes are made, a set of unit tests have been made. These unit tests use the built-in Python testing framework ‘unittest’. This means the tests can be made without creating a custom testing framework for the game. The unittest module includes features that cover test methods, organisation, discovery, execution and reporting. These tests can be run from the command line by simply entering ‘>python -m unittest’.

There are a total of 22 tests for the system that cover the game, the player system, the Minimax player, and the Monte Carlo player. The test report can be seen in Appendix 1.
The tests for the Game cover everything from game creation to copying to move making. These tests ensure that the game will always behave correctly even when copied and will always play by the rules. There are also important tests that deliberately send bad input data to the Game object. These ensure that when bad input is sent the Game will handle it correctly, either by rejecting the input or simply crashing. When creating a Game instance with bad inputs the intended behaviour is for the Game to crash, so a Game cannot be played with impossible boundaries.

This validation is also backed up by the GUI. As the GUI only allows certain values to be used when creating Game and Player objects and these values are limited to only valid parameters, it is impossible to set up a game with bad values while using the GUI. This is also true while playing the game using the GUI. Players are only ever called on in turn and only ever get to enter one move, which means moves cannot be entered in the wrong order. This is also helped by the design decision to make the Game object decide which player's turn it is. As the game decides, neither the User or the GUI has control over who is taking a turn, and so can only send the move to the game to be rejected or accepted.

Results and Evaluation

To analyse and evaluate the game and the players that have been implemented, a series of round robin tournaments will be performed. This tournament method has been chosen as the number of contestants is low and games can be completed quickly (23).

Parameter Selection

Three of the players can be modified with different parameters. These are the Ordered player, the Minimax player, and the Monte Carlo player.

Ordered Player

The Ordered player is affected by the order in which moves are returned and selected. A potential solution for choosing the best moves in order would be to rank the moves using a heuristic, however this is more of a best-first approach and would not lead to a static order, which is not the intention behind this player.
The chosen solution is to take the list of legal moves returned from the game object and to take a move at a particular index. 6 different variations of this player were tested against a random player to compare their strengths. These variations are: first move, last move, one quarter, halfway, three quarters and one/three quarters alternating. These correspond to where in the list the move will be taken from when the player is asked for a move. The first move strategy will always take the first move in the list, the halfway strategy will always take the move in the middle of the list and the alternating strategy takes a move at the one quarter mark on one turn then the three-quarter mark on the next turn.
These strategies were motivated by the order in which moves are returned from the game, which is shown in Figure 7. Taking moves from different points in this list should roughly correspond to where they are on the board.

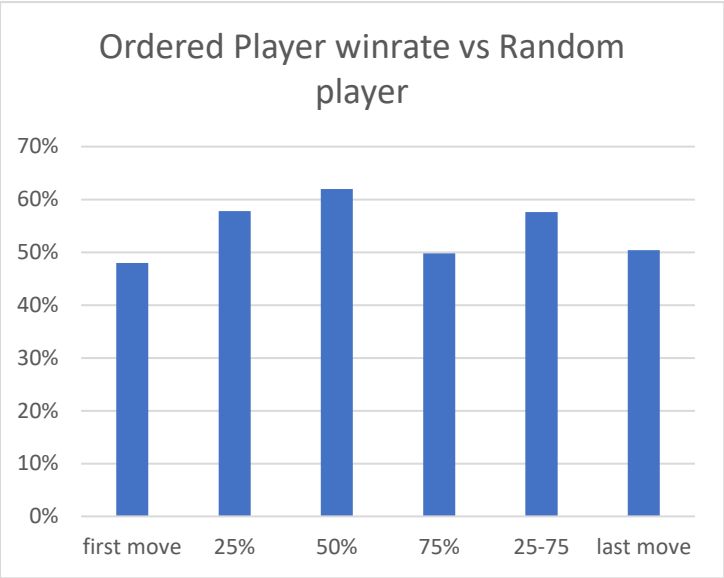

Figure 18 - Ordered player strategy comparison.

The results from these trials are shown in Figure 18. These trials were all done on a 4x4 game grid with the random player as player 1 and the ordered player as player 2. 500 games were played against each version of ordered player.
The win rates shown in the results are all close to 50%, which is expected against a random player.
The author expected the 25-75 alternating strategy to perform the best, as this would roughly correspond to picking moves close to the middle of the board, however this is not the case.
The strategy that performs the best is the halfway strategy, with a win rate of 62%. This implies that moves in the middle of the list are generally stronger. At the start of the game the halfway point of the list corresponds to the top-left vertical line. If player 1 makes moves either side of this halfway point, the halfway point of the list will then correspond to the bottom-right horizontal move. It is possible that this alternating of top-left and bottom-right moves creates a strategy that is marginally more effective than random move selection.

Minimax Player

Initially, the depth that minimax searched to was a variable parameter. However, once iterative deepening was implemented this was no longer an option. At this point, the max depth reachable by minimax became the only parameter to modify. Increasing the max depth would allow minimax to search deeper and further into the game, but only if it is still within the time limit. Consequently, the max depth value is not often reached as the time limit is reached first.

While the time limit is a parameter that can be modified to improve performance, increasing the time limit gives the Minimax player an advantage over the other players. For this reason, the time limit is kept at a static value that is the same for both the Minimax and Monte Carlo player. This makes the comparison fairer, as both players are given the same resources. The problem with increasing the time limit also shows when running tournaments. The longer each player is given to choose each move, the longer the entire tournament will last. Due to this, time limit is balanced between speed and performance.

Monte Carlo Player

The Monte Carlo player has one parameter that can be varied to modify performance. This value is the exploration coefficient **c**, which is a part of the UCB formula shown in the Monte Carlo background section. Modifying the exploration coefficient will make the MCTS algorithm explore more or less. With a high exploration coefficient, the algorithm will be biased towards nodes that have not been visited as much. With a low coefficient the algorithm will 'play it safe' and more often choose the nodes that have a higher known score.

(15) shows that the value for the exploration coefficient is theoretically optimal at $\sqrt{2}$, however the best value for a specific application should be chosen experimentally. To determine the best value

for Dots and Boxes, trials of different values were performed. Values ranging from 1.0 to 5.0 were tested to discover which values result in higher win rates.

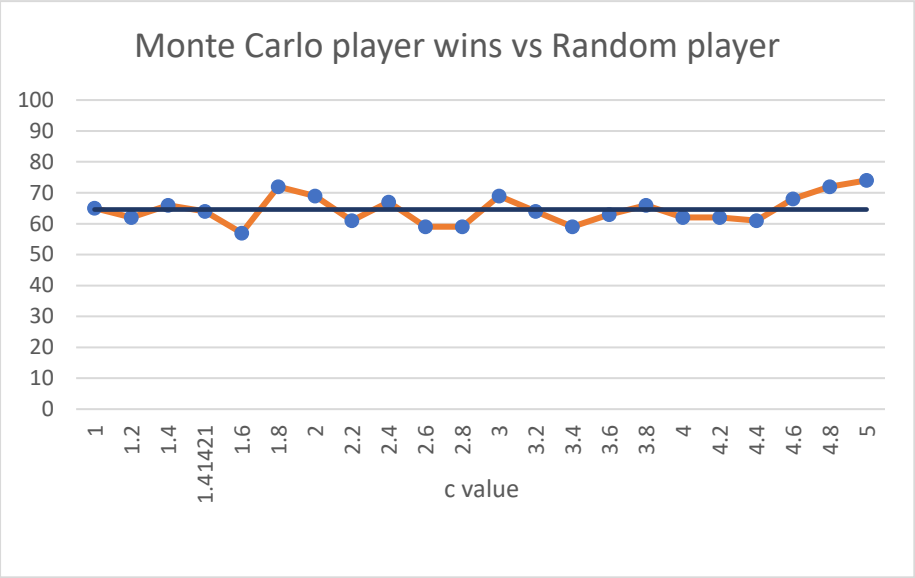

Figure 19 - Results of trials to determine optimal c value.

These trials were all performed on a 3x3 game grid. All were performed with the random player as player 1 and the Monte Carlo player as player 2. Each c value was tried in 100 games. The results of these trials, shown in Figure 19, show no obvious trend or optimal value. The horizontal line represents the average win rate of 64.59%, which is only marginally better than the ordered player's optimal win rate. As there is no obvious optimal value, the value of $\sqrt{2}$ will be used for the final tournaments.

Tournaments

3x3

The first tournament to take place will be the tournament on a 3x3 game grid. This tournament consists of each player competing against each other player in 100 games, meaning each player will play 800 games total. 200 of these games will be against copies of themselves, so each player plays 600 games against all of the other players.
In this tournament, the Minimax and Monte Carlo players each get 1 second to choose their move. This is because the small grid size leads to a smaller branching factor, so the complex players do not require a long time to reach a suitable depth.

As the 3x3 grid has 4 boxes, ties are possible. These are not included in Figure 20. The full tournament results, including ties, are in Appendix 2. The results are presented here in a table for readability.

PLAYER 1 / PLAYER 2	RANDOM	ORDERED	MINIMAX	MONTE CARLO
RANDOM	37.11%	42.00%	100.00%	83.00%
ORDERED	58.00%	100.00%	100.00%	99.00%
MINIMAX	3.00%	0.00%	100.00%	0.00%
MONTE CARLO	30.00%	5.00%	100.00%	71.00%

Figure 20 - 3x3 tournament results.

Figure 20 shows the results of each set of 100 games. The percentages correspond to the win rate of player 1. For visibility, the results in which player 1 came out on top are coloured green. Blue results

mean Player 2 won. While the results show the win rate of player 1, they do not factor in draws. For example, the match up of Monte Carlo vs Minimax shows a win rate of 0% for Monte Carlo. While it is true that the Monte Carlo player did not win a single game, the Minimax player did not win 100. Minimax only won 68 out of 100 games, with 32 draws.

As the table of results shows, the Minimax player is the strongest player. The Minimax player only lost three games in total over the course of the entire tournament; these were three games in which the random player won. This shows that the simple strategy implemented by Minimax in its evaluation function is useful when competing against other players without any game knowledge. The Monte Carlo player is shown to be the second strongest player. It came out on top of the uninformed players in every case, showing that the Monte Carlo method is also effective against uninformed strategies.

There is evidence to show that on a 3x3 grid, player 1 is in a stronger natural position than player 2. This is best shown by the matchup of Minimax vs Minimax. In this case, the player 1 Minimax instance won every game, showing that player 1 has more control over the game and can win more often than player 2. This is also true for the Monte Carlo player, though the results are not as definitive.
This is also backed up by the fact that while the Monte Carlo player did not win a single game against the Minimax player, it did draw 32 of its games when Minimax was player 2. The Monte Carlo player is clearly not as strong as the Minimax player, but when it is player 1 it can force a draw more often.

However, the matchup of random vs random does not support this conclusion. While the table in figure 20 shows that the random player that went first achieved a win rate of 37.11%, this does not mean that player 2 achieved a 62.83% win rate. The results table in appendix 2 shows the full picture; player 2 won 52 games out of 100, while player 1 won 36, with 12 draws. This shows a slight bias to player 2, which is not in line with the results from Minimax and Monte Carlo.

4x4

A game of Dots and Boxes on a 4x4 grid is different from a game on a 3x3 or 5x5 grid. This is because the 4x4 grid has 9 boxes, meaning the players cannot draw.
In the 4x4 tournament each player will play 100 games as in the other tournaments, but the time limit for the Minimax and Monte Carlo players is increased to 5 seconds. This is an attempt to see if the performance of these players can be further improved over the uninformed players.

PLAYER 1 / PLAYER 2	RANDOM	ORDERED	MINIMAX	MONTE CARLO
RANDOM	53.00%	66.00%	100.00%	86.00%
ORDERED	38.00%	0.00%	100.00%	74.00%
MINIMAX	0.00%	0.00%	6.00%	0.00%
MONTE CARLO	24.00%	26.00%	100.00%	44.00%

Figure 21 - 4x4 tournament results

Figure 21 shows the results of the 4x4 tournament. This tournament shows again that Minimax is the strongest player that has been implemented, winning 100% of its games. It also supports the conclusion that the Monte Carlo player is the second strongest, coming out on top of the random and ordered players in most of the games it played.

The results of this tournament show a slight difference in strength between the ordered and the random player. The ordered player's win rates of 66% and 62% indicate that it is marginally stronger than the random player. This implies that the ordered strategy of making top-left and bottom-right moves is at least somewhat stronger than a completely random strategy.

In contrast to the first tournament, the second tournament does not indicate that player 1 has a stronger position. The result of random vs random is 53/47, and Monte Carlo vs Monte Carlo is 44/56. Neither of these results strongly support the conclusion that either position is stronger from the start. The results of minimax vs minimax show the opposite of this; player 2 won 94 games out of 100, which would seem to support the conclusion that player 2 has a stronger position with more board control. This is in line with the real solution to the 4x4 game of Dots and Boxes presented by E. Berlekamp (5), which shows that player 2 can win every game on this grid size.

5x5

The results from the 5x5 size tournament took the longest to obtain, as there are more moves to make in the larger game. The complex players also got 5 seconds to think about each move in this tournament. The decision to give this much time to the players was made due to the larger branching factor of this board. If the players were given less time they would not be able to search to an effective depth, meaning they would be less effective on the larger board.

Similar to the 3x3 board previously, a 5x5 board has an even number of boxes. This means that it is possible to draw on this board. As before, draws are not factored into the results table below. They are included in Appendix 3 however, in the full list of results.

PLAYER 1 / PLAYER 2	RANDOM	ORDERED	MINIMAX	MONTE CARLO
RANDOM	49.00%	47.00%	100.00%	74.00%
ORDERED	46.00%	100.00%	100.00%	76.00%
MINIMAX	0.00%	0.00%	100.00%	0.00%
MONTE CARLO	18.00%	27.00%	100.00%	51.00%

Figure 22 - 5x5 tournament results

As in the previous two tournaments, Minimax comes out on top. This is another strong result for Minimax, as it won all 600 of the games it played against other players. This is also the strongest result yet for Monte Carlo, winning 303 of its 600 games against other players.
This tournament showed no significant results for the random and ordered players. With win rates of close to 50% against each other, there is no indication of one being stronger than the other. This may mean that the strategy that the ordered player uses loses its effectiveness on other board sizes. It was somewhat effective on the 4x4 board but lost any advantage it may have had on the larger 5x5 board.

There was also no evidence from the random or monte carlo player that would indicate either player position is stronger. These matchups also had win rates close to 50% for either player, showing no strong evidence to prefer either one.
The Minimax player did show a strong preference, however. The Minimax player that was player 1 won all of its matches against the player 2 Minimax player. This would seem to point to player 1 being stronger, but this may not be the case. As the minimax algorithm that has been implemented

is completely deterministic, it does not have any random elements like monte carlo, this may mean that the minimax player 1 simply wins every time with the same board, starting conditions and resources.

Performance and Ranking

After completing the tournaments on multiple board sizes, the players can now be statistically compared over all of the games.

It is clear that Minimax is the strongest algorithm in every setting, achieving win rates near 100% in every match up on every board size.

It also shows that Monte Carlo is stronger and performs better than the two uninformed players. The Monte Carlo player lost to Minimax, but when competing against the uninformed players it won more often than it lost or drew.

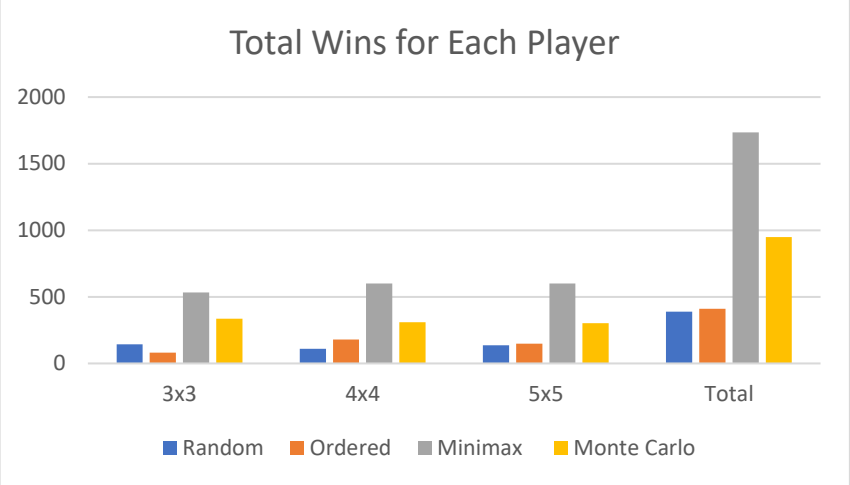

Figure 23 - Total number of wins achieved by each player.

These observations lead to an initial ranking, with Minimax being the strongest and Monte Carlo being the second strongest. This is shown in Figure 23; it shows the total number of wins achieved by each player on each board size. The bars clearly show Minimax winning far more than any other player, and show Monte Carlo taking the second most wins.

3x3 Board				
Player 1	***P1 Wins***	***Draws***	***P2 Wins***	***Player 2***
Random	4218	1551	4231	*Random*
Random	3938	1297	4765	*Ordered*
Ordered	5251	1169	3580	*Random*
4x4 Board				
Player 1	***P1 Wins***		***P2 Wins***	***Player 2***
Random	5025		4975	*Random*
Random	3665		6335	*Ordered*
Ordered	5953		4047	*Random*
5x5 Board				
Player 1	***P1 Wins***	***Draws***	***P2 Wins***	***Player 2***
Random	4752	570	4678	*Random*
Random	2917	578	6505	*Ordered*
Ordered	6558	558	2884	*Random*

Figure 24 - Table of results for large tournaments.

The two other players, random and ordered, require a closer look to rank. Statistically the ordered player won more games overall than the random player, however this is only by a very small margin and is not enough to definitively say that ordered is the better player. To clarify this, another set of trials were performed. As the random and ordered players are the uninformed, non-complex players, these trials could be completed very quickly. This means it was possible to simulate 10000 games in a short span of time. This should show with more precision which player comes out on top more often.

Figure 24 shows the results of these trials. The random and ordered players played sets of 10000 games against each other as both player 1 and player 2. These were played on all of the previously tested board sizes. The matches of ordered vs ordered were not

performed, as these matches always have the same outcome.
Figure 24 shows a clearer picture of which player is stronger. On all three boards, the ordered player wins more games than the random player, regardless of whether it is player 1 or 2. The large number of matches played means that these results likely show real trends and not just anomalies.

This shows that the ordered player's strategy is stronger than a strategy of picking moves at random.

With these new results, it is now possible to make a ranking of the four players. This ranking is as follows.

1. Minimax
2. Monte Carlo Tree Search
3. Ordered
4. Random

Determining if player 1 or player 2 has a stronger initial position is still difficult even with all of the results gathered. Figure 25 shows all of the wins for player 1 and player 2 from the three initial tournaments totalled up. As can be seen, on the 3x3 board player 1 won more often, on the 4x4 board player 2 took the lead and on the 5x5 board player 1 came out on top. The largest difference is for the 3x3 board, with player 1 winning 62.85% of games. The other boards have smaller differences and so there is even less confidence in those results.

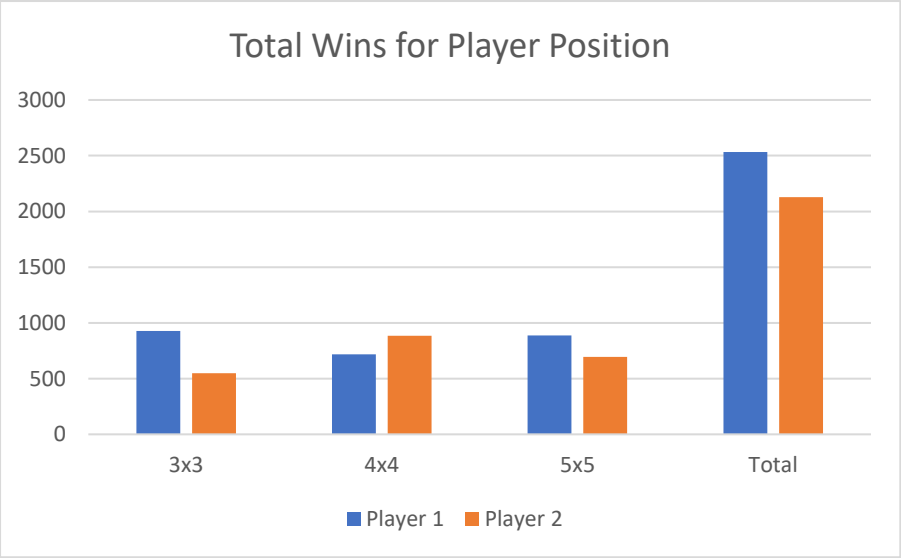

Figure 25 - player 1 vs player 2 overall results

The 4x4 board should show a clear advantage for player 2, as there is a known strategy for player 2 to win every time on this board. The fact that this is not obvious in the statistics shows that the methods used to determine this are not good enough to show the correct result. It may be that the players do not have enough of a well-defined strategy to exploit the advantage that certain positions come with.

Future Work

The implementation of the game and the player system has been designed with future work in mind. The game has been designed to be standalone and accessible; the player system has been designed so that the specific implementation of a player does not matter to the game. As creating new players and incorporating them to the game is designed to be as simple as possible, it would be easy to use this implementation to test many other players and strategies.
One possible area of future work is analysing different board sizes. As the board can be made any

size and this does not matter to the player, this implementation can be used to analyse many other shapes and sizes of board.

Improvements to Minimax

There are many improvements that could be made to the minimax algorithm. Firstly, the evaluation function can be improved to incorporate more game knowledge about specific situations. Currently, the evaluation function only factors in game score and number of complete edges for each box. The evaluation function could be updated to recognise chains of boxes and be taught to make beneficial sacrificing moves. These two parts of the game are recognised strategies that humans use to play and win. Most games will end up with one or more chains of boxes that can all be captured at once; having control of the chains usually means that the controlling player can win or force a tie (3). In many situations keeping control of the chains requires sacrificing boxes, something that the minimax player currently does not do intentionally.

Improvements to MCTS

The current implementation of MCTS does not use any game knowledge, nor does it use any of the advanced features that MCTS can incorporate. One of the first steps to improving MCTS would be to parallelise the algorithm. (24) MCTS has the advantage of being very easy to run in parallel, with multiple different methods of parallelisation. Doing this, especially when run on a more powerful computer with multiple processor cores, would speed up the execution of MCTS dramatically, thus improving its performance.

MCTS can also be improved with game knowledge. It can incorporate an evaluation function, similar to Minimax, that can weight nodes based on their static evaluation score. If this is an effective evaluation function, it will further improve the performance of MCTS.

Yet another potential improvement is to the rollout step of the algorithm. Currently, a rollout consists of playing purely random moves from a game state until the end. This can be improved by implementing some other strategy as part of the move selection. For instance, the selection strategy used by the current ordered player could improve performance, as this has been proven to be more effective than random move selection.

There is also, of course, the possibility of incorporating machine learning and neural networks. These methods have been proven to be very effective and have produced some of the best computer players of various games in the world to date. (9)

Conclusions

The aims of this project that were outlined in the important conclusions section included creating an implementation of the game, creating working computer players and analysing game playing strategies. Most of these objectives have been successfully achieved, with the exception of determining which player position is stronger.

A working and reliable implementation of the game Dots and Boxes has been created. The time spent designing and implementing the game in a modular way paid off; the game is reliable and the player system is very loosely coupled to the game system. This can be seen in the implementation section of the report, and in Figure 6, the UML class diagram. The game is proven to be reliable with the suite of unit tests created and described in the testing section.

A working GUI has also been created to make playing the game enjoyable and easy. The GUI has been designed so that users cannot enter data that would cause the game to crash. It has also been designed so that humans can play against each other, against the computer or can watch two computer players play against each other.

The computer players created for the game are successful in their implementation. They play reliably and are consistent in their strategies. Data was collected by simulating many games with each of the players, on multiple different board sizes. This data was analysed and a ranking was produced, ordering the players from strongest to weakest. This ranking is Minimax, Monte Carlo Tree Search, Ordered then Random. Multiple potential improvements to the players that could be made have been identified after analysis of the data obtained and further research. These are described in the future work section of the report.

The only aim that has not been achieved is determining if either player 1 or player 2 is in a stronger position on any board size. This was attempted and data was gathered, but the data was inconclusive and no definitive answer was found. This is likely because the computer players that have been implemented are not strong enough to exploit the strength inherent in either position. To determine this in future, stronger players would need to be produced and more trials would need to be performed.

Reflection

In reflection, I believe my project has been a success. Most of the aims that were laid out were achieved, and those that were not have been understood.
The time spend designing and implementing the game and the player system paid off as it allowed me to gather all the data I required easily. With a reliable game and player system I could set tournaments running in the background for multiple days at a time without fear of the program crashing.
The modular design of the system also helped to simplify the development process. As the program could be very easily decomposed into discrete sections, each part could be developed in isolation.

I have realised while undertaking this project the importance of flexibility in development. Initially, I planned equal amounts of time to develop the game, the GUI, the player system and then the players. When I came to develop the program I got through the first parts very quickly. I then made the decision to move on to other parts of the program early once I had finished these parts, rather than wasting the time I had by sticking rigorously to my schedule. Because of this, I had far more time to develop the later parts of the program and I could spend longer bug fixing, tweaking and fine tuning the advanced players. This extra time spent on the advanced computer players meant that they were better and more developed than I could have hoped for with the time previously allocated.

If I were to do this project again, I would change the way in which I collected my results. I did not have a very organised method for collecting results, simply collecting them when I could. I believe it would be beneficial to have a more rigorous testing schedule, determining what I need before collecting data. If I had done this, I would only need to perform each test/trial/tournament once. As it was, I collected tournament data multiple times due to changes that were made to the players after performing the tournaments. If I had planned out my testing, I would have done all of the parameter selection testing first and then completed all of the tournaments after I had already determined the optimal configurations of the AI players.

Table of Figures

Table of Abbreviations & Synonyms

Abbreviation	*Meaning & Synonyms*	*Description*
MCTS	Monte Carlo Tree Search	The Monte Carlo Tree Search method is used by a player agent to explore the decision tree.
UCB	Upper Confidence Bound	Formula used by MCTS to give nodes weighting based on their utility and an exploration parameter.
OO, OOP	Object-Oriented, Object-Oriented Programming	Programming paradigm focused on the interactions between defined objects.
GUI	Graphical User Interface	Visual interface used to interact with a digital computer system.
UML	Unified Modelling Language	Modelling language designed and used to describe systems.

Appendices

Appendix 1

Unit test result report.

```
C:\Users\Matt\Documents\University\YEAR3\PROJECT\code>python -m unittest -v -b
test_bad_moves (DotsAndBoxes.tests.test.TestGameMethods)
Ensure program behaves correctly when bad inputs are given. ... ok
test_box_checking (DotsAndBoxes.tests.test.TestGameMethods)
Test that the correct boxes are checked when move is made. ... ok
test_copy (DotsAndBoxes.tests.test.TestGameMethods)
Test copying a Game and make sure the copy behaves as intended. ... ok
test_create_game (DotsAndBoxes.tests.test.TestGameMethods)
Test creating different sizes of game. ... ok
test_create_game_bad_input (DotsAndBoxes.tests.test.TestGameMethods)
Test creating the game with various bad inputs. ... ok
test_game_equality (DotsAndBoxes.tests.test.TestGameMethods)
Test that game objects' eqality method works correctly. ... ok
test_game_finished (DotsAndBoxes.tests.test.TestGameMethods)
Test that a finished game is reliably finished. ... ok
test_game_not_finished (DotsAndBoxes.tests.test.TestGameMethods)
Test that a game in progress does not display as finished. ... ok
test_game_rules (DotsAndBoxes.tests.test.TestGameMethods)
Test that the rules of the game are followed. ... ok
test_legal_move_generation_middle (DotsAndBoxes.tests.test.TestGameMethods)
Test mid game legal move generation. ... ok
test_legal_move_generation_start (DotsAndBoxes.tests.test.TestGameMethods)
Test new game legal move generation. ... ok
test_move_making (DotsAndBoxes.tests.test.TestGameMethods)
Test that making moves claims lines as expected. ... ok
test_play_game (DotsAndBoxes.tests.test.TestGameMethods)
Test creating and playing a game. ... ok
test_saving_scores (DotsAndBoxes.tests.test.TestGameMethods)
Test that scores sent to save files are correctly saved. ... ok
test_scores_correct (DotsAndBoxes.tests.test.TestGameMethods)
Test that scores are awarded as expected when moves are made. ... ok
test_winner (DotsAndBoxes.tests.test.TestGameMethods)
Ensure the game declares the correct winner. ... ok
test_minimax_evaluation (DotsAndBoxes.tests.test.TestMinimaxMethods)
Test that the Minimax evaluation function is consistent for both players. ... ok
test_minimax_selection (DotsAndBoxes.tests.test.TestMinimaxMethods)
Test the correct operation of the Minimax algorithm. ... ok
test_monte_carlo_node (DotsAndBoxes.tests.test.TestMonteCarloMethods)
Test the functionality of the Monte Carlo Node class. ... ok
test_monte_carlo_tree (DotsAndBoxes.tests.test.TestMonteCarloMethods)
Test the functionality of the Monte Carlo Tree class. ... ok
test_game_with_players (DotsAndBoxes.tests.test.TestPlayerMethods)
Test that playing a game with two players works as intended. ... ok
test_playerfactory (DotsAndBoxes.tests.test.TestPlayerMethods)
Test that players returned from the player factory are correct. ... ok

----------------------------------------------------------------------
Ran 22 tests in 0.293s

OK
```

Appendix 2

3x3 tournament full results.

Player 1	P1 Wins	Draws	P2 Wins	Player 2
Random	36	12	52	Random
Random	58	9	33	Ordered
Random	3	31	66	Minimax
Random	30	9	61	Monte Carlo
Ordered	42	12	46	Random
Ordered	100	0	0	Ordered
Ordered	0	0	100	Minimax
Ordered	5	2	93	Monte Carlo
Minimax	100	0	0	Random
Minimax	100	0	0	Ordered
Minimax	100	0	0	Minimax
Minimax	100	0	0	Monte Carlo
Monte Carlo	83	11	6	Random
Monte Carlo	99	0	1	Ordered
Monte Carlo	0	32	68	Minimax
Monte Carlo	71	7	22	Monte Carlo

Appendix 3

5x5 tournament results.

Player 1	P1 Wins	Draws	P2 Wins	Player 2
Random	49	4	47	Random
Random	46	3	51	Ordered
Random	0	0	100	Minimax
Random	18	2	80	Monte Carlo
Ordered	47	3	50	Random
Ordered	100	0	0	Ordered
Ordered	0	0	100	Minimax
Ordered	27	0	73	Monte Carlo
Minimax	100	0	0	Random
Minimax	100	0	0	Ordered
Minimax	100	0	0	Minimax
Minimax	100	0	0	Monte Carlo
Monte Carlo	74	4	22	Random
Monte Carlo	76	0	24	Ordered
Monte Carlo	0	0	100	Minimax
Monte Carlo	51	0	49	Monte Carlo

References

1. *Solving Dots and Boxes.* **Barker, Joseph K, and Korf, Richard E.** 2012, Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, Vol. 26, pp. 420-426.

2. *La Pipopipette: nouveau jeu de combinaisons.* **Lucas, Édouard.** s.l. : Paris: Gauthier-Villars et fils, 1895, L'arithmétique amusante, pp. 204-209.

3. *Chapter 16: Dots and Boxes.* **Berlekamp, Elwyn R, Conway, John H and Guy, Richard K.** 2, s.l. : Academic Press, 1982, Winning ways for your Mathematical plays, Volume 3, Vol. 3, pp. 507-550.

4. *Improving Monte Carlo Tree Search With Artificial Neural Networks without Heuristics.* **Cotarelo, Alba, et al.** 5, 2021, Applied Sciences, Vol. 11, p. 2056.

5. **Haran, Brady and Berlekamp, Elwyn.** How to always win at Dots and Boxes - Numberphile. *YouTube.* [Online] 12 January 2015. https://www.youtube.com/watch?v=KboGyIilP6k.

6. **Wilson, David.** Dots-And-Boxes Analysis Results. [Online] [Cited: 01 04 2021.] https://wilson.engr.wisc.edu/boxes/results.shtml.

7. *Zur Theorie der Gesellschaftsspiele.* **Von Neumann, J.** 1928, Mathematische Annalen, Vol. 100, pp. 295-320.

8. *Mastering the game of Go with deep neural networks and tree search.* **Silver, David, et al.** 2016, Nature, Vol. 529, pp. 484-489.

9. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.* **Silver, David, et al.** s.l. : Cornell University, 2017.

10. **Champandard, Alex J.** AiGameDev. [Online] 12 August 2014. [Cited: 14 April 2021.] https://web.archive.org/web/20170313041719/http://aigamedev.com/open/coverage/mcts-rome-ii/.

11. *Monte Carlo Methods.* **Johansen, A.M.** 2012, International Encyclopedia of Education (Third Edition), pp. 296-303.

12. **Abramson, Bruce.** *The Expected-Outcome Model of Two-Player Games.* Columbia : Department of Computer Science, Columbia University, 1987.

13. *An Adaptive Sampling Algorithm for Solving Markov Decision Processes.* **Chang, Hyeong Soo, et al.** 2005, Operations Research, Vol. 53, pp. 126-139.

14. *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search.* **Coulom, Rémi.** Turin, Italy : Springer, 2006, Computers and Games, 5th International Conference, pp. 29-31.

15. *Finite-time Analysis of the Multiarmed Bandit Problem.* **Auer, Peter, Cesa-Bianchi, Nicolò and Fischer, Paul.** 2002, Machine Learning, Vol. 47, pp. 235-359.

16. **Choudhary, Ankit.** Analytics Vidhya. [Online] 2019. [Cited: 20 03 2021.] https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/.

17. **Riverbank Computing Limited.** PyQt5==5.15.2. Dorchester : Riverbank Computing Limited, 2020.

18. **TutorialsPoint.** Tutorials Point. [Online] [Cited: 15 March 2021.] https://www.tutorialspoint.com/design_pattern/factory_pattern.htm.

19. **Shockaert, Steven.** CM3112 Artificial Intelligence. Cardiff : Cardiff University, 2020.

20. **Geeks for Geeks.** GeeksforGeeks. [Online] 2019. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/.

21. —. GeeksforGeeks. [Online] 2019. https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/.

22. **Baeldung.** Baeldung. [Online] 2020. https://www.baeldung.com/java-monte-carlo-tree-search.

23. **Byl, John.** *Organizing Successful Tournaments-4th Edition.* Ontario : Human Kinetics, 2014. 9781450460279.

24. *Parallel Monte-Carlo Tree Search.* **Guillaume, M.J-B., Chaslot, Mark H.M. and Winands, Jaap van den Herik.** Beijing : Springer, 2008, Computers and Games, 6th International Conference, pp. 60-71. 978-3-540-87607-6.

25. **Tiger66.** Wikimedia. [Online] 21 11 2011. [Cited: 01 04 2021.] https://commons.wikimedia.org/wiki/File:Dots-and-boxes.svg.

26. *A Comparative Study of Game Tree Searching Methods.* **Elnaggar, et al.** 2014, International Journal of Advanced Computer Science and Applications., Vol. 5, pp. 68-77.

27. **Rmoss92.** Wikimedia. [Online] 3 April 2020. [Cited: 20 04 2021.] https://en.wikipedia.org/wiki/File:MCTS-steps.svg.