Cassio: An Artificial Intelligence Machine for Othello

Student Name: Matthew Fennell

Supervisor Name: Tom Friedetzky

Submitted as part of the degree of MEng Computer Science to the

Board of Examiners in the School of Engineering and Computing Sciences, Durham University

Abstract —

Context/Background Artificial Intelligence (AI) has been on the rise for years, but with the success AlphaGO has had with machine learning techniques, the concepts behind these algorithms are becoming more widely used as their potential is realised. Whilst these techniques can be readily applied to perfect information games, learning about the manner in which they operate will give an insight to where these techniques can be applied outside of the field that popularised them (LLC 1999).

Aims The aim of this project is to implement an AI (Cassio) for Othello that plays intelligently, where at the start of the game each square on the board has an equal probability of becoming unplayable thus diversifying the board designs. The underlying algorithms behind Cassio will possess the ability to adapt to each board design so that obscure and unique board layouts do not hinder the capability of intelligent play.

Method Modern machine learning approaches are used to solve the problem which is ultimately a search problem. Nodes represent game states which are used to generate game trees, enabling the algorithms to determine the strategy of the game and subsequently select intelligent moves. A Graphical User Interface (GUI) is implemented in C++ using the OpenCV library for interactive game-play.

Results Increasing the number of simulations for the Monte Carlo results in a more intelligent AI, however, the time cost is significant. The Minimax approach is heavily reliant on the strength of the evaluation function. Using a strong evaluation function produced from the genetic algorithm, the Minimax program consistently beats the Monte Carlo program on both standard and non-standard boards.

Conclusions This project demonstrates an AI based on the Minimax algorithm that shows outstanding promise whereas the performance of the Monte Carlo algorithm is underwhelming. The use of randomly generated boards does not hinder the ability of the Minimax program because the evaluation function still captures the critical pieces of information required to play intelligently such that it is still able to defeat the Monte Carlo programs with ease.

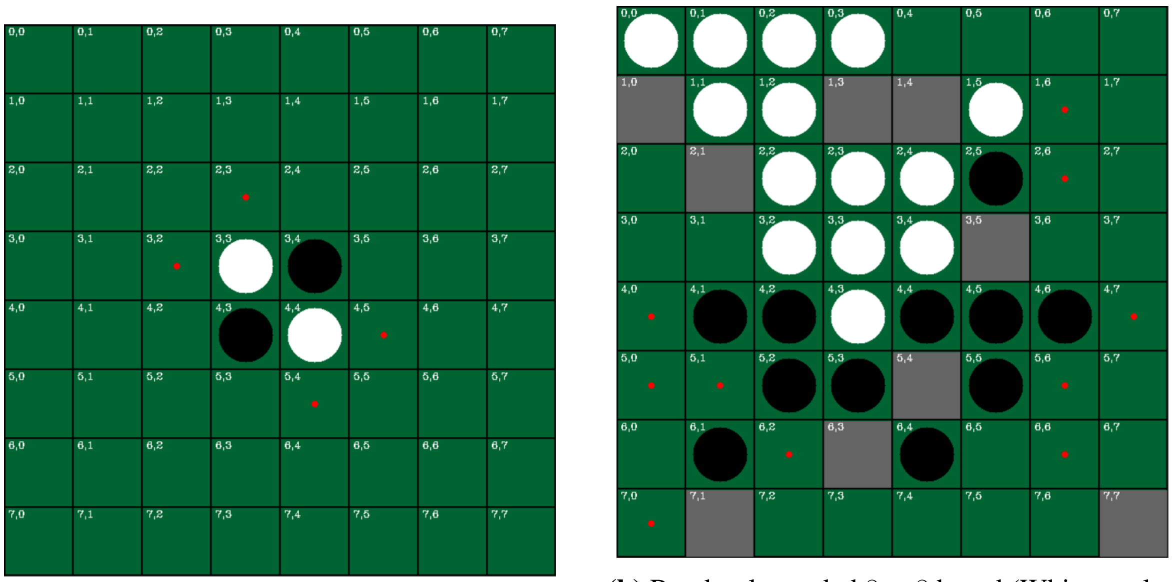
Keywords — Othello, Minimax, $\alpha - \beta$ Pruning, Monte Carlo, Heuristics, Random Boards, Evaluation Function, Genetic Programming

I INTRODUCTION

A The Rules of Othello

Othello is a two player perfect information board game (there is no information hidden from either player) where players take it in turns to place a disc in an empty square on the board. The game begins with four discs placed on the board (Figure 1a). Black plays first and must place a disc in a location that would cause a white disc to become trapped (horizontally, vertically or diagonally), where all discs trapped by the newly placed disc are flipped and change color. For example, in Figure 1a there are initially 4 moves available to black shown as red dots: *(2,3)*, *(3,2)*, *(4,5)*, and *(5,4)*. If black selected *(2,3)*, the white disc in *(3,3)* would be trapped by two black discs and subsequently turn black.

Players may only place discs in locations that cause at least one opposing disc to flip. If there are no legal moves available to a player then the turn is passed to the other player and if there are no legal moves for either player then the game ends with the winner being the player with the highest number of discs in their color.



(a) Regular 8 × 8 Othello board

(b) Randomly seeded 8 × 8 board (White to play next)

Figure 1: Othello Board Representation in the GUI using OpenCV, with red dots representing the moves immediately available to the active player and grey squares for unplayable locations

This project looks at a variation of Othello where non-standard boards are used. To achieve this, at the beginning of each game every empty square has an equal probability of becoming unplayable. These unplayable squares act as physical holes in the board, breaking all connections. Figure 1b gives an example of what is entailed from non-standard board designs. As a result of an unplayable square in *(6,3)*, White is unable to play in *(7,3)*. The expectation is that these small adaptations to the regular board have the potential to significantly alter the strategy that must be adopted in order to play intelligently.

B Purpose of the Project

The purpose of this project is to develop Cassio into an intelligent Othello AI that can play strategically at a high level on various board designs. This project also looks to investigate how non-standard boards can influence the strength of different techniques. There are already several Othello AIs with papers describing effective techniques that have been applied; Logistello (Buro 1997), BILL (Lee & Mahajan 1990) and IAGO (Rosenbloom 1982) are some of the most dominant, but none of these consider competing on non-standard boards. Some of the concepts used in these programs are applicable, given slight modifications so that they can perform on any board design.

For small games such as Naughts and Crosses, it is possible to solve the game using a brute force approach to evaluate every potential line of play. Conversely for Chess, if every atom of the universe was evaluating leaf nodes of the game tree at nanosecond speed since the beginning of time, not all of the leaf nodes would be evaluated to date. Whilst Othello does not have as large a tree depth (d) or branching factor (b) as Chess, it is more similar to Chess than to Naughts and Crosses where the number of leaf nodes grows exponentially at a rate of $\theta(b^d)$, so a brute force approach is clearly not suitable. More formally, Othello falls under the complexity class of PSPACE-complete (the amount of memory required to solve it is polynomial in the input length) whereas Chess is EXPTIME-complete (decision problems with exponential runtimes). For this reason, algorithms are required to not evaluate all potential lines of play, but to quickly disregard and ignore any lines of play that are less desirable, focusing on the more promising moves.

Minimax is a technique commonly applied to perfect information games, forming the basis of BILL, IAGO and Logistello. Cassio experiments with variations on Minimax and tests it against other techniques such as the Monte Carlo Tree Search approach, one which has become more popular with the success of Google's AlphaGO (Lee et al. 2016). The Monte Carlo technique develops an understanding of the game without being told the strategy, whereas Minimax relies heavily on an evaluation function, so it is of particular interest to see how the two methods compare on randomly generated boards where the design of an evaluation function becomes more difficult.

C Deliverables

As a high level goal, this project aims to develop a versatile AI for Othello and to investigate the relative affect on the performance of different techniques when competing on non-standard boards. This is achieved by splitting deliverables into basic, intermediate and advanced objectives. The basic objectives are to produce a correct and complete implementation of the board mechanics, alongside a GUI to visualize the board and provide a simple interface for users to select moves. The GUI will also provide a method to allow the generation of random boards for different seeds, where a larger seed results in a higher probability for any square to become unplayable. The intermediate objectives are to implement an advanced AI based on modern machine learning approaches (i.e. the Monte Carlo Tree Search), with an attempt to minimise memory usage. The advanced objectives are to develop an AI that uses an evaluation function to determine each move. The evaluation function is to be improved through use of a genetic algorithm in an attempt to increase the quality of the evaluation function, as well as making it specific to the board structure being played on.

II RELATED WORK

IAGO The logical place to start is with the AI IAGO (Rosenbloom 1982). This was the first dominant AI for Othello and one of the few to release information about its implementation. At the time of IAGO, computers were not allowed to enter in Othello tournaments, however, some tournaments were organized explicitly for Othello programs. In the Santa Cruz Open Machine Othello Tournament (1981) IAGO finished top of the 20 programs that entered, remaining undefeated.

The underlying algorithm used by IAGO is recursive Minimax with $\alpha - \beta$ pruning (this will be explained in depth later on). Minimax uses an evaluation function to analyse the strength of board positions and selects the optimal move (according to the evaluation function), assuming your opponent plays optimally as well. Since IAGO was designed to compete in tournaments where time constraints were imposed on it, a technique called Iterative Deepening was used in order to estimate the depth that IAGO can search to in the time allocated. Searching to a deeper depth will result in better performance, but success does not depend on searching to the deepest depth. In the North-Western tournament (1979) the strongest programs only searched to 4 or 5 ply, whereas the weakest programs searched to an average depth of 7 or 8, showing that the strength of a program comes from an accurate evaluation function and not from computational power (Rosenbloom 1982).

To increase the efficiency of IAGO's search, there are two considerations that are especially relevant to this project. The first is to simply increase the efficiency of the evaluation function through smart implementation; the second is to reduce the number of nodes that require evaluating through $\alpha - \beta$ pruning. A program implemented using Minimax can only play as intelligently as its evaluation function is accurate. Subsequently, a lot of effort was taken to find the optimal evaluation function where IAGO took a genetic approach, competing different versions against itself and selecting the best.

IAGO's Successor After IAGO, the next Othello program of note is BILL (Lee & Mahajan 1990). The development of BILL was inspired by IAGO and incorporates many of the techniques that IAGO uses. BILL introduces an algorithm based on Bayesian learning to determine a higher quality evaluation function. BILL incorporates both Iterative Deepening with Hash and Killer tables to order the nodes more efficiently, so that the number of cutoffs from $\alpha - \beta$ pruning are increased (Lee & Mahajan 1990). BILL also uses a large amount of pre-computed information to avoid calculating it at run time which would be extremely expensive. Since controlling the edge discs is critical, BILL uses a pre-computed edge table, storing information about all potential edge patterns. Unfortunately this approach is not particularly suitable to the adaptation that this project investigates. The main difference between BILL and IAGO is the use of Bayesian learning. The Bayesian learning approach determines an optimal quadratic combination for the evaluation function where the evaluation directly estimates the probability of winning (Lee & Mahajan 1990). The enhancements used in BILL reflect in performance with BILL consistently defeating IAGO.

Logistello Logistello is widely considered to be one of the strongest Othello programs ever. Almost all game related programs make use of an evaluation function to determine the quality of each available move and Logistello's evaluation function is entirely based upon statistical analysis, comprising of mobility measures and board patterns (Buro 1997). In total, Logistello built a game tree based off approximately classifying 3 million game positions by negamaxing game results. In 1997 Logistello played 6 games against World Champion Takeshi Murakami, winning all 6 games. After this match it was clear that computer programs had surpassed the ability of humans to play Othello and consequently research into Othello started to die down.

The Monte Carlo Approach All of the previously mentioned approaches are based off some variation of Minimax. Another technique that has been applied to perfect information games is the Monte Carlo Tree Search approach. The advantage of this approach is the lack of specialist knowledge required, consequently, it can often play poorly. When competing the Monte Carlo approach against a standard $\alpha - \beta$ technique the Monte Carlo process is consistently beaten (Nijssen 2007). Unless some major improvements are made on the Monte Carlo search process, it appears that it would continue to consistently lose to the $\alpha - \beta$ approach. However, when the Monte Carlo approach is substantialy improved, the rewards can be immense as shown by Google's AlphaGO (Lee et al. 2016).

III SOLUTION

A Software

The code for this project is written in C++. Aforementioned, the algorithms used require a substantial number of calculations due to the large number of leaf nodes in the game tree, therefore with the compiler optimisations for C++ it can be expected to process these calculations quicker than in other languages. It also provides access to tools to analyse performance such as Intel's Parallel Studio which helps find bottlenecks so that efficiency can be improved. Testing different AIs against each other requires a larger amount of computational power to simulate multiple games. Writing C++ code for this allows tests to be simulated on Durham University's supercomputer: Hamilton, enabling tests on a larger scale.

As the GUI is not the focus of this project, the OpenCV library was used because C++ has a suitable interface with this library.

The code is split up to increase its understandability by using Object Oriented Programming, implementing a Board, Player and Node class. The Board class facilitates all of the game mechanics, the Player class is responsible for selecting moves and the Node class is used to facilitate the generation of game trees. The game trees that produced by the algorithms are extremely large, so it is important to minimise the amount of data stored at each node.

B Methodology

For the development of this system a SCRUM approach will be used. The main stages in the SCRUM approach are *Plan, Sprint* and *Evaluate*. This lends itself to being an AGILE process which caters for reflecting and re-evaluating goals. Since this project revolves around the ability of Cassio to play intelligently, meaningful tests can only be carried out once an AI has been implemented. By splitting the work up into sprints, it is possible to focus on the stage of implementing an initial AI or improving a pre-existing algorithm during each sprint. Evaluating the algorithm after the sprint should give an insight as to where its strengths and weaknesses lie which are used as the focus point for planning the next sprint.

C Algorithms

C.1 Monte Carlo Tree Search

The Monte Carlo algorithm is divided up into four distinct phases: *Selection, Expansion, Simulation,* and *Backpropagation* which are repeated several times to produce a game tree (Archer

2007). Every node in the tree represents a board state. The root of the tree always represents the current board state and for each move available at any state there is an edge to a child node, where the edge represents the action taken to get to the new state (Figure 2). Each simulation updates the game tree and improves the understanding of the AI about the impact each available move has; after a specified number of simulations have been processed the algorithm determines which of the available moves it considers most desirable.

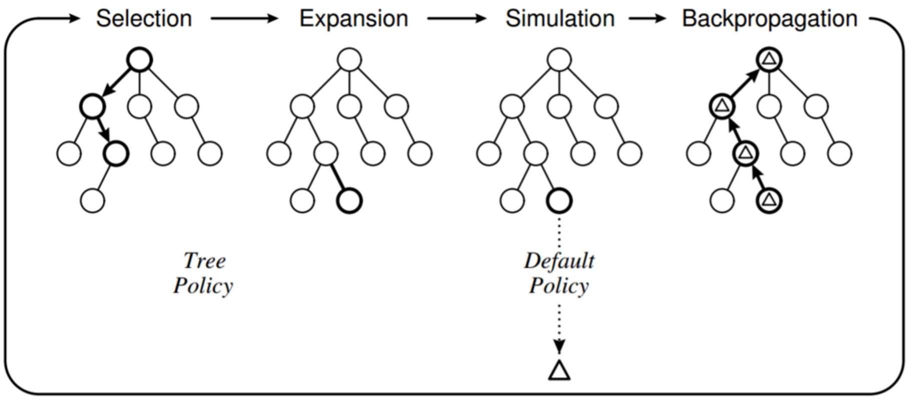


Figure 2: The four stages in the Monte Carlo Tree Search procedure (Browne et al. 2012).

Selection The first stage of the algorithm is selecting a child node to visit. There are several different methods for deciding which child node to select, but a very basic implementation is to select the node that has been visited the least so far. If this leads to a node that has been visited in a previous simulation, continue selecting children until an unvisited node is reached or the game ends.

Expansion Once an unvisited node is reached, this node is expanded. This means that it determines all possible moves from that board state and produces a child node for each available move.

Simulation After having expanded a node, legal moves are chosen randomly until the end of the game.

Backpropagation Once the game has finished and the winner has been determined, back-propagate up the game tree, updating win and visit counts on nodes in the selected path.

This four stage procedure is repeated hundreds or thousands of times, subject to any time constraints, with every simulation updating the selected nodes win and visit counts. At the end of the simulation the algorithm selects the child node of the root with the highest win rate as the next move. However, this is a very basic Monte Carlo algorithm and it can be done significantly better. The search problem at hand directly correlates to the Multi-Armed Bandit Problem (Auer et al. 2002), as there are a choice of available moves and it is not obvious which move has the highest pay-off. By reducing this problem to the Multi-Armed Bandit Problem, a much more effective selection policy can be employed.

Instead of selecting the child that has been selected the least, the idea of exploitation and exploration are balanced. Exploitation refers to focusing on moves that have high win rates whereas exploration emphasises exploring nodes that have not been selected many times. Through balancing out exploration and exploitation, the search is pruned such that stronger moves are examined more closely, but the supposed weaker moves are not discounted entirely.

When selecting a child node, instead of selecting the node with the highest win rate, the most *urgent* child node is selected, which combines the notion of exploitation and exploration. For any node, the most urgent child is the child node with the largest Upper Confidence Bound (UCB) (Browne et al. 2012). This UCB is given by:

$$UCB = \overline{X_j} + C\sqrt{\frac{2\ln n}{n_j}} \tag{1}$$

- $\overline{X_j}$ represents the win rate of node j (number of wins / number of visits)
- n represents how many times the parent node has been visited
- n_j represents how many times the choice of node j has been selected
- C is a tunable parameter, balancing *exploitation* ($\overline{X_j}$) vs *exploration* ($\sqrt{\frac{2\ln n}{n_j}}$)

The constituent parts of this equation are derived from a complex mathematical optimisation of the Multi Armed Bandit Problem, in which there is a gambler and lottery machines $S_0...S_n$ with expected returns of $X_0...X_n$ (unknown to the gambler). The gambler must select which machines to play and how many times to play each machine in order to maximise his return.

This derived equation balances out exploiting good moves against exploring moves that have not been selected as much. If a move has a high win rate, then the value of $\overline{X_j}$ is larger, increasing the UCB for that node and increasing the chances of that node being selected. Whereas, if a node has not been tested much, then the value of n_j is small, causing the exploration weighting to have a larger affect on the UCB value.

By applying this procedure logarithmic regret is experienced (Auer et al. 2002), which relates to the estimated loss from selecting non-optimal moves; this has been proven to be asymptotically optimal (Lai & Robbins 1985). When a node has n_j = 0, then the UCB for that node is set to ∞ to ensure that all nodes are selected at least once (Browne et al. 2012).

Monte Carlo is selected as a technique because of its unique nature. Unlike standard game playing techniques, the Monte Carlo approach uses no method of evaluating the strength of a given board position but instead calculates the relative strength of available moves through UCBs. Since devising an evaluation function for unpredictable boards becomes trickier and results in a less accurate function being used, a technique that does not rely on the accuracy of an evaluation function may give it an edge over techniques using a poor quality evaluation function. As the Monte Carlo technique discovers its own stategy without being told how to play, this algorithm should adapt extremely well to all board designs. The algorithm also does not need to evaluate any intermediate states, which can lead to the algorithm discovering unusual and unique strategies to win, if such strategies exist (Browne et al. 2012).

As more and more simulations are played, there comes a point where running more simulations has a negligible effect on the decision made by the AI. However, running more simulations has a cost in both time and memory which proved to be one of the major implementation issues. Running more simulations results in the production of a larger game tree and as a result if the program is designed for machines with small amounts of memory, then the number of simulations will be limited by this factor.

C.2 Minimax

Minimax is a technique applied to two player zero-sum games, where the two players are denoted as ‘Min’ and ‘Max’. The main idea of the approach is to minimise the maximum ‘damage‘ that the other player can inflict. It models the game from the Max players point of view and assumes that the Min player plays optimally to minimise the score that the Max player can achieve. To put a numerical value on the quality of a board state or to calculate the amount of potential damage your opponent could inflict, an evaluation function is used to analyse the board. This comprises of various board features such as: *Disc Stability, Mobility, X-Squares* and *Disc Count*.

Stability Stability is an extremely important concept in Othello. A disc is stable if the opponent cannot flip it for the rest of the game. Gaining control of stable edges is paramount to winning as they can be used to gain control of the central squares. Two extremely potent Othello AIs Iago (Rosenbloom 1982) and BILL (Lee & Mahajan 1990) use pre-computed edge tables to evaluate the edge positions as there are 3^8 potential edge configurations on a standard 8×8 board, although this project is not able to use this approach with the boards being randomly generated.

Mobility The Othello program IAGO has two forms of mobility: current mobility and potential mobility (Rosenbloom 1982). Current mobility refers to how many moves are immediately available to the player. This is one of the most important strategic concepts to play by as the player with the most mobility can dictate the direction of the game. If one player can limit the current mobility of their opponent whilst maintaining their own, it greatly increases the chances of their opponent being forced into making a poor move.

Potential mobility focuses on the idea of placing discs that will provide mobility in the future. The machine IAGO measures potential mobility by measuring the number of frontier discs (discs adjacent to empty squares) that the opponent has (Rosenbloom 1982). BILL uses an additional measure of mobility which is *sequence penalty*. The idea is that it is detrimental to have long sequences of discs in your own colour as they inhibit mobility, especially if they are frontier discs (Lee & Mahajan 1990).

X-Squares and Disc Count X-Squares are the squares diagonally adjacent to the corner squares. With the corner squares being the most powerful squares in the game, X-Squares are conversely the least desirable squares as they act as a stepping stone for the opponent to take the corner squares. It is important to try and force your opponent to take the X-Squares so that you are able to capture the corners. The disc count heuristic is considered only at the end of the game and simply refers to the number of discs in your colour.

The Minimax algorithm uses an evaluation function that incorporates a combination of these board heuristics to provide an accurate estimate about the quality of different board states. The algorithm produces a game tree of all possible lines of play to a specified depth, where all of the leaf nodes are assigned a numerical value based on this evaluation function. Figure 3 provides a graphical representation of a basic Minimax example (the non-leaf nodes are given a numerical value for clarity of explanation). The nodes represent board states and the number inside represents the evaluated result of that board state. This example looks at the game from the Max players point of view who is trying to maximise the number within the nodes, whereas the Min player is attempting to minimise the value that the Max player can obtain.

The numbers in red display which path would be taken by the Minimax algorithm in this example. If the Max player chose the central node *39*, then the minimising player would select *3*, as *3* is smaller than *24*. However, if the Max player selected the right node *19*, then the smallest value the Min player could select is *7*, which is larger than *3*. Whilst selecting the central node may result in a stronger board state than selecting the right node in a 1-ply lookahead, it is not optimal when looking further ahead in the game. In this example *24* would be the best outcome the Max player could hope to achieve, but Minimax assumes both players play optimally, so in this case the *7* is the best value that the Max player could possibly achieve.

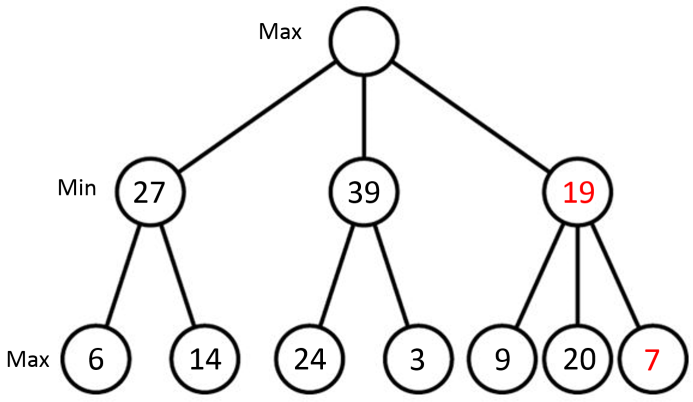


Figure 3: Visualisation of the MiniMax algorithm for a basic evaluation function, with the numbers in red representing the chosen path

Alpha-Beta Pruning

One of the issues with the Minimax approach is that all of the leaf nodes must be evaluated. When doing a 1 or 2 ply lookahead it is not a significant issue, but the tree grows at rate of $\theta(b^d)$, where b and d represent the braching factor and tree depth respectively. On a regular 8×8 board, Othello has an average branching factor of 9.9 meaning that searching to a high depth is computationally challenging (Rosenbloom 1982). A key aspect of the Minimax algorithm is searching to as large a depth as possible, but since the computation required to evaluate all leaf nodes at a larger depth grows exponentially it is imperative to optimise the search. The addition of $\alpha - \beta$ pruning aims to minimise the number of nodes that require evaluating. By remembering the best and worst moves found so far, the algorithm avoids traversing further down lines of play that are never going to be selected, with the procedure shown in Algorithm 1. The algorithm is initially called by alpha_beta(root, depth, -∞, +∞, TRUE). α represents the maximum value the Max player can be guaranteed to obtain and β represents the minimum value that the Min player can be assured of. The algorithm starts with $\alpha = -\infty$ and $\beta = +\infty$ so that both players begin with the worst possible score.

Algorithm 1 The Alpha-Beta Pruning algorithm

```
function alpha_beta(node, depth, $\alpha$, $\beta$, maximising_player)
if depth == 0 or game_finished then
    return node$\rightarrow$evaluate()
if maximising_player then
    find the value of the move the maximises the evaluation function
    v = -$\infty$
    for each child of node do
        v = max(v, alpha_beta(child, depth-1, $\alpha$, $\beta$, FALSE))
        $\alpha$ = max($\alpha$,v)
        if $\beta \leq \alpha$ then
            terminate the search down this path early
            break
    return v
else
    find the value of the move the minimises the evaluation function
    v = +$\infty$
    for each child of node do
        v = min(v, alpha_beta(child, depth-1, $\alpha$, $\beta$, TRUE))
        $\beta$ = min($\beta$, v)
        if $\beta \leq \alpha$ then
            terminate the search down this path early
            break
    return v
```

D The Evaluation Function

This section describes precisely how the evaluation function used by the Minimax with $\alpha - \beta$ pruning is designed and the implementation details of the constituent parts.

Stable discs

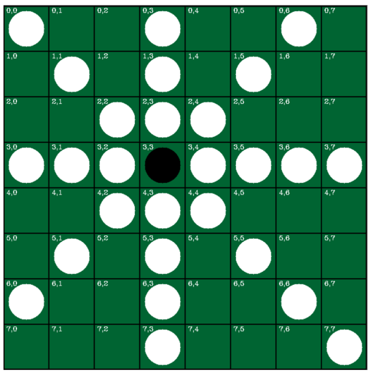
Stable discs are paramount to winning Othello, so it is extremely important to identify which discs are stable to allow the program to capture as many stable discs as possible. However, it is a non-trivial problem to determine whether or not a disc is stable. A disc is known to be stable if in each of the 4 directions ($\uparrow \nearrow \rightarrow \searrow$) at least one of the following conditions are met:

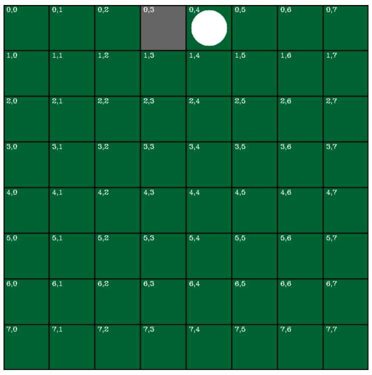
1. The lane is completely full of discs

2. The disc is adjacent to the edge of the board or an unplayable square

3. The disc is adjacent to a stable disc of the same colour

If these criteria are satisfied in all 4 directions then the disc is certainly stable. These three diagrams in Figure 4 illustrate various conditions under which discs can be considered stable. All three diagrams are positions that are unobtainable through standard game play but are explicitly made this way to show the conditions required for stability. Figure 4a shows how stability can


(a) Condition 1


(b) Condition 2

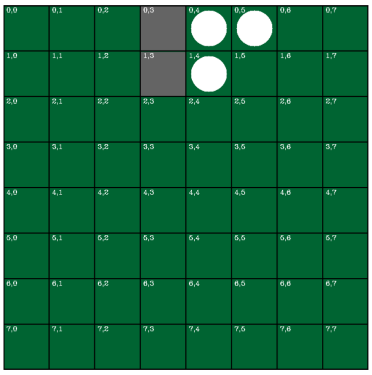

(c) Conditions 2 and 3

Figure 4: Conditions for disc stability

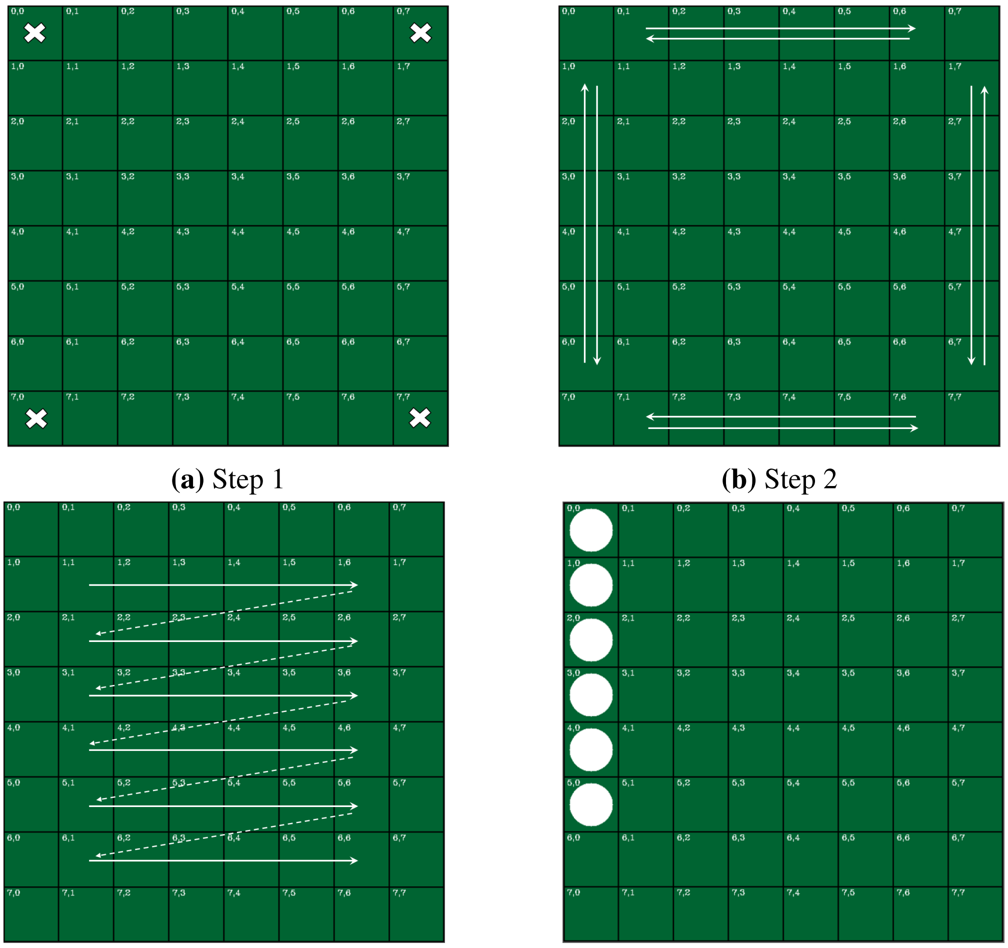
be obtained through condition 1 exclusively. In this example, the disc in location *(3,3)* is stable since all 4 of the lanes that it lies in are full of discs. This is because no further discs could ever be placed in a position that would result in the flipping of *(3,3)*.

Figure 4b shows how a disc can be stable with regards to the second condition. Condition two is satisfied in the three vertical directions (↑ ↗ ↘) since the disc in *(0,4)* is adjacent to the edge of the board in each of these three directions. The remaining direction to consider is the horizontal direction. Since the square *(0,3)* is unplayable then condition 2 is satisfied in this direction as well. Seeing that at least one of the three conditions is met in all 4 directions (condition two each time in this case), the disc in *(0,4)* can be considered stable.

Figure 4c shows a combination of conditions 2 and 3. The disc in *(0,4)* is stable for the same reasons as described for Figure 4b. As it is known that the disc in *(0,4)* is stable for White, this information can be used to discover more stable discs. When looking at disc *(0,5)* it is clear to see that it is adjacent to the top of the board, so condition two is satisfied in the three vertical directions, leaving just the horizontal direction to consider. Since the disc in *(0,4)* is stable, the disc in *(0,5)* is adjacent to a stable disc of the same colour in the horizontal direction. Subsequently condition three is now satisfied in the horizontal direction and the disc in *(0,5)* can now be considered stable. The disc in *(1,4)* is considered next. When looking in the horizontal direction and one of the diagonal directions (↘) it is clear to see that condition two is fulfilled due to adjacent unplayable squares. In the two remaining directions (↑ ↗) condition 3 is satisfied since from previous examination it is known that the discs in *(0,4)* and *(0,5)* are stable. Consequently the disc in *(1,4)* is stable since condition two or three is satisfied in all four directions.

However, it is worth noting that the order in which discs are considered is important. The disc in *(1,4)* is dependent on the discs in *(0,5)* and *(0,4)* being stable and the disc in *(0,5)* is dependent on the disc in *(0,4)* being stable. If the disc in *(1,4)* is considered before the other two then it could not be considered stable because condition three would not be satisfied. Due to these dependencies it is essential to examine discs in a logical order. It would be possible to guarantee finding all stable discs by repeatedly looping through the board several times and terminating after a loop discovered no more stable discs, but this would be inordinately expensive on the processing time, prohibiting searching to a sufficient depth in any reasonable amount of time.

The process, as shown in Figures 5a - 5c begins by checking to see if the corner squares are occupied, since if captured they are inherently stable due to condition 2 and it is especially rare for any stable discs to exist without the corners being taken first. The remaining edge squares are now examined. Each of these squares are checked twice, either from left to right and right to left or top to bottom and bottom to top. It is necessary to loop through each direction because of the dependency relationship between squares. It is common for a chain reaction of stability to run from one corner to another which can be missed if you only traverse in a single direction, as illustrated by Figure 5d. If these are examined from top to bottom then all of the white discs are considered stable. However, if examined from bottom to top, then the white disc in *(5,0)* is not stable, resulting in *(4,0)* not being stable and so on, such that only the disc in *(0,0)* would be considered stable. As a result, five stable edge discs are not identified, significantly decreasing performance. The interior discs are only examined once because they are not as vital to winning compared to the edge discs, and traversing through the interior squares multiple times would use up a significant amount of processing time that could be better spent identifying more important features.



(a) Step 1 **(b)** Step 2

(c) Step 3 **(d)** Ordering example

Figure 5: The Sequence of identifying stable discs

Overall, this process helps identify which discs are stable and separates them into three categories which all comprise part of the evaluation function: stable corners discs, stable edge discs adjacent to corner squares, and other stable edge discs.

Mobility

Mobility is regarded as one of the most important aspects to consider in order to play intelligently and comes in two forms: current mobility and potential mobility (Sannidhanam & Annamalai 2015), with Cassio's evaluation function using both forms. Current mobility is simple to calculate yet computationally demanding and refers to the number of moves immediately available to the active player. Potential mobility is used to consider mobility later on in the game. The manner in which potential mobility is calculated is inspired from IAGO (Rosenbloom 1982). It is determined by summing up the number of empty squares adjacent to discs of the opposing player. If a square is adjacent to two discs of the opposing player then it is counted twice (Rosenbloom 1982).

Figure 6 shows the direction each square needs to consider in order to minimise the time spent checking neighbouring squares for discs of the opposing player. For example, if location *(3,1)* is empty, it does not need to consider if any the squares in *(2,0), (3,0)* or *(4,0)* contain a disc of the opposing player as it is impossible to obtain a disc behind those squares in order to consider the empty square in *(3,1)* to be a potential move for future. The central 16 squares must examine all directions.

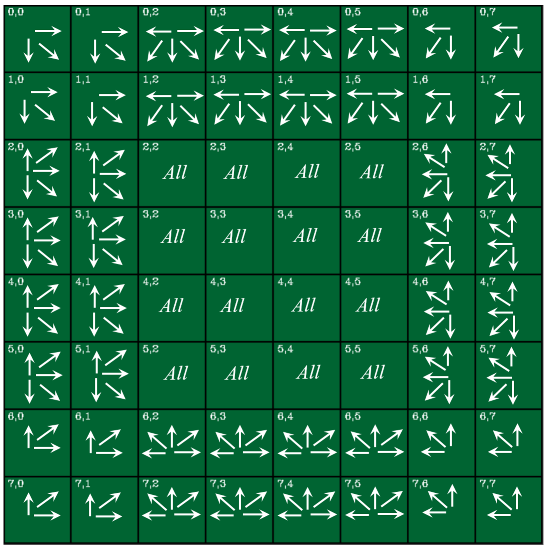


Figure 6: Potential mobility analysis, showing the direction each square must check for neighbouring discs of the opponent

X-Squares and Disc Count

X-Squares are the least desirable squares on the board and as such result in a penalty to whichever player has a disc in any of the X-Square locations (assuming the disc is unstable). The disc count heuristic has minimal importance until each player has very few moves left, at which point the weighting of disc count is significantly increased. Ultimately disc count is the determining factor in deciding the winner, so a brute force approach is taken when it is feasible to evaluate all remaining paths of play, where the only piece of information it considers are the final disc counts.

Producing the Numerical Value

A combination of the aforementioned heuristics are used to produce a numerical representation of any board layout from the perspective of a given player. One of the main issues this approach ran into was relativity. The evaluation function may identify a board position to be strong for player A, but even if that board position was stronger for player B, player A still considered it a favourable position. The solution to this is, for any board position the evaluation function calculates the difference between the numerical value for the respective players so that players are not necessarily competing to have what is considered a strong position, but instead a position stronger than their opponents.

E Genetic Programming

The strength of the Minimax approach relies on the quality of the evaluation function. Since devising an accurate evaluation function by hand is difficult, humans let computers compete against themselves to identify what results in a high quality evaluation function. To search for the optimal parameter set, a genetic approach has been taken, applying the concept of *Survival of the Fittest*. (Eskin & Siegel 1999). Algorithm 2 describes the procedure followed by the genetic algorithm.

Algorithm 2 The Genetic algorithm

```
1: Input: Population Size N, Number of Generations T
2: G ← Generate_Random_Population(N)
3: for $i = 0$ to $T$ do
4:     G ← Round_Robin_Tournament()
5:     G ← Select_Next_Generation()
6:     G ← Crossover (5% chance of mutation)
7:     G ← Transfer_Generations()
```

A genetic approach is used because it allows Cassio to devise its own evaluation function in a manner that caters to random boards. After the generation of a random board, applying this genetic procedure and having the AIs compete against themselves should result in discovering which aspects of the evaluation function are more influential to winning. Logistello improved and perfected its performance by competing against itself over 100,000 times (Buro 1997). Often after several generations, the 'gene pool' (distribution of evaluation functions) can become extremely small, with little variation. This causes any further generations to have a minimal effect on improving the evaluation function. The 5% mutation probability attempts to increase the variety of evaluation functions within the population.

Testing

Several testing stages occurred through out the project. Following the Agile development methodology provides opportunity for testing to be performed at the end of each sprint, examining two main areas: Validation (is the right system being built?) and Verification (is the system being built right?). The program was verified several times throughout the development of the system, with Object-Orientation providing a suitable environment to test various aspects of the system individually and rapidly identify bugs within the system. Validation occurred at the end of the project whereby a series of Unit Tests are performed, ensuring that all units of the program provide the correct functionality. The system was then manually integrated and tested in its entirety to ensure that all the components worked together. The final stage of validation was User Acceptance Testing, where various different users trialled the system, ensuring that the system worked as specified and required.

IV RESULTS

A Test Setups

There are a large number of factors that can be investigated with this project and it is impractical to test all combinations. The performance of the Monte Carlo approach is investigated by looking at the effect of varying the number of simulations, through a round robin tournament. The effectiveness of the genetic algorithm is evaluated through another round robin tournament, investigating whether running the genetic algorithm for a larger number of generations results in stronger performance. After obtaining an evaluation function from this process, the Minimax approach competes against itself whilst searching to different search depths. Finally, the Monte Carlo algorithm competes against the Minimax algorithm on both standard and randomly generated boards to investigate the influence of randomly generated boards on the separate techniques.

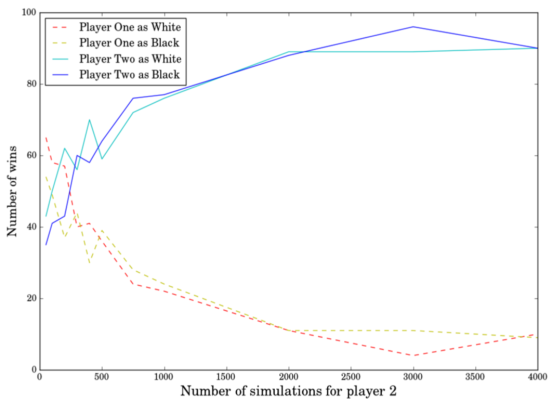
B Monte Carlo

The first test scenario consisted of Monte Carlo programs competing against one another. One of the Monte Carlo programs (A) fixes the number of simulations used (*200*) and the opposing Monte Carlo program (B) uses a varying number of simulations. Figure 7a shows how the number of wins varies as the programs compete against each other. Since the Monte Carlo approach is probabilistic, several games are carried out to decrease the variance in the results, with each program playing a total of 100 games (50 as each colour). These values for the number of simulations and the total number of games played are dictated by the time constraints of performing the tests; given more time these values could have been increased.

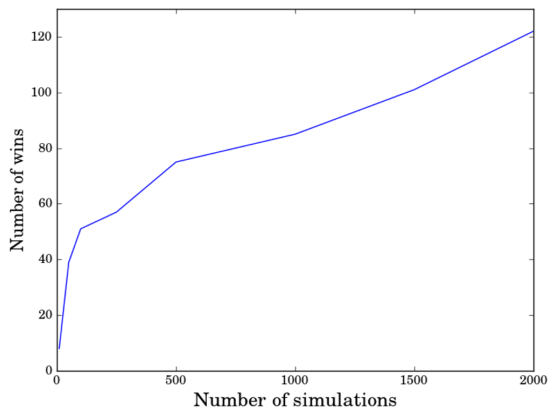
Figure 7a clearly shows that as the number of simulations for program B increases, program B starts winning a higher proportion of the games, showing how the level of performance of a Monte Carlo program directly correlates to the number of simulations performed. After approximately 2000 simulations, increasing the number of simulations for program B has a reduced affect on the frequency with which B beats A, with the point at which simulations can be terminated being dependent on the strength of the opposing player.

The second experiment incorporated the Monte Carlo programs competing against each other in a round robin format. A population of 8 Monte Carlo programs is used with each program using a different number of simulations. Each program plays 140 games each, competing 20 times against each other program. Figure 7b shows how many wins each program achieved across the tournament. Ideally a larger population size would have been used, but it was necessary to repeat games several times to reduce the variation, so the size of the population was again limited by time constraints since the runtime is exponential in the population size.

The results shown in Figure 7b are as expected. Programs that use a larger number of simulations win a much larger proportions of their games, with the strongest program winning 87% of its matches compared to the weakest program winning only 6%. The relationship is not perfectly linear which is likely to be a result of only performing 140 games per result; testing a larger sample of games per result would be expected to produce a set of results with less variance and a stronger linear relationship.



(a) Varying the number of simulations



(b) Number of wins achieved by each AI after a round robin tournament

Figure 7: Monte Carlo results when competing against itself

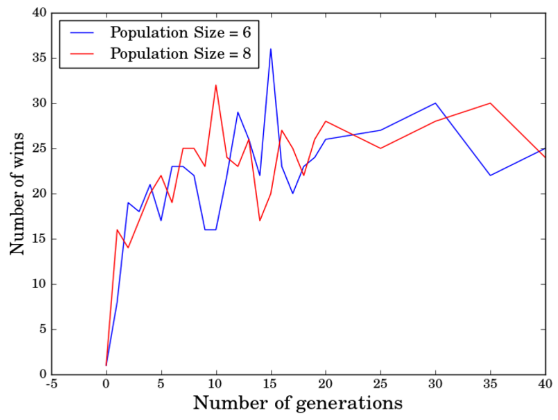
C Genetic Algorithm Performance

Since the optimal relative weightings of the evaluation function are unknown, the genetic algorithm is used before any Minimax testing took place so that the Minimax tests can be performed with a strong evaluation function. Algorithm 3 describes the test procedure to evaluate the effectiveness of the genetic approach, with the results of the approach taken shown in Figure 8.

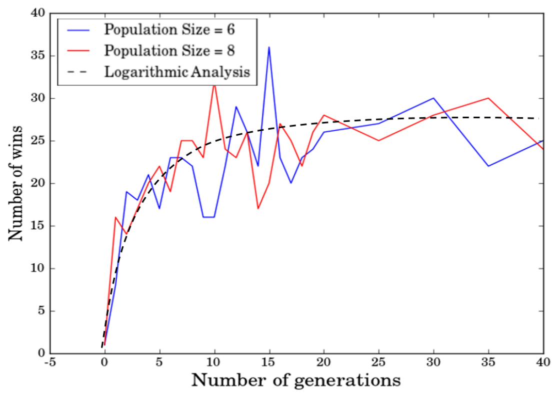
Algorithm 3 Genetic Algorithm test procedure

```
1: Input: Population Size $N$, Number of Generations $G$
2: for $i = 0$ to $G$ do
3:     Produce evaluation function $E_i$ from a genetic algorithm on $G$ generations
4: for $i = 0$ to $G$ do
5:     for $j = i$ to $G$ do
6:         Compete $E_i$ against $E_j$
```

The results shown in Figure 8a show a large increase in the proportion of wins from 0 generations to 5 generations, which is most likely due to the poor quality evaluation functions that are quickly removed from the earlier generations. However, there is not a strong correlation between the number of generations and the number of wins. To experience a significant improvement after running for several generations it is necessary to use a substantially larger population size with a far wider ‘gene pool’. There is no distinguishable difference between using a population size of 6 or 8, but these matches are only carried out between programs using the same population size. When looking at the graph universally, a logarithmic relationship can be seen as illustrated in Figure 8b. It is possible that the effectiveness of the genetic algorithm is proportional to the logarithm of the number of generations used.



(a) The number of wins achieved by each generation



(b) Analysis of results

Figure 8: Genetic algorithm results from performing a round robin tournament

D Performance of Minimax

The first aspect of the Minimax algorithm to investigate is the effect on performance of searching to a deeper depth. This is tested on depths of 1 to 6 with each program using the same evaluation function as described previously. The results are as expected, with the program searching to a deeper depth winning every time. The difference in computation time from searching to a depth of 1 does not vary much compared to searching to a depth of 5. Only after searching to a depth of 6 or greater are significant time issues experienced, such that searching to a depth of 7 is infeasible.

The next stage of tests consisted of comparing Monte Carlo based programs against Minimax based programs. Testing this on the regular 8×8 board results in the Minimax based program winning 100% of the matches when searching to a depth of 5. Even when the Minimax program only searches to a depth of 2 or 3 it still consistently beats the Monte Carlo based program unless impractically large numbers of simulations are used (*50,000+*). The total time taken to select all moves for Monte Carlo program is in excess of 350 seconds, compared to Minimax taking less than 1 second. This is when performing on non-optimized implementations, where the room for improvement on the Minimax approach is larger than for the Monte Carlo approach. This further highlights the time efficiency of the Minimax technique. After analysing the mobility levels over 100 games using a Minimax search depth of 5, the Minimax program has an average of 10.2 moves available per turn compared to 3.3 of the Monte Carlo program, regardless of the number of simulations used, highlighting the importance of mobility throughout the game.

E Performance on Randomly Generated Boards

The final area of interest is to investigate how the relative performance levels of Monte Carlo and Minimax based algorithms compare when competing on randomly generated boards. When random boards are created with the maximum seeding (20% chance per square), the Minimax programs continues to consistently beat the Monte Carlo based programs until as before, the

Minimax search depth is decreased to just 2 or 3 and the number of simulations for the Monte Carlo increased massively, suffering from the same time issues as before. For any seed used, the Minimax approach continued to outperform the Monte Carlo approach, showing how the use of random boards appears to have a minimal affect on game-play.

V EVALUATION

Suitability of Approach

Before beginning to implement any of the software, it was necessary to decide on design aspects such as the programming language and development methodology. The decision to use C++ as the programming language has proven to be an excellent choice as it provided the luxury of running the more CPU intensive tests (Monte Carlo based methods) on Hamilton as well as facilitating an Object-Oriented approach. This provided a method for data abstraction whilst also making it far simpler to extend the project further. Running tests on Hamilton was especially applicable to the Monte Carlo based programs due to the probabilistic nature of this technique, making it necessary to simulate several games to reduce uncertainty in the results.

The SCRUM approach taken has both advantages and disadvantages. The nature of sprint cycles allowed for rapid implementation of various sections. However, this often resulted in a lack of emphasis on documentation and a larger proportion of bugs produced, especially related to program memory. To resolve these issues, sprints are taken with an aim to solve purely memory issues when necessary, alongside frequent testing to identify any significant bugs. Weekly supervisor meetings aided in keeping the project focused and ensuring that the project was heading the correct direction.

Strengths and limitations of the project

After looking at the results from competing programs against each other it is easy to see where the strengths lie. When given an accurate evaluation function, the Minimax technique with the addition of $\alpha - \beta$ pruning showed extreme dominance and performed well. Not only did this approach play highly intelligently, but it also regularly selected moves within a single second when searching to a depth of 5. This approach also performed surprisingly well on the randomly generated boards. On reflection, the heuristics used are largely transferable between board designs, as mobility is equally as important and in most cases having control of stable edge discs continues to result in winning. The genetic algorithm shows promise, however it suffers from an exponential running time with regards to the size of the population and it is believed that a much larger population size is necessary for significant improvements. Thereafter, it is infeasible to perform the genetic algorithm at run time, but instead needs to be a method of pre-processing. Therefore it is not particularly applicable for use on spontaneous random boards, however it would be suitable to analyse a (large) sample of random boards and produce evaluation functions specific to each board.

Whilst not a large focus of the project, the GUI that has been implemented works well, providing a clear, aesthetically pleasing interpretation of the board, with the addition of showing users the moves available to them, with a simple interface for users to select moves.

The main weaknesses of the implemented algorithms lie with the Monte Carlo algorithm. Not only is its performance substantially weaker than that of Minimax, but it also uses significantly more memory. For it to play at a reasonably intelligent level the runtime is inordinately large, averaging over 10 seconds per move for programs using 50,000+ simulations.

VI CONCLUSIONS

In summary, this project compares the relative strength of two different techniques to compete at Othello on both standard and pseudo-randomly generated boards. The Minimax based method competes at a strong level, playing intelligently whereas the Monte Carlo based technique struggles to stay competitive with the Minimax approach on all standard boards and pseudo-random boards alike, with randomly generated boards not having a significant affect on game-play. The genetic algorithm used displays potential but would benefit from running on a substantially larger scale in order to produce a greater performance enhancement.

Whilst the project produces a high quality AI, there is still a large scope for further work with regards to improving the existing implementations and also taking the project in a new direction. Despite the Minimax approach having the strongest performance there are many ways in which it could be improved. Iterative Deepening is a technique that allows the AI to spend time more efficiently is reached. One variation of this technique known as Trappy Iterative Deepening attempts to identify and set traps that other players could fall in (Gordon & Reda 2006).

There is also the opportunity for different adaptations of Othello to be investigated. The mechanics and GUI have been implemented using hexagonal tiles instead of square tiles. Given more time an AI could have been produced to play specifically on the hexagonal grid, which also could have the additional modification of randomly generated hexagonal grids to see if unplayable squares have a larger impact on hexagonal boards than square boards.

With regards to the genetic algorithm there are several modifications that could be trialled. A thorough selection process could be employed such as a roulette wheel approach (Zhang et al. 2012) instead of selecting the population members with the greatest success, in an attempt to jump out of any local maxima. Due to the large number of tunable variables that the population members consist of, there is a huge scope for experimentation with crossover methods. As is the nature with genetic algorithms, it is often very hard to predict the relative success of different approaches and it is only possible to compare approaches after implementing them, but it leaves open endless opportunities for future work.

References

Archer, R. (2007), Analysis of Monte Carlo Techniques in Othello, PhD thesis, BS Thesis, The University of Western Australia.

Auer, P., Cesa-Bianchi, N. & Fischer, P. (2002), 'Finite-time analysis of the multiarmed bandit problem', *Machine learning* **47**(2-3), 235–256.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. & Colton, S. (2012), 'A survey of monte carlo tree search methods', *IEEE Transactions on Computational Intelligence and AI in Games* **4**(1), 1–43.

Buro, M. (1997), 'An evaluation function for othello based on statistics', *Technical Report 31, NEC Research Institute* .

Eskin, E. & Siegel, E. (1999), ‘Genetic programming applied to othello: introducing students to machine learning research’, *ACM SIGCSE Bulletin* **31**(1), 242–246.

Gordon, V. S. & Reda, A. (2006), Trappy minimax-using iterative deepening to identify and set traps in two-player games, *in* ‘Computational Intelligence and Games, 2006 IEEE Symposium on’, IEEE, pp. 205–210.

Lai, T. L. & Robbins, H. (1985), ‘Asymptotically efficient adaptive allocation rules’, *Advances in applied mathematics* **6**(1), 4–22.

Lee, C.-S., Wang, M.-H., Yen, S.-J., Wei, T.-H., Wu, I., Chou, P.-C., Chou, C.-H., Wang, M.-W., Yang, T.-H. et al. (2016), ‘Human vs. computer go: Review and prospect’, *arXiv preprint arXiv:1606.02032* .

Lee, K.-F. & Mahajan, S. (1990), ‘The development of a world class othello program’, *Artificial Intelligence* **43**(1), 21–36.

LLC, M. (1999), ‘MS Windows NT kernel description’.
URL: *http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm*

Nijssen, J. (2007), ‘Playing othello using monte carlo’, *Strategies* pp. 1–9.

Rosenbloom, P. S. (1982), ‘A world-championship-level othello program’, *Artificial Intelligence* **19**(3), 279–320.

Sannidhanam, V. & Annamalai, M. (2015), ‘An analysis of heuristics in othello’.

Zhang, L., Chang, H. & Xu, R. (2012), Equal-width partitioning roulette wheel selection in genetic algorithm, *in* ‘Technologies and Applications of Artificial Intelligence (TAAI), 2012 Conference on’, IEEE, pp. 62–67.