Cardiff University

Final year project

Gomoku AI Player

Daniel Ford - C1224795

supervised by

Yukun Lai

May 6, 2016

1 Abstract

This report will investigate different AI (Artificial Intelligence) approaches towards the game Gomoku. It also shows the process taken to get to the end system. This begins at the planning and research stage then proceeds to the design stages of each part of the system such as the UI (User Interface) or AI algorithms, through to summaries of the implementations.

The AI algorithms explored in this report include a Heuristic Function, MiniMax, Iterative Deepening and Monte Carlo Tree Search. The report will show a comparison of how each approach played against each other and make conclusions as to why the results are as they are.

The report will end with showing the work that could potentially be planned for the future and a reflection of the work carried out.

Contents

1 Abstract **1**

2 Introduction **4**

3 Background **5**
3.1 The game of Gomoku 5
3.2 Gomoku and Artificial Intelligence 6
3.2.1 Approaches 7

4 Creating the Game **8**
4.1 Project Management 8
4.2 Research 8
4.2.1 Gomocup 8
4.3 Design 9
4.3.1 The Board 10
4.3.2 User Interface 12
4.3.3 The Player 15

5 Creating the AI Player **16**
5.1 MiniMax 16
5.2 Heuristic Evaluation 18
5.2.1 Implementing the Heuristic 20
5.3 Alpha Beta Pruning 22
5.4 Depth limited MiniMax implementation in Java 24
5.4.1 Deciding a move 25
5.4.2 Minimising and maximising 27
5.5 Iterative Deepening 29
5.5.1 Limiting the depth 29
5.5.2 Incrementing the depth over time 31
5.5.3 Iterative Deepening implementation 33
5.6 Monte Carlo Tree Search 36
5.6.1 Multi-Armed Bandit Problem 36
5.6.2 UCB-1 37
5.6.3 Selection 38
5.6.4 Expansion 39
5.6.5 Simulation 39

5.6.6 Back Propagation 40
5.6.7 Full MCTS Pseudocode 40
5.6.8 MCTS Implementation 41

6 Experiments and Discussions **45**
6.1 Experiments 45
6.2 Results and Conclusions 46
6.2.1 Set Up 46
6.2.2 Heuristic VS other algorithms 46
6.2.3 Depth Limited MiniMax VS Iterative Deepening 48
6.2.4 Iterative Deepening VS Monte Carlo Tree Search . . . 49

7 Future Work **51**
7.1 Improving the Heuristic 51
7.2 Improving MiniMax and Iterative Deepening 51
7.3 More algorithms 52
7.3.1 Threat Space Search 52
7.3.2 Reinforcement Learning 52
7.4 Game Improvements 53
7.5 Testing 53

8 Reflection **54**

Appendix A Java Code Listings **58**
A.1 MiniMax - minimising 58
A.2 MiniMax - maximising 59
A.3 MCTS - selection 60
A.4 MCTS - expansion 61
A.5 MCTS - random simulation 61
A.6 MCTS - back propagate 62

2 Introduction

The project that I have chosen to do for this assignment is creating an AI (*Artificially Intelligent*) player for the game Gomoku. The main aim in this project will be to implement multiple AI approaches and give a comparison of how they play through gathering statistics by playing them against each other with different configurations.

Another important aim of the project that I have been given is that a UI (*User Interface*) should be implemented so that a user can easily interact and play with the AI player, or another human, should they so choose. For this project, I intend to write all of the code for the UI layer, 'data layer' and AI players from scratch.

3 Background

3.1 The game of Gomoku

Gomoku is a board game which originates from Japan. It is a game played on a Go board typically of size 15 x 15 or 19 x 19. In the game, players will take turns placing counters until a player has managed to connect 5 or more counters in a row. The player that goes first will either be the player who has just won or the player who has the black counters [12].

Unlike commonly played board games such as Chess, Gomoku is played on a Go board. This means that you place the counters on the vertices rather than within the 'squares' on the board. This means that as there are 18 x 18 squares, there are in fact 19 x 19 playable places on the board. In the game of Gomoku there are a large variety of different methods for playing the game in attempt to make the rules 'fair'. Normal Gomoku where you simply have to connect 5 with no restrictions is known as freestyle Gomoku. An example of game rules which have been made in an attempt to make the game fair for both players are the Renju rules.

An example of Renju rules is that White is not restricted in anyway. By this I mean that white can win by creating an overline i.e. more than 5 in a row. However, black is not allowed to make an overline. The black player is also not allowed combinations of moves such as a three-three or a four-four [2]. However, even with this rule it has been found that black can still have the upper-hand.

Another rule introduced by Renju is known as the "Choice Rule". This rule makes it so that black must play their first move in the center whilst white plays their stone at one of the eight squares connected to black's stone. Black's second move is unrestricted. White then has the choice of continuing the play or swapping colours with the black player [2].

Initially I would like to implement freestyle Gomoku rather than using Renju rules. I would like to implement freestyle Gomoku because of the simplicity of the game and so that I can spend the limited time of this project on the AI approaches rather than creating the actual game. If I have time I would also like to implement Renju rules so that there is a variation of game types that the user can choose from.

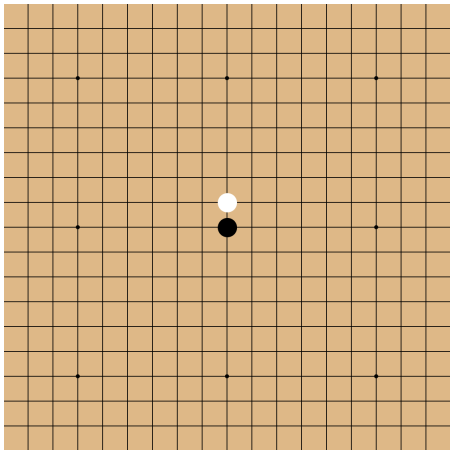

Figure 1: Counters placed on a 19 x 19 Go Board

3.2 Gomoku and Artificial Intelligence

Gomoku is often used as a starting point for creating AI players for the game of Go due to being simpler to program and providing an easier way to test UCT (**U**pper **C**onfidence Bound 1 applied to **t**rees) in e.g. the Monte Carlo algorithm [5].

When discussing solved games, freestyle Gomoku comes under the category of **solved** rather than *partially solved* or *unsolved*. A solved game is a game whose outcome can be predicted from any position given that both of the players play perfectly [14].

On a board size of 15 x 15 it has been proven that, in perfect play, black should always win. I will be implementing freestyle Gomoku which follows along the theory that, in perfect play, first player should either win or the game should result in a draw. However, this case has only been proven in a board of 15 x 15 but it is thought that it is most likely the case in boards of larger sizes [3]. Whether or not an AI player wins when they are first player will entirely depend on the strength of the algorithm used and many other variants. These variants may be how far the tree is explored or if a heuristic evaluation function is used, how good that heuristic function is.

However, freestyle Gomoku is not the only version that has been declared

solved. Renju rules are also said to have been solved in 2001 [15].

3.2.1 Approaches

Due to Gomoku being a common starting point to create AI approaches for Go [5], a number of approaches to Gomoku already exist.

Most approaches are based on a simple **MiniMax** implementation or similar algorithms such as the NegaMax. These approaches can then be further improved by developing a heuristic and other performance enhancements.

Other commonly used approaches include the **Monte Carlo Tree Search** algorithm. However, for a game such as Gomoku or Go, due to the large search space involved, light random playouts do not always play well and as such the algorithm can be improved with a strong heuristic evaluation to help with e.g. move selection in the simulation phase [5].

The approach which has been used to solve freestyle Gomoku is threat space search [2]. This approach was developed to model the way that a human Gomoku player finds winning threat sequences. Using this algorithm, a strong player was created that even won Gold in the Computer Olympiads, winning all games using black and half of the games using white.

For this project, the AI approaches that I would like to implement first are the MiniMax algorithm, Iterative Deepening, a Heuristic evaluation function and Monte Carlo Tree Search. As well as these algorithms, I would like to implement a number of improvements on top of each implementation.

If I have time, I would like to also research and implement other possible implementations which I could then also use for comparison against the other approaches mentioned.

4 Creating the Game

4.1 Project Management

To keep the progress of this project at a good rate of development, I will use various methods to manage my project.

For version control, I will be using a private Github repository. Although there are alternatives such as Gitlab or Bitbucket to github, the open source reach of Github far exceeds both of its competitors so if this project were then made open source, I believe this would be the best place for it.

To keep track of enhancements, issues and features that I would like to implement, I will be using the Github issue tracker. This allows me to create issues as I need them as well as referencing them in my commits. I believe this is useful as it creates a clear link about what was implemented when, but also provides further detail behind the thought process of each feature.

In terms of software development methodologies, I will be using the Agile methodology. Specifically from Agile, I intend to use agile practices such as:

- Agile modelling
- Test-driven development
- Refactoring
- Continuous integration

I believe these practices are important as they allow me to focus on creating a stable, well documented and designed code base whilst also allowing for me to go back and make changes, such as refactoring, as is necessary.

4.2 Research

4.2.1 Gomocup

In the interest of seeing what a high level AI player for Gomoku looked like, I found a tournament called the Gomocup. From examining the rules of the

website [6], I found that Gomocup AI players are usually:

- Given 30 seconds to decide a move
- Not allowed to have a starting size larger then 2MB
- Not allowed to exceed the size of 20MB at any point during the tournament

In the Gomocup, a standard interface is provided which your program, referred to as the 'brain', must interact with. The Gomocup allows the AI players to be written in a range of languages, including Java, so in the future it may be interesting to see how the AI players created in this project would compete. However, part of this project is creating the actual board interface so compatibility with other programs will not be the primary focus. Although, it may be simple enough to adapt the code for the AI players to interact with another interface in the future.

4.3 Design

In terms of game design, I would like the game to be designed in such a way that I can change the dimensions of the board, and everything will still work, changing dynamically. I intend to make all of the classes extremely object orientated and re-usable.

As a simple, initial design summary of the entire program, in the form of a UML class diagram, I think that the program could look something like this:

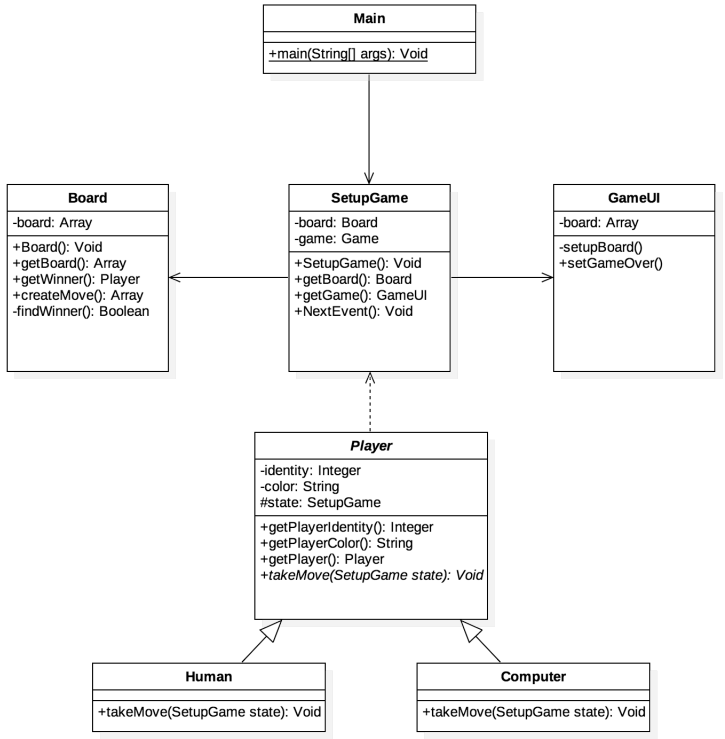

Figure 2: Initial UML class diagram representation of the project

4.3.1 The Board

Visualising the board's interface as rows and columns, one can see that the board would be seen as a multi dimensional array where placing a counter in the top left corner would be seen to be placing a counter at position **0, 0**.

Going forward in the design, I would like to create two separate representations of the board. A data representation and a user interface representation. These representations will be implemented separately, yet interact with each other. I believe that this is important as it follows the principle of *Separation of Concerns* [11].

By having this separation, it allows me to create a 'data layer' for the board

which will be the single model of truth for both users. By this I mean that both players will actually be interacting with the board at a data level and any changes will be then propagated to the presentation layer after some forms of validation. This helps increase performance by not doing unnecessary operations on the presentation layer and it also means that e.g. a 'Computer' player and a 'Human' player will be interacting with a board that both can easily understand. It also means that I could visualise the board without a user interface should it be necessary in a case such as debugging e.g. by looping through the multi dimensional array in the 'data layer'.

In the case of a human player, I intend to make it so that any place they click on the board will result in a coordinate being produced which will reference a space in the previously mentioned multi dimensional array.

As an initial design, I would like the multi dimensional array for the board to be one that can contain more than just 1 and 0. Instead I would like it to contain as much information about the Player as possible. This would allow various configurations such as, if the user wanted to, changing the counter colour or the player's name.

Listing 1: An initial board design

```
public class Board {

    private Player[][] board;
    private List<Player> players;
    private int dimensions;
    private int winningScore;
    private int maxCounters;

   public Board(int dimensions, int winningScore) {
      this.dimensions = dimensions;
      this.winningScore = winningScore;
      this.maxCounters = dimensions * dimensions;

       // Creates a multi dimensional array of specified size
       board = new Player[dimensions][dimensions];
       players = new ArrayList<>();

    }
}
```

4.3.2 User Interface

In terms of the user interface, it will need to be one that shows clear intentions and also be easy / self-explanatory for the user to use and navigate. In terms of code design, I think that the User Interface could be as simple as the following class diagram.

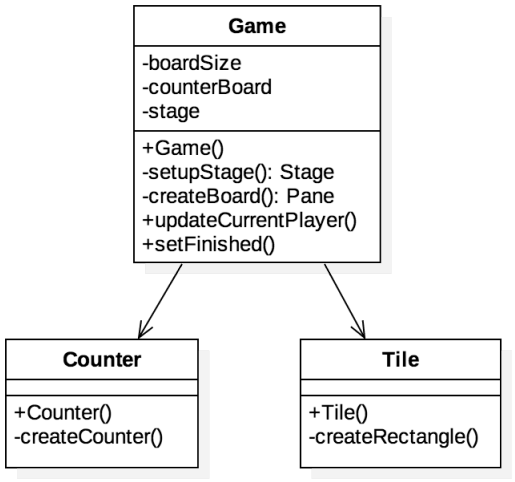

Figure 3: An initial UML design of the board

The Game class would be created in the main 'setup' class, this setup class setting up both the data layer and this UI class.

When this class is called it will create the UI representation of the board, creating tiles using the Tile class and the counters using the Counter class. I have created these two elements as two separate classes as I believe it creates a clear distinction of layers and when the user interacts with the board, it will clearly indicate which class the users are interacting with.

In terms of the actual UI, I would like the user flow to look like the following simplified flow diagram:

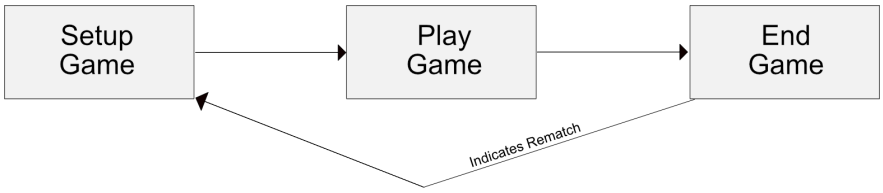

Figure 4: A simple diagram of a user's flow through the system

Figure 4 shows that I would like the first screen to be a set up screen. This would allow the user to configure what kind of players are playing the game, for example, is it two human players or a human vs an AI using the MiniMax algorithm? I think that the configuration could also go beyond this allowing the user to configure the algorithms. The options may also allow the user to change the board size and how many counters in a row there must be to achieve a win.

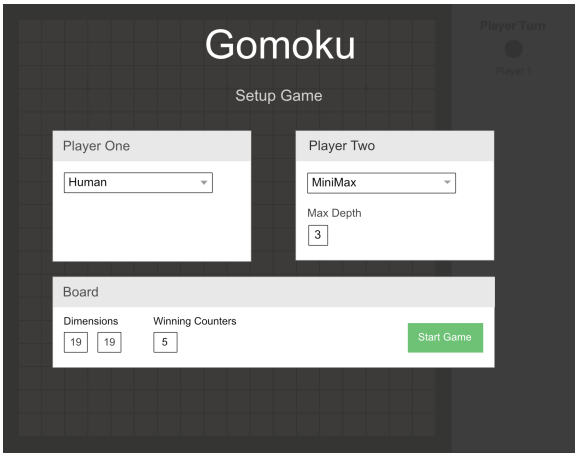

Figure 5: An initial concept design of the game setup

While the user is playing the game, they will be shown who the current player is. I believe that it might also be nice to implement features such as showing

how an AI player valued moves, or a summary of how it made it's decisions. Perhaps another useful feature might be to have an 'undo move' button and also perhaps a 'concede' button.

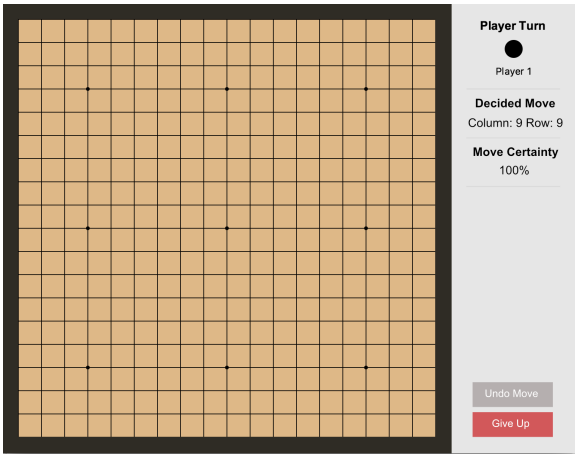

Figure 6: An initial concept design of the gameplay

After the game is finished, the game will tell the user who has won and then present them with the option of a re-match. If they confirm, then the game will begin once again and the winner will play first.

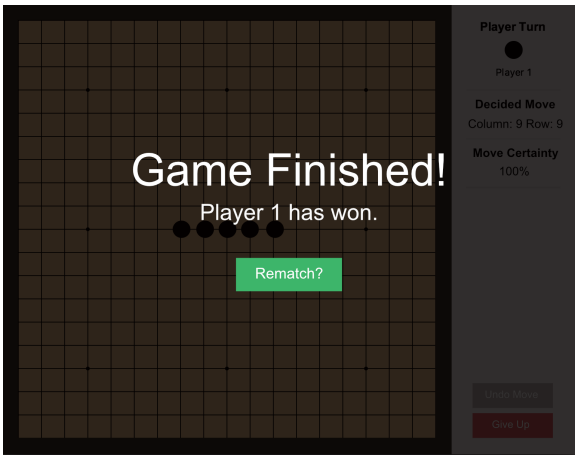

Figure 7: An initial concept design of the end game state

4.3.3 The Player

The player class will be needed to be created in an abstract way so that it can be extended upon by both a Computer style class and a Human styled class. It makes sense to have a super extendible class as it will provide basic functionality which would other wise be repeated whilst also providing a structure for the sub classes to follow.

As an initial approach, as part of my design, I believe all that would need to be implemented for this abstract class would be a 'unique identifier', a colour for that player, the game state to interact with and an abstract method for the sub classes to use to actually take a move.

Listing 2: A basic abstract Player design

```
public abstract class Player {

   private int identity;
   private Paint color;

   public Player(int identity, String color) {
      this.identity = identity;
      this.color = Paint.valueOf( color );
   }

   public abstract void takeMove( SetupGame state );

}
```

5 Creating the AI Player

Now I will discuss the AI players that I have implemented. I will do this by firstly discussing the algorithm's approach and the specific parts of it which make the algorithm different to other approaches. I will then continue explaining the algorithm by providing pseudocode that I have created.

I think it is important to provide pseudocode and not just code snippets alone as I believe that this makes the report easier to read due to being more 'programming language agnostic'. By this I mean that I would like to provide code in a way that is easily understood by readers coming from different programming language backgrounds.

As well as this psuedocode, I will explain how I have translated it into the programming langauge I have used for this project i.e. Java, so that a reader could see how the pseudocode translates into 'real code'.

5.1 MiniMax

The MiniMax algorithm was designed to address two players playing against each other in a zero-sum based game. A zero-sum game being a game where each player's gain of utility is balanced by the other player's loss of utility. In a zero-sum game, if you were to subtract one player's utility by the others, the result should sum to zero.

MiniMax works by building a tree of possible outcomes where the MiniMax player will attempt to maximise it's own score, whilst the other player will try to minimise the MiniMax player's score by maximising it's own score. This can be formally visualised as:

$$MINIMAX(s) = \begin{cases} UTILITY(s) & \text{if } TERMINAL\text{-}TEST(s) \\ \max_{a \in Actions(s)} MINIMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MAX \\ \min_{a \in Actions(s)} MINIMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MIN \end{cases}$$

Figure 8: [9]

Programmatically, we could view this algorithm as using recursion. For each move in a *List*, we would recursively see the outcome of this move based on seeing the minimum and maximum possible score from the range of child moves.

Algorithm 1 Basic MiniMax

```
1:  function chooseMove
2:      for move in board.getLegalMoves() do
3:          board.createMove( move )
4:          current = getMinValue( board )
5:          if  current.getScore() > bestScoreSoFar  then
6:              bestScoreSoFar = current.getScore()
7:              bestMove = current.getMove()
8:          board.resetMove( move )
9:      return bestMove
10:
11: function getMinValue( board )
12:     if  board.isTerminal()  then return evaluateState( board )
13:     else
14:         score = ∞
15:         for move in board.getLegalMoves() do
16:             score = getMaxValue( board.createMove(move) )
17:             result = Min( score )
18:         return result
19:
20: function getMaxValue( board)
21:     if  board.isTerminal()  then return evaluateState( board )
22:     else
23:         result = −∞
24:         for move in board.getLegalMoves() do
25:             score = getMinValue( board.createMove(move) )
26:             result = Max( score )
27:         return result
```

However, in reality, this algorithm alone would be infeasible in a game with a large search space such as Gomoku. This would be because, every iteration of the tree would involve looking at a very large search space on a 19 x 19

board, starting at a size of 19! × 19! (*where ! means factorial*). This would lead to the algorithm taking a large amount of time to make even a simple decision.

To improve this algorithm, steps would need to be taken to make it so that the algorithm would not have to search the entire search space of the board. This could be done by implementing: a Heuristic Evaluation function, Iterative Deepening or pruning techniques such as Alpha Beta pruning.

5.2 Heuristic Evaluation

In games with large search spaces, it is useful to create a heuristic function which will evaluate which moves might be relevant to the Computer Player at the current state of the game, also optionally providing 'scores' for those moves. It might be useful for a heuristic to also provide scores for a move as this could help in situations where, even with the heuristic function, the end of the game was not met. This will help the Computer Player to weigh up the value of moves at the point it did manage to reach in the search tree.

For the purpose of this project, the evaluation function I will make will be a simple one, however, I can think of many improvements which would be useful for taking the project forward and making the existing algorithm even stronger.

Figure 9 shows that black player has placed 2 counters, and the white 'computer' player has only played one, so it is the computer's turn. In the terms of the heuristic, I have designed it such that it will take the current most valuable move to itself i.e. it should block the black player.

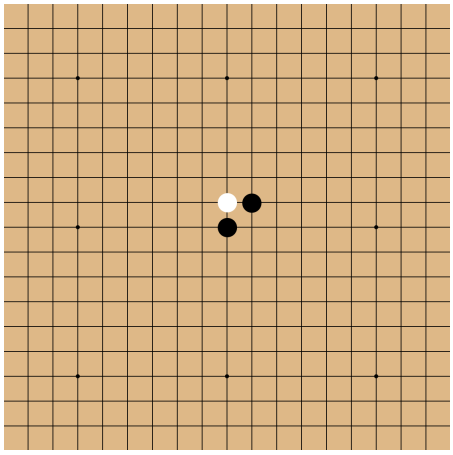

Figure 9: Black has played 2 counters, Computer is white.

This heuristic will first gather moves by looking at the search space around all existing counters. A move is then evaluated and considered by the heuristic counting how many counters there are in a chain either side of the proposed move. The heuristic will then assign a negative or positive value depending on whether or not the move belongs to the current player.

For example, in the case of the Figure 9, the heuristic would assign a score of -3 to column 11 and row 7 (or column 8 row 10) as, if the black opponent plays here, they will have a score which has a greater impact on the white opponent's own score.

This heuristic is similar to the Threat utility function where moves are chosen by the computer player based on a greedy approach [2]. By this I mean, the move that will be picked will be the most valuable for it's current state i.e. it will block the opponent if it is more valuable than furthering it's own 'chain' of counters.

The weakness of this approach is that the heuristic doesn't look ahead in the game. By this I mean that a move might be valuable for it's current state. However, it may be that upon going further into the game tree, another move may have been better for it. An example of this is that an opponent might play moves with space between them, in an attempt to eventually create a situation where they will have winning combination either way.

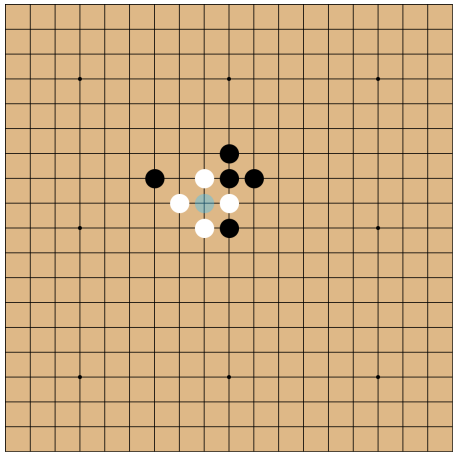

Figure 10: White will now play in the blue space, leading to black losing.

However, by employing this heuristic within a search such as the MiniMax algorithm, a 'look ahead' is then done into the game's tree to see which of the currently available moves will be in-fact the most useful in the future.

5.2.1 Implementing the Heuristic

To implement the heuristic function, I intend to firstly make a separate Class. Within this class I would like to make a very re-usable function that can traverse through the rows, columns and diagonals of the board.

This is necessary as, when we look at the board in terms of a multi dimensional array, we would traverse between coordinates on the board by incrementing two values i.e. the column and the row.

I think this can be made very re-usable as we would just need to make the method in such a way that, upon receiving different parameters it will increase the specific counters to either traverse through the row or the column (or both in terms of diagonals). The code which backs the heuristic in this case is very similar to the board's code which it uses when it wants to check if any players have won.

An important issue that needs to be addressed in the method is that it needs to be able to change directions in the coordinates so that we can count a full

line of counters, rather than counting in only one direction. Once the end of the row has been met, we could then reset the starting point back to the initial point and then reverse the counting variables.

Algorithm 2 Simple Heuristic Function

```
1: function getScoredMoves
2:     for move in possibleMoves do
3:         if move does not contain Player then
4:             These method calls will add scored moves to a list
5:             findRow(position);
6:             findColumn(position);
7:             findLeftDiagonal(position);
8:             findRightDiagonal(position);
9:
10: Where checkDiagonal changes the diagonal direction
11: function findMoves(checkColumns, checkRows, checkDiagonal)
12:     winningScore = 5
13:     while currentScore != winningScore do
14:         switch the current column and row if necessary
15:         if currentPosition == currentPlayer then
16:             currentScore = currentScore + 1
17:         else
18:             if Not yet complete then
19:                 switch coordinate traversal direction
20:             else
21:                 add to Coordinate possibleMoves
22:     return score( possibleMoves, currentScore )
23: function checkLeftDiagonal(move) findMoves(true, true, false)
24: function checkRightDiagonal(move) findMoves(true, true, true)
25: function checkRows(move) findMoves(true, false, false)
26: function checkColumns(move) findMoves(false, true, false)
```

5.3 Alpha Beta Pruning

As mentioned previously, the Minimax tree grows exponentially with depth. With a game like Gomoku, which has a large search space, it would lead to an infeasible amount of waiting time whilst even trying to make simple decisions. This is of the large amount of nodes that we would need to visit to make the correct decision.

However, although the exponent can not be eliminated, it can effectively be cut in half by using pruning. Alpha beta pruning makes it possible to reach the same move that the Minimax algorithm should eventually choose by pruning away branches which it considers would never influence the finally selected node. [9]

To implement this, alpha and beta variables would need to be created which would start at negative infinity and positive infinity respectively. These variables would then get updated with the scores throughout the recursion through the tree. For example, the Beta value would get updated in the minimising stage whilst the Alpha value would get updated in the maximising stage. If these bounds are exceeded while descending the tree, this would stop the current recursion, effectively stopping that branch from being explored further i.e. pruning that branch from the search tree.

Continuing from the earlier pseudocode for MiniMax, including Alpha-Beta pruning could look like this:

Algorithm 3 MiniMax, changed for Alpha-Beta pruning

```
function initialCall
    getMinValue( board, −∞, ∞)

function getMinValue( board, alpha, beta )
    if board.isTerminal() then return evaluateState( board )
    else
        score = ∞
        for move in board.getLegalMoves() do
            score = getMaxValue( board.createMove(move), alpha, beta )
            beta = Min(beta, score);
            result = Min(score)
            if beta <= alpha then
                break
        return result

function getMaxValue( board, alpha, beta)
    if board.isTerminal() then return evaluateState( board )
    else
        result = −∞
        for move in board.getLegalMoves() do
            score = getMinValue( board.createMove(move), alpha, beta)
            alpha = Max(alpha, score);
            result = Max(score)
            if beta <= alpha then
                break
        return result
```

5.4 Depth limited MiniMax implementation in Java

To show the implementation, I will show and describe what each of the main methods of the class are doing one by one. This approach will be the minimax algorithm with depth limiting implemented, this is discussed more in section 5.5.1. After the section discussing the implementation of Iterative Deepening i.e. 5.5, I will then discuss how I implemented Iterative Deepening.

With the MiniMax implementation, I first built a constructor such that the initalPlayer i.e. the computer player would be kept safe and also the MiniMax had a board to play with. This board is a cloned copy of the 'data layer' version of the board passed from the Computer player class.

This cloned board makes it so that any changes that the MiniMax algorithm makes will not be reflected on the actual board unless it is the desired move.

Listing 3: MiniMax - Constructor

```
public class MiniMax {

    public MiniMax(Board board, Player initialPlayer) {
       this.board = board;
       this.initialPlayer = initialPlayer;
    }

}
```

As part of the design of my AI players the primary entry point, if one wants to use the algorithm directly, is *makeMove*. This method will start the algorithm and then return the move which it deems best. This move then being used by the Computer player class to update the UI and board.

Listing 4: MiniMax - makeMove

```
public Coordinate makeMove(int startDepth) {

      PositionAndScore best = decideMove(startDepth);
      return best.getCoordinate();

}
```

5.4.1 Deciding a move

In Listing 4, the method decideMove is being called. This method sets up the MiniMax algorithm and starts it. In this method, I setup the MiniMax algorithm by setting up some configuration which I find useful and important throughout the algorithm. These configurations including an easy way of seeing who the *opponent* is, who the current player is, a copy of the depth variable which can be used for comparison in the move scoring function later.

After this configuration is done, I retrieve all of the moves considered important on the board. This method uses the Heuristic class to retrieve all of the moves within some specified parameters. This heuristic method is a very useful way of decreasing the search space but, if not implemented correctly, it can lead to an all around weaker player due to not examining all of the possible useful combinations. As a forward optimisation, if the heuristic class only returns one move then this move is returned and it has a score of some unreachable number (in this case 2000) then it will be immediately returned. This is because in the heuristic class I have said that if no moves have been played yet, simply return the current centre of the board. However, this approach may not be the best and should be replaced with an implementation of something like an opening book however, in Gomoku it still leads to a good starting position.

If the algorithm progresses past this stage, this means that it is not the opening move. The algorithm then loops through each available move, tries it and then scores the outcome of that move by using the MiniMax algorithm with alpha beta pruning. After that move has been played, the position will be reset on the board so it does not affect future decisions.

This class is also public as this class can be used in an implementation such as Iterative Deepening. In the Iterative Deepening implementation, this class will be called multiple times and as such will need to return the score for the best move so far. This allows the algorithm to exit early whilst still providing a score for the current move and allows the Iterative Deepening implementation to compare the different scores of each of MiniMax instances that it has started.

Listing 5: MiniMax - decideMove

```
public PositionAndScore decideMove(int depth) {
  // Set up the important variables
  setupMinimax( depth );

  // Setup the initial moves.
  List<PositionAndScore> moves = getMoves(board, currentPlayer);
  best = null;
  double currentScore;
  // I.e. The 'default' starting move, played straight away.
  if ( moves.size() == 1 && moves.get(0).getScore() == 2000 ) {
     best = moves.get(0);
     return moves.get(0);
  }
  for ( PositionAndScore move: moves ) {
     // Make the AI player first again and take this move
     currentPlayer = board.setCurrentPlayer(initialPlayerIndex);
     board.createMove( move.getCoordinate(), currentPlayer );

     // AI player has taken move, so simulate the other player
         by minimising
     currentScore = minimiseMove(depth,
         Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY,
         move);

     // If better than current best, replace best!
     if (best == null || currentScore > best.getScore()) {
        best = move;
        best.setScore( (int) currentScore );
     }

     // Reset the move as we're done with it.
     board.resetMove( move.getCoordinate() );
     if ( finished ) break;
  }
  // Finished so set the current player to be the computer
  board.setCurrentPlayer(initialPlayerIndex);
  return best;
}
```

5.4.2 Minimising and maximising

I will now discuss the two most important methods of my minimax implementation. The minimising and maximising methods.

In terms of functionality, the implementation of the minimising and maximising methods could be combined into one. However, I feel that in terms of readability it make a lot more sense to separate the method into two. In theory they could be combined as the only differences between the two are:

- Whether or not we are scoring beta or alpha
- What the result variable is initialised to
- Handling of the current player

Due to these implementations spanning more than a single page and being very similar, I have put them in the appendix [A].

From the previous listing, you can see that for each move in the possible moves, we then start the MiniMax algorithm with the minimising method in [A.1]. This is because, the algorithm has now played one of the possible moves and the minimising method sees what the possible out come will be, starting with the opponents move. This opponent player is attempting to minimise the computer players score by maximising its own score, so it will pick the move which has the best outcome for itself.

The first thing that the minimising method [A.1] does is to check if it should stop the algorithm. It does this by checking if the depth has been fully traversed, the game is in a terminal state or the boolean flag of finished has been set. If any of these parameters evaluate to true, then the current state of the board is evaluated.

If these conditions do not evaluate to true, the algorithm then continues by getting the possible moves for the current board state. Each of the moves is then played from the current board state, the minimising algorithm then calls the maximising algorithm [A.2]. The maximising algorithm represents the initial player who wants to maximise their score.

Once the end state has been met, the algorithm will end recursively sending the results back up the tree. The result of this move is then minimised or

maximised depending which method the algorithm is in. The result is then either minimised or maximised with the value of alpha or beta as part of alpha-beta pruning. If alpha exceeds the value of beta then the bounds have been exceeded, so the current branch is pruned.

5.5 Iterative Deepening

5.5.1 Limiting the depth

For the implementation of the Minimax algorithm in this game of Gomoku, I have combined it with the heuristic and pruning methods already mentioned. However, even with the heuristic and pruning methods in place the search space will still be too large and still lead to the player taking a long time over a relatively simple move.

This is because, even though the pruning and heuristic is in place, the search space will still exponentially grow and continue growing until some end state has been met. The Minimax algorithm is basically taking into account that both players will play the same i.e. trying to maximise their own score which could lead to the tree creating nodes for the tree until the entire board has been filled.

To combat this, the Minimax tree would need to be limited by depth. This means that the Minimax tree will only create child nodes up until a certain level.

This could be done by adding to our earlier implementation of the algorithm an initial starting depth, and then telling the algorithm to exit after that depth has been exceeded:

Algorithm 4 Depth limiting in the MiniMax algorithm

```
function initialCall
    getMinValue( board, −∞, ∞, 4)

function getMinValue( board, alpha, beta, depth )
    if board.isTerminal() || depth == 0 then
        return evaluateState( board )
    else
        for move in board.getLegalMoves() do
            score = getMaxValue(
                board.createMove(move),
                alpha,
                beta,
                depth - 1
             )

function getMaxValue( board, alpha, beta, depth)
    if board.isTerminal() || depth == 0 then
        return evaluateState( board )
    else
        for move in board.getLegalMoves() do
            score = getMinValue(
                board.createMove(move),
                alpha,
                beta,
                depth - 1
             )
```

However, this would not work on its own as it would lead to incorrect decisions being made due to not going deep enough into the tree or, if the depth is set too high, it could again take too long about simple decisions.

The way that this is solved would be to include some initial sensibly low starting depth which would get incremented every time that the Minimax algorithm completes, over some time period.

5.5.2 Incrementing the depth over time

Initially, during implementation, I had attempted to increment the depth over time using a while look and a simple break which would tell the while loop to no longer create new MiniMax instances. However, this would wait for the current iteration of MiniMax to finish, this leading to the possibility of a large amount of waiting for the MiniMax algorithm to finish which would depend on the current depth level.

After seeing this, I instead decided to implement Iterative deepening with threading. In this case, a thread would be created at the start of the algorithm and in this thread the algorithm would progressively get deeper and deeper each time a MiniMax instance has finished.

At some point, the Iterative Deepening instance would notify the child instances that they should stop. I.e. I have made it so that the 'Master' Iterative Deepening class will notify the thread that it should stop, which in turn tells the MiniMax instance that it should stop. However, to stay within the time limit set, the MiniMax algorithm would need to stop instantly rather than progressively exiting out, as it could be very recursively deep which could add extra time.

The problem with this approach is that, without some extra code, the most recent MiniMax instance when stopped could lose all of the work that it has accumulated so far.

The solution I have created to this problem is that, throughout the iterative deepening process, all MiniMax instances results are continuously compared. This includes the amount of depths that the algorithm has visited and the end score (this being made up of the heuristic value and some arbitrary score to represent if the player has won or not). Finally, when the 'Master' class notifies the sub classes that they should stop, I have added some extra code which means that the MiniMax algorithm will not lose all of its work. Instead, it will return what is has managed to gain so far and compare it against the other MiniMax threads' values.

Algorithm 5 Iterative Deepening with threading

```
function iterativeDeepening
    deepeningThread = new deepeningThread()
    while  time limit has not been met  do
        result = deepeningThread.getBestSoFar()
        if  result < bestSoFar  then
            bestSoFar = result
    result = deepeningThread.getBestSoFar() > result ? result : result

function deepeningThread
    depth = 3
    while  uninterrupted  do
        setBestSoFar( new MiniMax( depth ) )
        depth = depth + 1
    interrupt minimax and getBestSoFar from it
```

Through using the depth, heuristic and an arbitrary value, I have made a strong scoring function for the MiniMax algorithm. By using the depth, the algorithm can see the urgency that a move will need to be played. For example, if the algorithm sees that it will lose on the first level of the tree, it can see that this is a very important move to play straight away in order to prevent itself from losing. At the same time, the MiniMax algorithm can exit sooner if it can see a winning move on the top level as it knows with certainty that it will win. Essentially, I see this scoring method to provide a 'confidence' value for the computer player to make a decision about a move.

Algorithm 6 Scoring MiniMax moves

```
function scoreState
    Heuristically evaluates state
    result = gameOver && winner == "Draw" ?
                0 :  evaluateMove()

    result = winner.isOpponent() ?
                ( result + depth ) - 100 :
                ( result + 100 ) - depth
```

5.5.3 Iterative Deepening implementation

To implement Iterative Deepening, we make use of the minimax implementation show in 5.4. The Iterative Deepening class is simply made up of a thread and a 'master' class which calls the thread. The master thread tells the thread to repeatedly call the MiniMax implementation once it has finished while some time has not been met. Each time the MiniMax implementation is 're-called', the maximum starting depth is incrementing. This allows the algorithm to create a more informed decision over time whilst the depth increases.

Once the time limit has been met the master class, IterativeDeepening, tells the thread to stop which tells the current instance of MiniMax to stop, getting the best move so far. After everything has finished, the MiniMax instance's score is compared against the best instance's score found so far. If it is better, then it is used.

The constructor for Iterative Deepening takes the two parameters which are important, the board which is the beginning state for this instance, and how long it should create new, deeper instances of MiniMax for.

Listing 6: Iterative Deepening in Java - Constructor

```
public class IterativeDeepening {

    public IterativeDeepening(Board board, long timeLimit) {
       this.board = board;
       this.timeLimit = timeLimit;
    }
}
```

Like MiniMax, as mentioned earlier, this method has a makeMove method which will be the entry point for the Computer player to use this algorithm directly. This method continually grabs the best move found so far by the deepening thread. If it is better than the current best, then it is set to be the best so far. If it is not the best, then this thread sleeps for 200 milliseconds. This is so the method doesn't 'spam' the deepeningThread with continual requests which would make the overall performance worse.

Once the deepeningThread says it has finished, the algorithm will exit. The

deepening thread only finishes early when it knows for sure that it has a winning combination which can not be prevented. Once it has exited, it once again checks if there was a new best move in case another instance of MiniMax has just finished.

Listing 7: Iterative Deepening in Java - makeMove

```
public Coordinate makeMove( int startDepth ) {
   long start = System.currentTimeMillis();
   PositionAndScore best = null;
   // Create the deepening thread.
   DeepeningThread deepeningThread = new DeepeningThread(board,
       startDepth);
   deepeningThread.start();
   // While time limit not finished or deepening not finished
   while ( System.currentTimeMillis() < start + timeLimit ||
       bestSoFar == null ) {
      // Get the best so far, might be null
      best = deepeningThread.getBestSoFar();
      if ( best == null ) {
         // Try again in 200 milliseconds
         try {
            Thread.sleep(200);
         } catch (InterruptedException e) {
             LOGGER.severe(String.valueOf(e)); }
      } else {
         setBestSoFar( best );
      }
      // If finished, break!
      if ( deepeningThread.getFinished() ) {
         break;
      }
   }
   // Shut down the thread and minimax
   deepeningThread.setFinished();
   // Checks if there was any better moves
   setBestSoFar( deepeningThread.getBestSoFar() );
   // Return our best move
   return bestSoFar.getCoordinate();
}
```

The deepening thread itself is quite simple. The class has a method with a *for loop* which will continually deepen the MiniMax instances after the previous one has finished, until it has been told to stop. For each instance, it scores the best one it finds so far, meaning that it can be safely interrupted whilst ensuring that a move is available. If the the thread finds that MiniMax has returned it a score greater than 100, then it knows that it is onto a combination of moves that can't be prevented and has definitely won, so it exits early.

Listing 8: Iterative Deepening thread in Java

```
public void run() {

    PositionAndScore best = null;

    // Loop until finished or IterativeDeepening interrupts
   for ( ;; depth++ ) {
      // Create a new MiniMax instance with the new depth
      minimax = new MiniMax(board, board.getCurrentPlayer());
      PositionAndScore current = minimax.decideMove(depth);
      // Attempt to add the current to the best score
      if (best == null || current.getScore() > best.getScore()) {
         best = current;
         setBestSoFar( best );
      }
      // If the score - depth is greater or equal top 100, this
          means that we have definitely won
      if ( best.getScore() - depth >= 100 || finished ) break;
   }

    // Attempt to set minimax bestSoFar as the best (in case best
        is null)
    setBestSoFar( minimax.getBestSoFar() );

    // Set finished to true, used by IterativeDeepening
    finished = true;

}
```

5.6 Monte Carlo Tree Search

The Monte Carlo Tree Search is another algorithm based on random sampling aimed at targeting problems with large search spaces. It is part of the Monte Carlo methods which date back to the 1940s, they are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results [13].

In 1987, Bruce Abramson explored the algorithm for games such as Chess, Othello and TicTacToe. In his paper, he found that the expected value proposed for a node from the expected outcome model, was the value of a game's outcome based on random play outs from that node onwards. He found this method of scoring moves to be *precise, accurate, easily estimable, efficiently calculable, and domain-independent* [1].

In the case of this project, domain in-dependency means that the algorithm could be applied to multiple games without having to encode game specific knowledge. However, combining Monte Carlo Tree search with game specific knowledge in the form of heuristics is referred to as a *heavy playout* which I will discuss more later.

5.6.1 Multi-Armed Bandit Problem

Bandit problems are part of the class 'sequential decision problems' where one needs to choose from K actions in order to maximise the cumulative reward by consistently choosing the optimal action from the available actions [5].

The case of the multi-armed bandit problem describes the situation where a gambler i.e. the computer player, is faced with a row of slot machines i.e. one-armed bandits. The gambler would like to maximise their net gain through a sequence of lever pulls from a selection of machines. The gambler would then decide how many times each of the chosen machines should be played. Each of these machines would provide the gambler with some reward based on a probability distribution specific to that individual machine. The ideal strategy to this problem would be one that balances playing all of the machines to gather the information with concentrating the plays on the observed best machine. One such strategy is called UCB-1. [4]

5.6.2 UCB-1

As mentioned, the difficulty with the multi-armed bandit problem is finding the balance between playing all of the machines to gather information whilst concentrating the plays on the observer best machine. More formally, rather than describing the problem in the case of machines we could describe this in terms of a tree. From the view point of a tree, we would want to select child nodes from the current root by finding a balance between *exploitation* ***vs.*** *exploration*. By exploitation vs exploration I mean that we need to find a balance between the exploitation of known rewards vs the exploration of unvisited nodes in the tree. The reward estimate for a node is based on a random simulation from that node. Each node must be visited a certain amount of times before that move can be seen as reliable enough to play from.

The formula to balance this exploitation vs exploration was introduced by Kocsis and Szepervari. They formalised it by extending UCB to the minimax tree search and as such named it Upper Confidence Bounds for Trees i.e. UCT. This algorithm is used in the vast majority of MCTS implementations [7].

To choose the best node in the tree, Kocsis and Szepervari recommended to choose the node which has the highest value according to the following formula which I will first show mathematically and then as pseudocode.

$$\frac{w_i}{n_i} + c\sqrt{\frac{\log t}{n_i}}$$

Where:

- w_i - the wins for the current node
- n_i - the number of visits for the current node
- c - the exploration parameter which is usually chosen empirically
- t - the total number of simulations for the nodes considered, i.e. the total sum of all n_i

In the following pseudocode, the exploration parameter i.e. c, is set to the value epsilon.

Algorithm 7 UCB-1

```
epsilon = 1e-6

function getUCBScore
    return ( childNode.getWins() / childNode.getVisits() ) + epsilon *
            sqrt ( log ( parent.getVisits() / child.getVisits() ) )
```

5.6.3 Selection

The selection stage of the Monte Carlo algorithm is where the UCB formula is used. The selection phase lasts while statistics exist to treat each position met like a multi-armed bandit problem. The move which would then be selected would be chosen through the use of the UCB formula, rather than randomly, and applied to the next position to be considered. The selection phase would then continue until a child is reached where no statistics have yet been generated for the children [4].

In the form of pseudocode, this can be visualised as:

Algorithm 8 Monte Carlo - Selection

```
function selection
    while curentNode's children has statistics do
        currentNode = makeUTBSelection( currentNode, children )
        make UCB selected move on the board
    return currentNode

function makeUCBSelection(parent, childNodes)
    for node in childNodes do
        if ucbValue > best then
            best = ucbValue
    return best
```

5.6.4 Expansion

The expansion phase of the Monte Carlo algorithm begins after Selection has finished i.e. it begins when we no longer have the statistics to treat each position as a multi-armed bandit problem. During the expansion phase, of the currently unplayed children a move is randomly selected and played.

In the form of pseudocode, this can be visualised as:

Algorithm 9 Monte Carlo - Expansion

```
function EXPANSION( children )
    move = getRandomLegalMove ( children )
    make move on the board
    return move
```

5.6.5 Simulation

After a random child node has been selected from expansion, the simulation phase then begins. The simulation phase works by continually playing with itself on a copy of the current board state until a winner has been found. The simulation would select moves randomly if it using light playouts however, this will usually only perform well with a low branching factor [4] i.e. this wouldn't work very well for Gomoku. Instead, heavy playouts would need to be used for this game.

Using random, light playouts, the simulation phase could look like the following pseudocode. Converting this to heavy playouts could be as simple as getting the moves via the heuristic already implemented rather than randomly.

Algorithm 10 Monte Carlo - Simulation

```
function SIMULATION( node )
    while game is not over do
        board.createMove( some random legal move )
        switch the current player
```

5.6.6 Back Propagation

Now that the simulation phase has finished. The board should now have a winner, loser or be in a state where neither player can play any further i.e. a draw. This result now needs to reflect in the tree, to show the path that has been taken to get to this node.

The back propagation phase is done recursively to make its way back up the tree and score the nodes with the result that has been obtained from the simulation on this node. This needs to be done from the 'perspective' of the current winner i.e. as the algorithm recurses up the tree a visit counter for that node could be incremented. However, the win counter is only incremented if the player who played that move is also the winner.

Algorithm 11 Monte Carlo - Back Propagation

```
function backPropagation( node )
    As the root node has no parent, this will break when the root is reached
    while node has a parent do
        node.incrementVisit()
        if node.getPlayer() == winner.getPlayer() then
            node.incrementWins()
        node = node.getParent()
```

5.6.7 Full MCTS Pseudocode

After the back propagation phase has completed, the whole process begins again with the root node. However, thanks to the last round of back propagation, statistics now exist for the current path in the tree. The selection phase will use UCB to find its way through the tree up until a point where the current node's children no longer have statistics and so the process continues.

The algorithm will continue until some condition is met. This is usually done by setting a value such that a for loop will continually run the algorithm until this value has been met. However, in most instances it will also be desirable for the algorithm to be time limited. This could be done similar to the earlier discussed implementation of threaded iterative deepening, where new

instances are created when other instances have finished until a time limit has been met.

Algorithm 12 Monte Carlo - Full algorithm

```
function MCTS( rootNode, maxMoves )
    for i < maxMoves do
        selection phase
        expansion phase
        simulation phase
        back propagation phase
    return getBest( rootNode's children )

function MCTSTHREAD( timeLimit )
    while time limit not met do
        mcts( rootNode , maxMoves )
    return move from best mcts instance
```

5.6.8 MCTS Implementation

Like the deepening thread, I have implemented two main classes for the Monte Carlo Tree Search implementation. The 'parent' class starts the thread and keeps the MCTS algorithm going until the specified time period. This gives the MCTS algorithm a chance to find possible better moves by building the tree multiple times. It is important to create multiple trees and compare the results as, due to the randomness involved in MCTS and the various limits, it is possible that the algorithm might make an incorrect choice and end up with a result that is not that beneficial to the current state.

In Listing 9, while this instance has not been interrupted, the algorithm will continually build new Monte Carlo Tree Search trees (using a clone of the main board) until the time has limit has been met. Once the algorithm has finished, it sets the boolean variable finished to true, this notifies the 'parent' class that it has finished and it can continue by deciding which 'node' out of those provided by the different MCTS trees had the best result.

Listing 9: MCTS in Java - run method

```
public class MonteCarlo {

    public void run() {

       finished = false;
       interrupted = false;
       bestNodes = new ArrayList<>();

       while ( !interrupted ) {

          // The rootNode has not been expanded and therefore will
              not yet have any children
          Node rootNode = new Node(board.clone(), null);

          mcts(rootNode);

          // Return the best move from the tree, scored through
              back propagation
          addBestNode(rootNode.getPlayedChildren());

       }

       finished = true;

    }

}
```

Then following the Monte Carlo Tree Search algorithm, while some arbitrary number has been met the tree is built with nodes. This arbitrary number is essentially saying how many nodes will be built up in the tree, the more nodes being added usually makes the end decision better. The concept of this is similar to the depth limiting which was shown in the iterative deepening section 5.5.1. However, I have noticed it is important to find the right balance between how long the algorithm will run for and how many moves the algorithm should be limited by. I have found this is the case, as if the correct balance is not found then the algorithm performs really badly.

A good balance I have found for quite a few games are the limits of 20000 moves and 20000 milliseconds. In this configuration, the algorithm uses the full time limit and does not create multiple trees early on in the game. However, as the game comes closer to the end the algorithm's simulations are faster and it builds multiple trees which it can compare against at the end of the game.

The root node is then made the active node and the board state for that root node is cloned. On each iteration the active node is reset to the root node so that the selection and expansion phases begin at the root, this possibly opening new branches of the tree which were previously unexplored.

On the first run, the selection phase [A.3] will not actually select a move. This is because UCB can only select a move if all of the children have statistics which it compare against to select a move. The expansion phase [A.4] then begins if the board is not in a terminal state and the current node still has unplayed children. The expansion phase will randomly select an unplayed child node, expand it and make it the current active node. The simulation phase [A.5] will then begin from this randomly selected node playing either completely random, semi-random or heuristically chosen moves until some end state is met. The end result is then back propagated [A.6] back through the nodes all the way back to the root. If *maxMoves* has not been exceeded then the tree will continue to be expanded and the process begins again from the root node.

Listing 10: MCTS in Java - mcts method

```
private void mcts( Node rootNode ) {
   for ( int i = 0; i < maxMoves; i ++) {
      // Make the current node the root node
      Node node = rootNode;
      boardState = rootNode.getBoard().clone();
      // Gathers the child nodes which have not yet been played
      node.findUnplayed();
      // Selection
      node = selection( node );
      // Expansion
      if ( getWinner( node.getBoard() ) == null &&
          !node.getUnplayedChildren().isEmpty() ) {
         Node expandedNode = expansion( node );
         if ( expandedNode != null ) {
            node = expandedNode;
         }
      }
      // Simulation
      simulation();
      // Back propagation
      backpropagate( node );
      if ( interrupted ) break;
   }
}
```

6 Experiments and Discussions

6.1 Experiments

For the experiments section of this report, I will be comparing the different AI approaches that I have discussed in competitive play against each other. I have collected the data from making different combinations of AI players play a number games against each other and with different configurations.

I believe that the data gathered from these experiments will help to show not only the strength of the approaches but to also provide a clear comparison of how strong one approach is against another.

My prediction will be that the Heuristic will be the weakest approach due to being a 'short-sighted' greedy approach and not looking ahead. This being followed by the Depth Limited MiniMax due to the approach not being able to look too far into the game tree. However, different results could appear for different depth limits but this would be an unfair comparison against the other approaches due to taking significantly longer to complete.

I think that the Iterative Deepening and Monte Carlo Tree Search implementations that I have implemented will be quite evenly matched and will be the two best approaches of this project. I think the approaches might be quite even for multiple reasons, one of these being that firstly being that they are backed by the same heuristic. Normally, one would expect the Monte Carlo Tree Search approach to easily beat an iterative deepening one however, I think that due to the threading and 'early exit' implementation I have implemented in the Iterative Deepening approach it has evened the playing field a bit more.

6.2 Results and Conclusions

6.2.1 Set Up

As a brief summary, these are the configurations used for the games:

- The games were played on a 19 x 19 board
- 5 counters in a row or more were required to win
- The time limited algorithms were limited to 15 seconds
- The Monte Carlo Tree Search was limited to trying 2000 moves

In the following graphs, the numbers reflect **boolean values** i.e. True = 1 and False = 0. For example, if a player has won, then their value on the graph will be at 1.

6.2.2 Heuristic VS other algorithms

In the following experiments, Heuristic is player 1.

As predicted, the heuristic performed the worst in the games. However, it performed evenly against the depth limited MiniMax. However, this would not be the case if the MiniMax was limited at at a larger depth but then the trade-off would be that the algorithm would take a lot longer.

In Figure 11, it shows that whoever is first player between the heuristic and the MiniMax player is the winner in the game. This corresponding to the fact that it is normally the first player who will win in freestyle Gomoku.

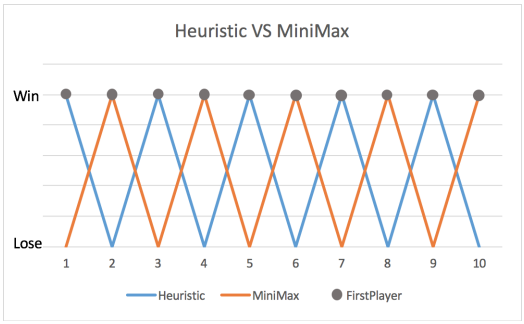

Figure 11: Heuristic is even with depth limited MiniMax.

In Figure 12 and Figure 13 it is clear to see that regardless of who is first player, both the Monte Carlo Tree Search and iterative deepening players consistently beat the heuristic.

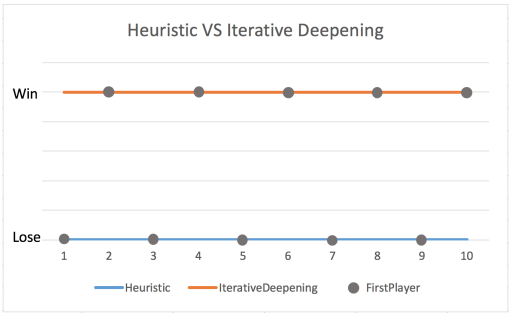

Figure 12: Heuristic consistently loses to Iterative Deepening

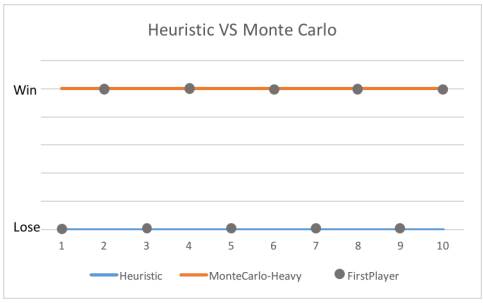

Figure 13: Heuristic consistently loses to Monte Carlo Tree Search.

6.2.3 Depth Limited MiniMax VS Iterative Deepening

In the following experiments, MiniMax is player 1.

I believe this result would be as expected, that the Iterative Deepening would consistently beat the MiniMax algorithm. This would be the case as the Iterative Deepening is an improvement on top of the depth limited MiniMax algorithm, allowing the algorithm to explore the tree even further for the specified amount of time.

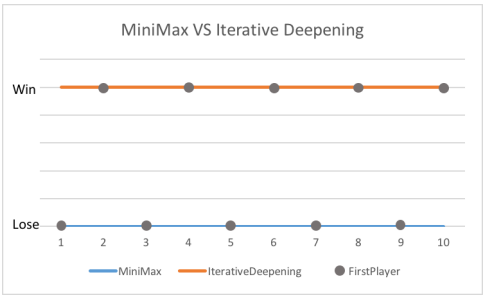

Figure 14: MiniMax consistently loses against Iterative Deepening.

6.2.4 Iterative Deepening VS Monte Carlo Tree Search

In the following experiments, Iterative Deepening is player 1.

I was not expecting result shown in Figure 15, however it turns out that the iterative deepening algorithm almost consistently beats the Monte Carlo Tree Search on a 19 x 19 board. However, I believe that this also makes sense. I believe this makes sense as the Monte Carlo Tree Search algorithm, as mentioned, requires a strong heuristic function within the simulation phase for large search spaces. In this case, I think that the heuristic function I have created may be strong enough for the iterative deepening but not for the Monte Carlo Tree Search on a board of this size.

However, one note that I did make while watching the two algorithms compete is that, they both seem to make logical decisions but there is a main difference. This main difference being that the MCTS (Monte Carlo Tree Search) approach sometimes attempts to play a few counters away from the main counters and build up 'combination moves' that way. This is different to the iterative deepening approach which consistently plays next to existing counters, even building up combination moves this way leading to a strong end result.

I believe that it is because the Iterative Deepening player keeps the counters so close to the existing ones that it ends up the stronger player in this case as eventually so many counters are close together that, even if some are blocked, the unblocked counters combined together will eventually lead to a win.

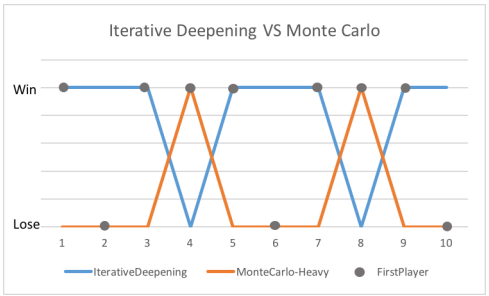

Figure 15: Iterative Deepening VS Monte Carlo Tree Search (19x19)

Due to this result not being in line with my initial prediction, I will now also show what the Monte Carlo Tree Search player looks like against the Iterative Deepening player on a smaller board configuration. On this smaller board configuration, the board is of size 9 and the winning combination is that of 4 or larger.

In this configuration, the results are closer to what I would expect. In this setting, whichever player goes first is the winner. However I believe that, as mentioned, if the Monte Carlo algorithm was improved it would beat the Iterative Deepening more often and this may not be a fair comparison due to the number of improvements done to the Iterative Deepening implementation.

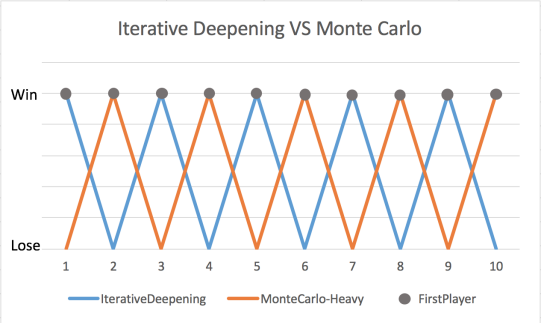

Figure 16: Iterative Deepening VS Monte Carlo Tree Search (9x9)

7 Future Work

7.1 Improving the Heuristic

In the future, to improve this Heuristic we could also add other conditions or expert knowledge to further evaluate the value of a move. For example, we could do a 'jump' over an arbitrary number of certain spaces to see what else is there. This jump could reveal that there is a 'wall' blocking the possibility of a full 5 connection or it could reveal that there are already some of the Computer player's already there, strengthening the value of the move.

From improving the heuristic, all of my other implementations (i.e. minimax, iterative deepening and MCTS with heavy playouts) would also improve how they will play

7.2 Improving MiniMax and Iterative Deepening

Although the AI player combined with the heuristic and iterative deepening is quite strong, there are still improvements to be made in the future.

One possible improvement that I could apply to the Iterative Deepening algorithm would be, instead of waiting for a MiniMax instance to finish I could spawn multiple instances which will all work at the same time and then interrupt them all at the some time period and compare the results.

Another improvement I could implement would be performance features such as a Transposition Table. This is effectively the practice of memoization applied to tree search in that, different game states will be stored in a hash table. These game states will be then compared against in e.g. the MiniMax tree so that if it is encountered earlier, then we could effectively prune more possibilities from the game tree by preventing identical states from having to be possibly analysed multiple times. This would be a useful improvement to have in the algorithm as it would mean that the algorithm could get deeper into the tree sooner, resulting in a better move being eventually chosen.

7.3 More algorithms

7.3.1 Threat Space Search

My heuristic acts in a way that is already similar to the Threat Utility function i.e. by using a greedy approach and scoring moves most valuable to it's current state based on threats and possible rewards. However, I think that it would also prove interesting to see how the Threat Space Search approach would play against my existing implementations being that I have two strong implementations which are also backed by my heuristic i.e. Monte Carlo Tree Search with heavy playouts (the heuristic) and threaded iterative deepening.

The Threat-space search approach was developed to model the way that a human Gomoku player finds winning threat sequences. It was found in a 1993 paper [2] that conventional tree-search algorithms do not mimic human experts when searching for a winning threat sequence. The approach detailed in the paper show that the player is very strong and even won Gold in the Computer Olympiads winning all games using black i.e. first player; and winning half of the games using white.

7.3.2 Reinforcement Learning

Another approach, which would be interesting to view the results of is Reinforcement Learning. I believe it would be interesting as it has been proven that an approach based on Reinforcement Learning and the Monte Carlo Tree Search can create a very strong player. Such was the case in 2016, where Google found success with their AI player AlphaGo in the game of Go by beating the human champion Lee Sedol [8].

The approach used by Google involved using Monte Carlo Tree Search guided by a value and policy network which were both implemented by a deep neural network [10]. Once the player had proceeded to a proficient level it was further trained on instances of itself using reinforcement learning. However, as mentioned in the article, Google leveraged their vast amount of data centres and internet connection in order to beat the champion Lee Sedol.

From this, I believe it would be interesting to see the following points in a

future implementation:

- How well a reinforcement learning approach alone plays VS the existing approaches I've implemented
- How well the approach implemented by Google compares to the previously mentioned item and other approaches I've implemented
- How powerful all of the approaches are when greatly increasing the resources available to the algorithms e.g. by leveraging external resources from other machines

7.4 Game Improvements

As mentioned earlier in the report, I have implemented the game using freestyle Gomoku. However, as mentioned, in freestyle Gomoku the first player i.e. black has a much higher chance of winning. To combat this I think it would be important to implement different approaches of gameplay, such as Renju rules. Renju rules attempt to make it fair for both players in the game and is an approach used often in tournaments.

7.5 Testing

Although I have created tests for the important parts of the system and the parts I particularly wanted to make sure were behaving properly, I would have liked to create more tests. I think this is important as it shows that a system is stable and also that it behaves as expected.

Going forward, I believe it would be useful to eventually create a test for each important piece of functionality. An example of this is that a setter and getter could be tested using *mocking* where the output from a getter would be expected to be the same as some mock data.

8 Reflection

In reflection to the project, I think that there are some goals I have achieved and others which I believe I could have done better on. I believe that I have implemented a generally strong code base with a solid User Interface whereby a user can play Gomoku with either a human or Artificially Intelligent player easily. I am particularly pleased with the way I have managed to implement the board so that it is very configurable in that you can e.g. pass in different dimensions and all of the rest of the logic will update and the User Interface will scale to the change.

I think that the AI players that I have implemented follow the sources I have listed well and have been implemented error free. However, I wish that I could have added more to the implementation of the Monte Carlo Tree Search player so that it could be stronger. I believe that it could be a lot stronger as the impression I have gotten from the sources listed is that, with the correct heuristic, the player should easily and perhaps consistently beat the iterative deepening player.

Another thing I would improve if I could do the project again is to try and plan my time a bit better. Although I have been continually working on this project throughout the allotted time span, I believe that I have wasted a few days on tasks which could have been completed quicker. This wasted time could have been used to improve the Monte Carlo player and also possible make the other improvements listed in the future work section [7].

Overall, I believe what I have implemented has been implemented well and could provide a strong starting base for either myself or another to continue development on. I believe that the next steps for this project would entail mainly, as mentioned, creating a stronger Monte Carlo player but also making the other enhancements mentioned in the future work section [7]. I believe that the AI approaches I have listed in the future work section would add to the completeness of what I have found and create an interesting comparison.

Over the course of this project, I have learnt that it is definitely important to do literature reviews and to fully read the articles you are reading. I have had numerous problems due to the fact that I hadn't fully or properly read a literature review but when going back later found the information I had

missed. An example of this is that I didn't realise that the Monte Carlo Tree Search should be back propagated from the perspective of the winner however, this has now been changed.

Going forwards, I believe that this project has helped be to improve my own skill set in various ways. Previously I have not had experience with JavaFX or JUnit which I believe will be useful going forward in places such as industry. I also believe in the fact that it is important to first do designs and then translate these designs into the coded form as this helps to keep make the end result stable and perform well. Some other minor points where I have improved upon in this project is that I hadn't used LaTex before and I can now see the value in its usage. I have also come to the realisation of the importance of providing both pseudocode and real code as I believe it is important to provide a language agnostic way of describing the code whilst also providing how this could possibly translate into real code. I also believe that a benefit from this project is that I have improved my knowledge of tree searches and Artificial Intelligence in general. I believe this will be useful in the future if I intend to research it further or use it in my own projects.

References

[1] Bruce Abramson. The expected-outcome model of two-player games. Report, Columbia University, 1987.

[2] L.V. Ailis, H.J. van den Herik, and M.P.H. Huntjens. Go-moku solved by new search techniques. Report, AAAI, 1993.

[3] L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence/ Louis Victor Allis*. Ponsen & Looijen, 1994. ISBN 9789090074887. URL `https://books.google.co.uk/books?id=c7FTAgAACAAJ`.

[4] Jeff Bradberry. Introduction to monte carlo tree search, September 2015. URL `jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/`.

[5] C Browne, E Powley, D Whitehouse, S Lucas, Peter C, P Rohlfshagen, S Tavener, D Perez, S Samorhtakis, and S Colton. Ieee transactions on computational intelligence and ai in games. Report, ieee, 2012.

[6] Gomocup. Gomocup - detail information, February 2016. URL `http://gomocup.org/detail-information/`.

[7] Levente Kocsis and Csaba Szepesvri. *Bandit based Monte-Carlo Planning*. Springer, 2006.

[8] Cade Metz. Alphago vs lee sedol, April 2016. URL `http://www.wired.com/2016/03/go-grandmaster-lee-sedol-grabs-consolation-win-googles-ai/`.

[9] Stuart Russel and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall, 3 edition, 2010. ISBN 860-1419506989.

[10] David Silver and Demis Hassabis. Alphago: Mastering the ancient game of go with machine learning, January 2016. URL `http://googleresearch.blogspot.co.uk/2016/01/alphago-mastering-ancient-game-of-go.html`.

[11] Wikipedia. Separation of concerns, April 2016. URL `https://en.wikipedia.org/wiki/Separation_of_concerns`.

[12] Wikipedia. Gomoku, February 2016. URL `https://en.wikipedia.org/wiki/Gomoku`.

[13] Wikipedia. Monte carlo methods, April 2016. URL https://en.wikipedia.org/wiki/Monte_Carlo_method.

[14] Wikipedia. Solved games, April 2016. URL https://en.wikipedia.org/wiki/Solved_game.

[15] J Wgner and I Virg. Solving renju. Journal, ICGA, 2001.

Appendix A Java Code Listings

A.1 MiniMax - minimising

Listing 11: MiniMax - Minimising

```
public double minimiseMove( int depth, double alpha, double beta,
    PositionAndScore takenMove ) {
   double result;
   // Check the current score
   WinState winner = board.checkWinner();
   if ( depth == 0 || isGameOver( winner ) || finished ) {
      // Depth has been reached or game is over so score!
      result = scoreState( isGameOver( winner ), winner, depth,
          takenMove );
   } else {
      // Get the possible moves
      result = Double.POSITIVE_INFINITY;
      List<PositionAndScore> moves = getMoves(board,
          board.getPlayer( opponentIndex ));
      for ( PositionAndScore newMove : moves ) {
         currentPlayer = board.setCurrentPlayer( opponentIndex );
         board.createMove( newMove.getCoordinate(), currentPlayer );
         // Score the move!
         double score = maximiseMove( depth - 1, alpha, beta,
             newMove );
         // We're done with it, remove it from the board.
         board.resetMove( newMove.getCoordinate() );
         // Minimise the move score
         result = Math.min(result, score);
         beta = Math.min(beta, score);
         // Out of bounds so break
         if ( beta <= alpha ) {
            break;
         }
      }
   }
   return result;
}
```

A.2 MiniMax - maximising

```
public double maximiseMove( int depth, double alpha, double beta,
    PositionAndScore takenMove ) {
   double result;
   // Check the current score
   WinState winner = board.checkWinner();
   if ( depth == 0 || isGameOver( winner ) || finished ) {
      // Depth has been reached or game is over so score!
      result = scoreState(isGameOver( winner ), winner, depth,
          takenMove );
   } else {
      // Get the possible moves
      result = Double.NEGATIVE_INFINITY;
      List<PositionAndScore> moves = getMoves(board,
          board.getPlayer(initialPlayerIndex) );
      for (PositionAndScore newMove : moves ) {
         currentPlayer = board.setCurrentPlayer( initialPlayerIndex
             );
         board.createMove( newMove.getCoordinate(), currentPlayer );
         // Score the move
         double score = minimiseMove( depth - 1, alpha, beta,
             newMove );
         // Remove the move, we're done with it.
         board.resetMove( newMove.getCoordinate() );
         // Maximise the move score
         result = Math.max(result, score);
         alpha = Math.max(alpha, score);
         // Out of bounds so break
         if ( beta <= alpha ) {
            break;
         }
      }
   }
   return result;
}
```

A.3 MCTS - selection

```
private Node selection( Node node ) {

      while ( node.getUnplayedChildren().isEmpty() &&
          !node.getPlayedChildren().isEmpty() ) {

         node = makeUCTMove( node, node.getPlayedChildren() );

         // Create the move UCT move on the board
         // ( Players will switch due to node being re-assigned to
             child move )
         boardState = node.getBoard().clone();

      }

      return node;

}

private Node makeUCTMove ( Node parent, List<Node> children ) {

      Node best = null;
      double bestScore = Double.NEGATIVE_INFINITY;

      for ( Node child : children ) {

         double uctValue = ( child.getWins() / child.getVisits() )
             + epsilon * Math.sqrt( Math.log( parent.getVisits()
             /child.getVisits() ) );

         if ( uctValue > bestScore ) {
            best = child;
            bestScore = uctValue;
         }

      }

      return best;
```

```
}
```

A.4 MCTS - expansion

```
private Node expansion( Node node ) {

      Node move = node.expand();

      boardState = move.getBoard().clone();

      return move;

}
```

A.5 MCTS - random simulation

```
private void simulation() {

      while ( !hasSimulationMoves() ) {

         // The game has ended i.e. a Draw or a player has won, so
             exit
         if ( gameEnded(boardState) != null ) {
            break;
         }

         // Switch player
         boardState.setCurrentPlayer(
             boardState.getCurrentPlayerIndex() == 0 ? 1 : 0 );

         // Create a random move on the board for the current player
         boardState.createMove(
            boardState.getLegalMoves().get( random.nextInt(
                boardState.getLegalMoves().size()) ),
            boardState.getCurrentPlayer()
         );
```

```
    }

}
```

A.6 MCTS - back propagate

```
private void backpropagate( Node node ) {

    WinState gameEnd = gameEnded( boardState );
    Player winner = getWinner( boardState );

    // If the parent is null, we are back at the parent
    while ( node != null ) {

        node.visitNode(gameEnd, winner != null &&
            winner.equals(node.getPlayer()) );

        node = node.getParent();

    }

}
```