Contents

Analysis

Overview

Type of project (solution/investigation)

Solution

Description

I am designing an AI for the board game Othello (sometimes called Reversi) that a user can play against. The solution will feature a visual representation of the Othello board and allow a user to interact with it to input their moves.

Rules

After choosing Othello as the game I will develop an AI for, the first key step will be to ensure I have a full and accurate understanding of the game's rules. I am using the instruction manual that comes with the physical board game as my source for the rules. A digital copy of this can be found at (Othello Instruction Sheet, 2023).

Overview

- Othello is played on an 8x8 board and consists of 2 players with one playing as white and one playing as black.
- The winner of the game is the player who has the majority of their colour discs face up on the board at the end of the game.
- The game ends when it is no longer possible for either player to move.
- Standard Othello notation, as defined by the instruction manual, uses the letters A-H for the column and 1-8 for the row with A1 being the top-left square (*Figure 1)*. I will describe positions using the appropriate location e.g. B4 followed by the letter indicating the colour of the tile. W or B. For example, *Figure 1* shows the position C4B, D4W, D5B, E4B, E5W.

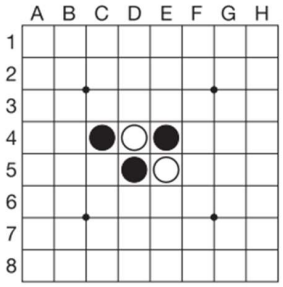


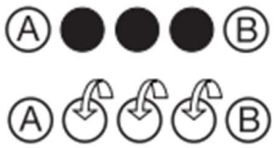
Figure 1 (Othello Instruction Sheet, 2023)

- If a player has a legal move, it must be played, and they cannot forfeit their turn.

Legal Moves

- Every move must consist of '**outflanking'** the opponent's disc(s) and flipping the outflanked discs to the player's colour.

- **Outflank** – place a disc on the board so that the opponent's row (or rows) of discs is bordered at each end by the player's disc colour.
  e.g.
  As white, placing disc B outflanks the 3 black discs, flipping them to white.



- A row of discs that can be outflanked can exist as a straight line in any direction – horizontally, vertically or diagonally.
  e.g.
  Playing a white disc in C7 (*Figure 2*) will result in 3 black rows being flipped, one in each direction (*Figure 3*).

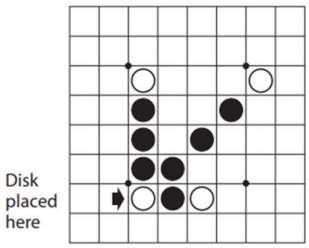


FIGURE 2

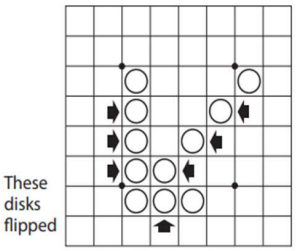


FIGURE 3

Setup

- The game is always started in the same position with 4 discs on the board, 2 white and 2 black (*Figure 4*). These discs are located as D4W, D5B, E4B, E5W.

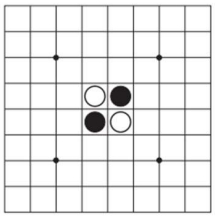


Figure 4

- Black will always move first with the only legal opening moves being C4, D3, E6 and F5.

From these rules, I can create some initial objectives for my project.

The primary objective can be decomposed into sub-problems that need to be solved.

Objective 1: Recreate the game of Othello in a virtual application.

- 1.1: Have an 8x8 board that can contain black or white tiles.
- 1.2: Display the winner of the match when the game is finished.
- 1.3: Flip the correct discs in accordance with the rules when a player makes their move.
- 1.4: Have a button to allow the player to restart the match.
- 1.5: Setup the board in the correct position (D4W, D5B, E4B, E5W) every time the match starts.

Analysis of Other Systems

In order to set myself objectives that my final product will have to meet, I will research similar systems to identify flaws and benefits which I can improve on or adapt into my product. I will also show these systems to my client in order to gather their opinions and see if/how they coincide with or differ from my thoughts and analysis. This will allow me to then create detailed objectives for the project to work towards. These can be evaluated thoroughly once the project is complete.

The Physical Game

My client and I decided to play 3 rounds of Othello in order to assess our skill levels and to analyse what makes the game enjoyable and what could be improved by playing the game physically instead of virtually.

System 1 Othello Board Game (Anhlee, 2023)

My first observation when playing the game was that it was challenging to easily have an accurate overview of the position at any given time. Initially, I would keep track of the number of tiles that were my colour and the number that were my opponent's colour, however, I found this too challenging especially when I was also trying to work out the best move. This meant I stopped paying attention to the precise number of white and black discs and estimated every time. I feel like counting discs is not a skill aspect to the game and is simply time consuming. This means that, as the information is available anyway, the game would be made more entertaining by including a live counter to show the number of white discs and black discs on the board at any one time.

Objective 2: Have a system to demonstrate to the user the number of white and black discs face up on the board in a given position that updates every time a move is made.

In game 2, I made an illegal move by placing my disc in a position that was not valid and began to flip the tiles I incorrectly thought I was allowed to flip. This resulted in confusion when trying to return the board to the state it was in before, after I recognised my error. In order to prevent this from happening, valid moves will need to be checked by the game and my client suggested it may help the player to visually display all valid moves. I decided to research other online board games to compare ways they display valid moves and evaluate other features of them.

I asked my client what online board games they play or have heard of in order to research.

Q: What online board games do you play or know about?

Answer: I'm a big fan of chess and mainly play on LiChess. I also used to play draughts against my friends in school on a website I think was just called draughts.org.

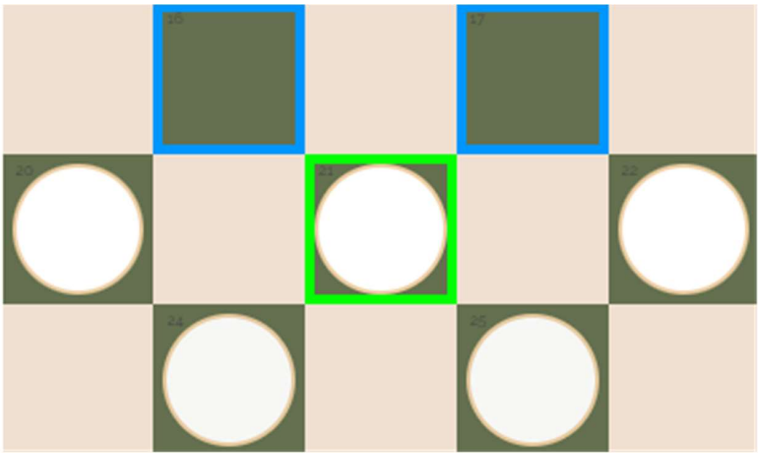
Q: Do you know the website address for LiChess?

Answer: I think that's also just LiChess.org

Draughts.org

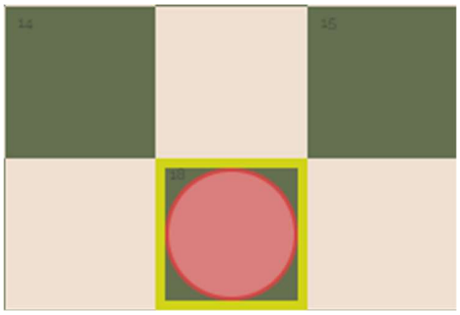
Draughts.org uses colour coded outlines on playing squares to convey information to the player when playing against the computer or other opponent.

- When a player clicks on a piece they wish to move, the playing square is outlined green to show it is selected and valid moves are outlined blue.



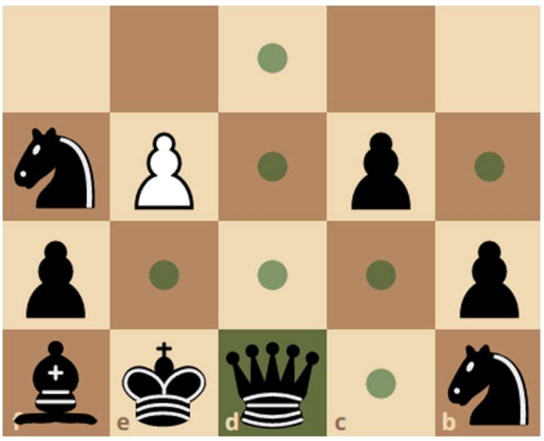
System 2 Draughts.org (Draughts.org, 2023)

- The most recent opposition move is outlined in yellow in order to indicate to the player what has just happened. This system would be very helpful in Othello as lots of pieces can change colour simultaneously so it may be hard for the player to understand where a disc was played.



LiChess.org

LiChess.org uses small green dots in the centre of each valid playing square to indicate where a selected piece can move. I would have assumed it may be quite hard to notice on an 8x8 grid, however, chess is also played on an 8x8 board, and it works for LiChess. Differences between the game of chess and Othello might contribute towards why this system works well for chess. In Othello, a valid move could be anywhere on the board as long as it neighbours an already played piece. This includes moves being on opposite sides of the board which may be hard to notice on initial inspection of the board. However, in chess, the dots only appear after choosing which piece to move and there is a limited number of spaces that piece can move to, constrained by the type of piece it is. Whether or not this system works practically in Othello needs to be tested.



System 3 LiChess Move Options (LiChess, 2023)

When comparing the Draughts.org method of indicating valid moves and the LiChess method, I feel like the Draughts method is easier to notice, especially when there are 64 available spaces to play on the Othello board. However, the LiChess method is much more subtle and looks slightly sleeker. I have decided to consult my client to get their opinion on what they prefer.

Client Interview 1

Question: What do you think about the look of this method (Draughts.org) of highlighting valid moves?

Answer: I think the available options are clear, however, I feel like the shape of a square is not perfect and could be better. It doesn't look very nice and the squares on squares doesn't really work.

Q: When you say squares on squares doesn't work, what do you mean?

Answer: It just doesn't seem to fit. The style of the game looks quite neat and tidy especially with the neat number system for squares and the way the playing pieces have a border to help them stand out. I think the squares for the board is necessary but adding additional squares looks harsh and like there hasn't been much effort put in to make it look nice.

Q: Do you have any thoughts about the colouring of the highlighted squares?

Answer: The neon colouring is quite garish and not very in-keeping with the style of the board that uses a sort of darker colour palette. Also, it probably could be overwhelming when there are a large number of valid moves.

Analysis of responses: The client seconds my opinion that the method of showing valid moves is very obvious and hard to miss describing it as being 'clear'. They initially showed a dislike of the squares used and, upon further questioning, revealed that they think the 'harsh' nature of the squares doesn't fit with the sleek and refined style of the board. I had not previously considered the shape with which the boxes are highlighted an issue, however, after my attention had been drawn to it by the client, I understand and recognise potential problems. My initial dislike was with the neon colouring, so I decided to ask my client their specific thoughts on that. They described the colouring as 'garish' and highlighted the contrast between this garish colour and the 'darker' colours used elsewhere on the board. It may be useful for me to research colour palette creation to help decide which colours I use. The final comment from the client was that the method may be 'overwhelming' when there are many moves. I agree with this so I will either need to find a way to reduce this either by using a different method or potentially by simply changing the colour palette.

Question: What do you think about the look of this method (LiChess.org) of highlighting valid moves?

Answer: The way of doing it looks a lot nicer and I think although it's simpler than the other way (Draughts.org) it looks better.

Question: Do you feel the method (LiChess.org) is still clear enough when there could be many valid moves?

Answer: I think it would be okay. It might be harder to notice which moves are valid because if the game window is small and then has to be divided into 64 boxes which all would need to then have these small dots inside of them, the dots would be very small overall and hard to notice.

Question: Can you think of a solution to this problem or another way of showing moves that might work better?

Answer: The colour could probably be made bright enough that it would be obvious but there is a risk of it being too garish and looking like the Draughts way of doing it. I can't think of any other ways of showing moves as everything I have seen is pretty similar to one of the two shown.

Analysis of responses: The client seems more approving of Method 2 (LiChess.org) in terms of how it looks compared to Method 1 (Draughts.org). He describes it as being 'a lot nicer' and says it looks 'better'. I agree with this, however, I had concerns about how obvious the indicators would be so decided to prompt a response from my client about it. The client says the dots may be 'hard to notice' which I agree with so wanted to ask the client for any ideas as to how to resolve this. Making the colour 'bright enough' was the suggestion, however, the client cautioned not to make it 'too garish' so I will likely get feedback from them before finalising the colours I use for any part of the project.

After considering the feedback of my client, I have decided to use a method similar to the Draughts method for highlighting the most recently played move as I feel it stands out, however, I will adjust the colour as I feel it looks too garish and will adjust the shape as the client is not happy with the square. I agree and feel it looks too boxy so more rounded corners may help.

Objective 3: Have a system to visually indicate to the player the most recently placed tile by the computer by outlining the square the computer played on.

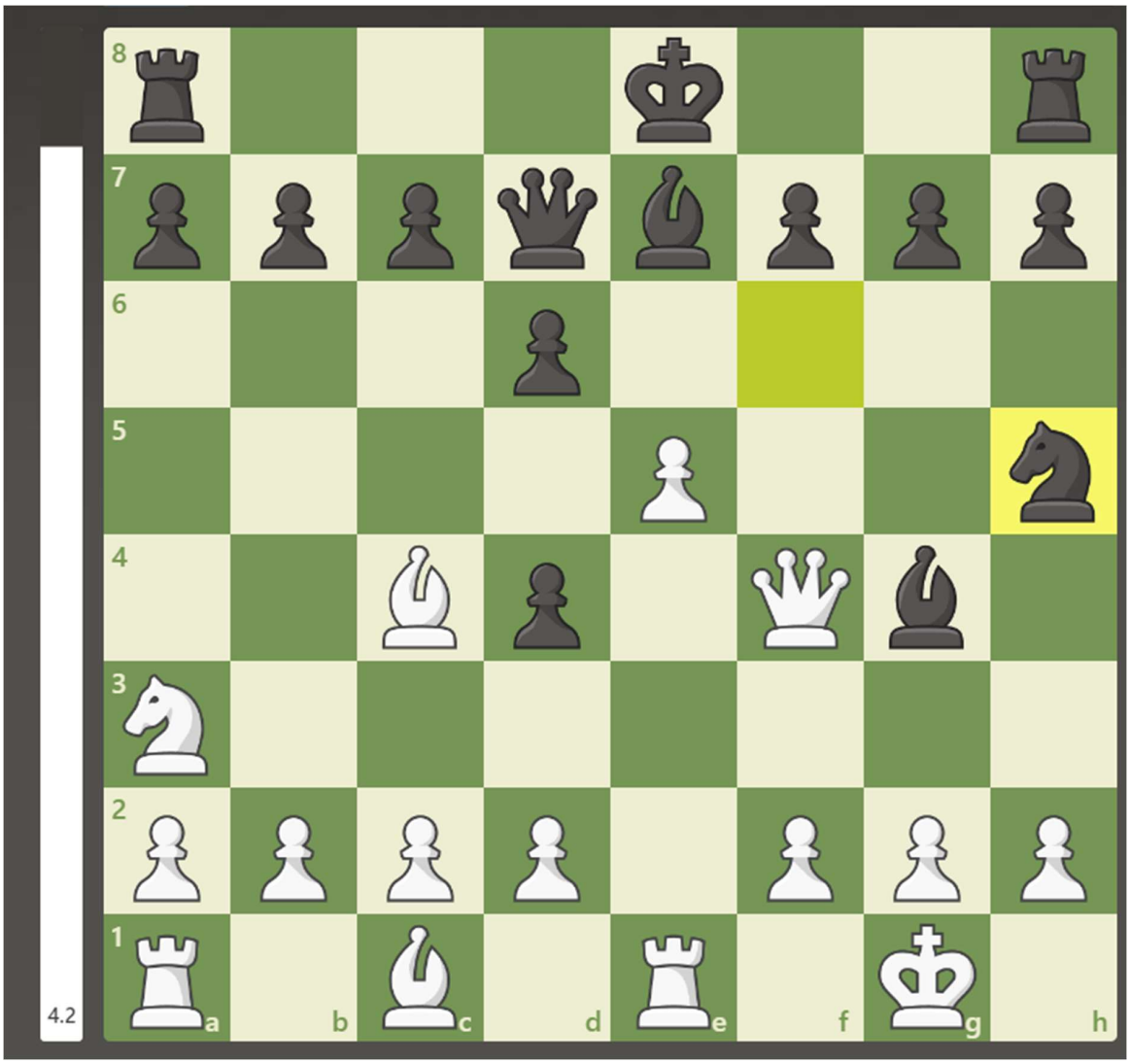
For showing which moves are valid, I will use a method similar to the LiChess method as I personally find this approach to indicating moves more subtle and more refined especially when there will be a large number of valid moves. As my client noticed, the colouring of the dot will be important as it will need to contrast the likely green background I will use for the board. I also think the size and transparency of the dot can be adjusted to make it more or less subtle based upon client feedback when I show them designs.

Objective 4: Have a system to indicate to the player which moves are valid and legal in the current board state by adding a semi-transparent dot in the centre of the playing square.

As the product needs to include an AI system, I will have to research other systems which implement a competitive AI into a board game. My client did not know which systems may help so I decided to research possible systems to investigate. I discovered Chess.com which uses the Stockfish engine for evaluating positions which is widely regarded as the strongest engine in the world.

Chess.com

The website Chess.com is the most popular chess website with over 100 million users and a UI which has been refined since 2005. One of the features that chess engines use and Chess.com implements is a numerical system to evaluate a position. In standard chess: a pawn is assigned one point, a knight or bishop three, a rook five and a queen nine. A position is analysed by an AI and given a numerical evaluation that corresponds with these values. The system uses positive numbers to show an advantage for white and negative to show an advantage for black. 0 indicates a drawing position. For example, in the image below, the evaluation is +4.2 for white. If you count the pieces, both players are equal on points, however, the engine looks at possible future moves for white and sees an order of moves to allow white to be winning by a significant amount.



System 4 Chess.com (Stockfish Analysis, 2023)

I would like to use a similar numerical method to evaluate Othello positions where each white disc is assigned a value of 1 point and each black disc is assigned a value of -1 point. For example, if the game is ended and neither player has a valid move and white has 20 tiles and black has 10, the position should evaluate to +10. However, if black has one valid move left, after which the game ends with black having 20 and white having only 15, the evaluation should be -5 regardless of the number of tiles face up in the current position. Furthermore, evaluation algorithms reward 'strong' positions of pieces with a greater positive evaluation and punish 'weak' positions. For example, in chess, a knight could be worth 4 points in the evaluation of the position if it is positioned in the centre of the board but only 2 if it is in a corner.

In Othello, for example, pieces in the corners of the board can never be flipped so should be worth more than pieces in the centre of the board that could be easily flipped.

Objective 5: Write an evaluation function to provide a numerical evaluation of a given Othello position.

Another piece of functionality that Chess.com implements is the ability to view previous board states by clicking arrow keys to step backwards to previous positions.

System 4 Chess.com

The single chevron allows the user to step backwards and forwards through the match one move at a time and the chevron with the bar line means the user can skip backwards to the start of the match or forwards to the current position. This allows the user to review the match after it has finished so that they can determine where they could have played better moves against the AI. I would like to implement this system for the benefit of the user experience. However, one issue of the Chess.com implementation is that these arrow keys can be challenging to press due to their small size on the screen. I aim to resolve this by not only making the chevrons larger, but also allowing the user to use the arrow keys on their keyboard in order to step forward and back through the game.

Objective 6: Allow the user to click on arrows or use the arrow keys to step forwards and backwards through the match they are currently playing or reviewing.

At this point, I felt satisfied with my research of other systems and decided to interview my client, showing them my decisions, and asking for any further ideas and suggestions they may have regarding the design of my product.

Client Interview 2

Question: Do you understand the description of how the evaluation function operates and are you satisfied with it as a method of indicating who is winning in a given position?

Answer: I think I get it. My one concern is that, I think, a position is evaluated and given a value seemingly based on what it could become in the future. Does this not mean that a position could only be good if one specific move is made by the player?

Question: Yes, the function assumes perfect play by both the computer and the player. It aims to give the true evaluation of a position. Do you think this is a problem for you and is there anything you would like to see implemented to improve the system?

Answer: I don't think it's an issue as long as, when reviewing the game, you can see what the best move would have been.

Analysis of responses: Given that I am very familiar with the evaluation method used by Chess.com I had not considered the confusion that its assumption of perfect play by the player could cause. However, I think it is relatively simple to understand for a user so it could be beneficial to add a method of informing the player of how the AI function works.

Objective 7: Allow the user to request information explaining how the AI evaluation function works so they can better understand the feedback they are given.

My client also raised the issue that I have not yet determined a way to tell the player what the best move would have been in a situation. This shouldn't be told to the user while they are in the process of playing to maintain the integrity of the game, however, if they would like to review the match it should become an option. After considering ways of implementing this, I have determined a method of achieving this that can be broken down into a step-by-step solution.

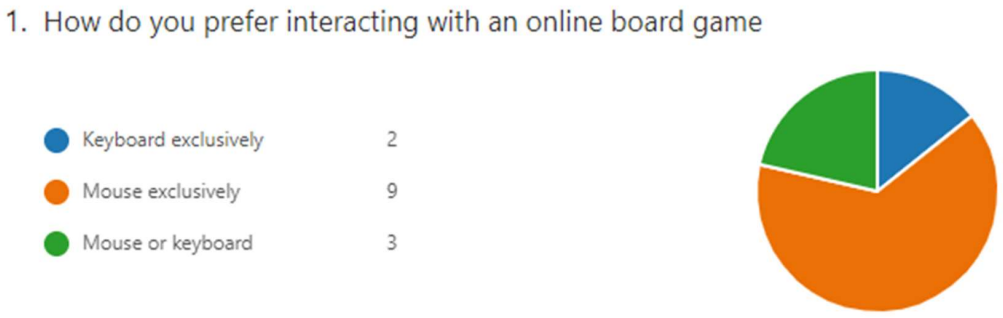
1. Before the game is launched, allow the user to choose whether they wish to play a match or review a match.
2. If they are playing a match, when the match is finished, all the moves of the match will be uploaded to a database along with the date and time it is played. The primary key associated with that match will be printed to the user.
3. If the user chooses to review a match, they will be shown a list of all the previous matches and when they were played and be asked to input the key of the match they wish to review.
4. After inputting the key, the game played will be loaded and they can step through the match seeing the evaluation of the position and the best move in each position.

Objective 8: Allow the user to review matches played.

- 8.1: Before launching the game, give the user the choice between playing a match or reviewing one.
- 8.2: Once a match is complete, store it in a database along with the time it was played.
- 8.3: Display to the user a list of all previous matches played that are available for review.
- 8.4: Allow the user to input the key of a match they wish to review and load that match into the review system.
- 8.5: Write a match review system where the user can step through the match, seeing the evaluation and best move in each position.

Survey

In order to develop some further objectives as to how the product will operate, I created a survey which I gave to 14 people that I know play online board games. I asked 2 questions on the survey: one about the method of interaction with the game and one about the skill level of the AI.



Question 1

From the response, we can see that the majority (64%) of participants prefer using exclusively a mouse to interact with online board games while only 14% prefer keyboard. Of the 2 participants who prefer keyboard exclusively, both are fans of the game connect 4. This is a game where arrow keys are more logical for input due to only 7 possible locations and both respondents agree that when there are 64 possible locations – like in Othello – mouse input would be better. Thus, I will aim to use mouse input for the player.

Objective 9: Allow the player to select the location of their move by clicking on the appropriate square with their mouse/trackpad/touch screen.

Question 2 has the possible responses of

- One you can beat easily.
- One that you sometimes beat but sometimes lose to.
- One that you regularly lose to but sometimes beat.
- All of the above.

These responses were designed to correspond to a conventional Easy, Medium or Hard difficulty (in that order) seen in many AI settings for board games.

Question 2

The responses indicate that there are a variety of opinions about AI skill levels. The modal response is 'all of the above' and the relatively equal split between the other 3 options indicate that most people want the choice of which difficulty but lots

of people have differing opinions about difficulty with a slight trend in favour of a medium difficulty. As a result, I will program an AI that can perform at different skill levels in order to provide a choice to the player as to what difficulty their match will be.

Objective 10: Have an AI that can perform at different difficulties.

The difficulty for a given match should be selected by the user before the match begins. By adding this functionality to the AI, it becomes necessary to alter the GUI at the beginning of the match in order to allow the player to select the difficulty.

Objective 11: Allow the user to select the difficulty AI they wish to play against at the beginning of the match.

Summary of objectives

Area	Number	Objective	Justification
GUI	1.1	Have an 8x8 board that can contain black or white tiles.	Allows for the user to play Othello on the standard board size and means that the game is easily recognisable due to the standard colouring.
	1.2	Display the winner of the match when the game is finished.	Provide the user with a visual gratification if they have beaten the AI.
	1.4	Have an option to allow the player to restart the match.	Improves user experience as they do not have to rerun the program if they want to restart and can simply enter that they wish to replay while remaining in the application.
	2	Have a system to demonstrate to the user the number of white and black discs face up on the board in each position that updates every time a move is made.	Allows for an improved user experience as they are not forced to recount the number of tiles on the board after a move is made and can have a continuous overview of the state of the match.
	3	Have a system to visually indicate to the player the most recently placed tile by the computer by outlining the square the computer played on.	Allows the user to keep track of the moves the AI is making as these may be made quickly in some situations and large numbers of tiles may flip, making it hard for the user to follow the game.
	4	Have a system to indicate to the player which moves are valid and legal in the current board state by adding a semi-transparent dot in the centre of the playing square.	Reduces the complexity for users playing the game as they can quickly evaluate which moves would be beneficial for them to play and they are less likely to overlook a possible move.
	8.1	Before launching the game, give the user the choice between playing a match or reviewing one.	It means that both aspects of the project can be contained within the same program.

	8.3	Display to the user a list of all previous matches played that are available for review.	Means that the user can see the appropriate key for a match they played at a certain time or day if they haven't saved the key.
	11	Allow the user to select the difficulty AI they wish to play against at the beginning of the match.	Improves user experience as they are presented with an obvious choice as to what difficulty they play and can make that decision themselves.
Functionality	1.3	Flip the correct discs in accordance with the rules when a player makes their move.	The alternative is to force the user to click which pieces need to be flipped manually which may lead to mistakes and would take a long time. It is therefore logical to write an algorithm to do it instead.
	1.5	Setup the board in the correct position (D4W, D5B, E4B, E5W) every time the match starts.	Necessary according to the rules of the game.
	6	Allow the user to click on arrows or use the arrow keys to step forwards and backwards through the match they are currently playing or reviewing.	Allows the user to think about what has happened previously in the match and review their moves on the go.
	8.2	Once a match is complete, store it in a database along with the time it was played.	Needed for persistent data storage so that it can be reviewed another time.
	8.4	Allow the user to input the key of a match they wish to review and load that match into the review system.	Allows the user to choose which match to review based on factors such as if they won or lost and the difficulty of the AI.
	8.5	Write a match review system where the user can step through the match, seeing the evaluation and best move in each position.	Means the user can determine where they made errors in their playing style and where they could have played a better move.
	9	Allow the player to select the location of their move by clicking on the appropriate square with their mouse / trackpad/ touch screen.	Enables convenient interaction with the game board for the user.
AI	5	Write an evaluation function to provide a numerical evaluation of a given Othello position.	Allows the AI to choose the best move and allows the user to review a given position.
	7	Allow the user to request information explaining how the AI evaluation function works so they can better understand the feedback they are given.	Explains how to interpret the AI evaluation metric so that they can use it to benefit their playstyle more.
	10	Have an AI that can perform at different difficulties. The	Allows for a more fun user experience as they can choose a difficulty at which they

		difficulty for a given match will be selected by the user before the match begins.	will win easily or struggle depending on their preference.

Software decisions

I have decided to program this project primarily using the language Python. Due to the complex nature of the calculations that will be needed for objective 5 (Write an evaluation function to provide a numerical evaluation of a given Othello position) I am choosing to use a language I am more familiar with as it will make programming more complicated maths that will potentially be necessary such as matrix multiplication significantly easier.

Additionally, python contains many libraries that can be imported to provide extra functionality. In order to meet the **GUI** objectives, I will use the PyGame library as this provides the ability to create a game window which the board and tiles can be drawn to for the user to see and interact with. It also allows for functionality such as objective 9 (Allow the player to select the location of their move by clicking on the appropriate square with their mouse/trackpad/touch screen) by providing systems to detect mouse clicks and track the location of the cursor on the screen.

In order to meet functionality objective 8.2 (Once a match is complete, store it in a database along with the time it was played) and other databasing requirements, a library will need to be used to store the match info. I have decided to use SQL for the database and the library MySQL to interact with it. Most Python implementations of SQL are very similar just with different syntax and MySQL is widely used and allows for persistent storage of information.

Develop some code and testing

As part of my initial development, I will need to choose what colours I use for the playing squares and for the valid move indicator (Objective 3). This is important as my client and I have identified that the colouring of these will be important as otherwise it will be hard to see which moves are valid. I generated 2 colour palettes I liked and showed them to my client for feedback.

Palette 1

Palette 2

Question: Which of these 2 colour palettes do you prefer the look of?

Answer: I think palette 2 because I am not a fan of the reddish colour in the middle or the purple colour on the right in palette 1. I do think the middle pinkish colour in palette 2 may be a bit dark.

I will use a variation of palette 2 as a result of this feedback. Also, because of my client's concerns about the middle colour in palette 2 being too dark, I have decided to write a small program to compare colours and transparency values of different circles in order to decide the exact RGB value of the playing pieces and of the valid move indicator.

I will use the green colour for the board squares, the white and black for the respective playing piece colours, the pink for the semi-transparent move indicator (Objective 4) and the yellow to highlight the most recently played move (Objective 3).

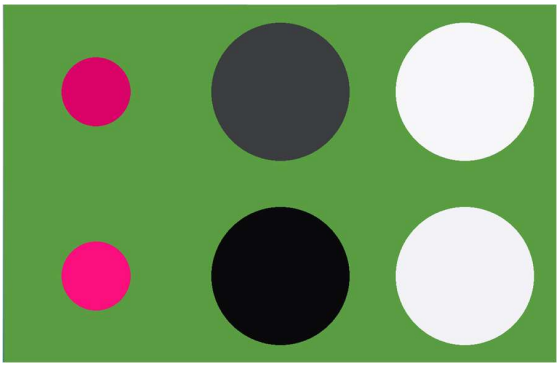


Figure 1 One run of the colour shade comparison code

After showing my client a few different combinations of shades of colour, we decided on the 3 colours we will use in the final project.

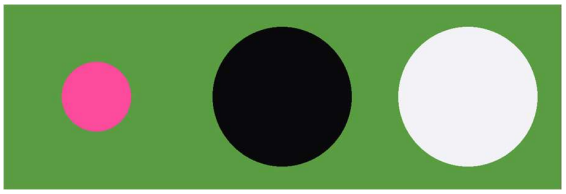


Figure 2 Final colour choices

I then decided to prototype the other visuals to represent the playing board.

Current objective: 1.1 - Have an 8x8 board that can contain black or white tiles.

I will be primarily using Python to create the product as it contains lots of the required functionality I will need for my code and allows for an easy mixture between object-oriented and procedural code, allowing me to pick the best algorithmic solution to each problem without limitations. I will use PyGame for the GUI as it contains all the necessary functionality such as getting the location of mouse clicks and displaying text.

The first issue I recognised was that I needed to define the width and height of the PyGame window. I decided it would be overall beneficial to the user to allow for the size of the window to be adjusted when the game is started. This size is defined as *height* and *width* in my program. The board is now drawn slightly offset from the edges of the screen (by the values *wBuffer* and hBuffer) to allow space for information such as the current score and the position evaluation to be added.

```
wBuffer = 50     # gap between side of the window and start of board (width-wise)
hBuffer = 50     # gap between top of the window and side of board (height-wise)
hBoard = height - 2* hBuffer
wBoard = width - 2* wBuffer
pygame.draw.rect(screen,ASPARAGUS,(wBuffer,hBuffer, wBoard,hBoard))
```

Figure 3 Drawing board background offset from window sides

Additionally, an 8x8 grid is now generated using white bar lines to maximise contrast between the black window background and the board. This can be easily changed in the future if the colour scheme is changed further.

```
for lineNum in range(9):
    pygame.draw.line(screen,FINW,(wBuffer,hBuffer + lineNum*(hBoard/8)),
                     (wBuffer+wBoard,hBuffer + lineNum*(hBoard/8)),max(1,hBoard//200))
    pygame.draw.line(screen,FINW,(wBuffer + lineNum*(wBoard/8),hBuffer),
                     (wBuffer + lineNum*(wBoard/8),hBuffer+hBoard),max(1,wBoard//200))
```

Figure 4 Method for drawing board bar lines

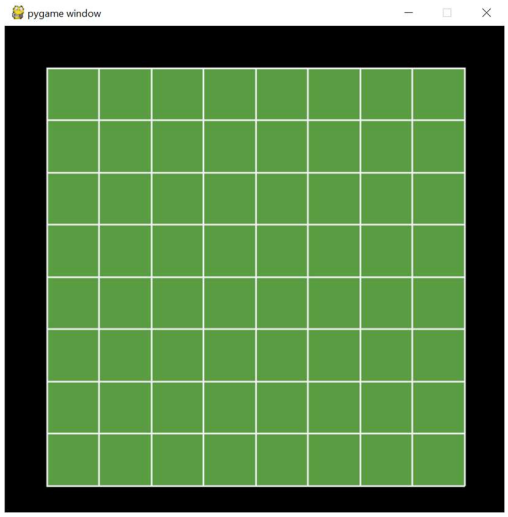


Figure 5 Board drawn in a 600x600 window

The next step will be to allow tile pieces to add functionality to draw the black or white tiles on the board. I have achieved this using a procedure which will take in 2 values, a number from 1-64 indicating the tile number and a second integer from 1-3 indicating the type of circle to draw.

- 1 – White tile
- 2 – Black tile
- 3 – Semi-transparent pink tile valid move indicator

```
def drawPiece(position, colour):
    """
    colour 1 is white
    colour 2 is black
    colour 3 is pink
    """
    x, y = positions[position]
    x = int(x)
    y = int(y)
    if colour == 1:
        pygame.draw.circle(screen, FINW, (x, y), int(wInterval * 0.35))
    elif colour == 2:
        pygame.draw.circle(screen, FINB, (x, y), int(wInterval * 0.35))
    else:
        # Slightly smaller semi-transparent circle to be used for valid move indicator
        pygame.draw.circle(surface, FINP, (x, y), int(wInterval * 0.15))
        screen.blit(surface, (0, 0))
    pygame.display.update()
```

Figure 6 drawPiece procedure

After using PyGame to detect mouse clicks and to determine the click location, some Modulo arithmetic can be used to calculate the corresponding tile pressed as a number from 1-64.

```
if event.type == pygame.MOUSEBUTTONDOWN:

    if board.collidepoint(pygame.mouse.get_pos()):
        boardPos = (wBuffer, hBuffer)
        relativePos = (pygame.mouse.get_pos()[0] - boardPos[0], pygame.mouse.get_pos()[1] - boardPos[1])
        tileNumber = (relativePos[0] // wInterval) + (relativePos[1] // hInterval) * 8
        drawPiece(int(tileNumber), tileType)
```

Figure 7 Mouse press detection and tile calculation

A written small sample program demonstrates all of the above features, demonstrating that the concept for the GUI works.

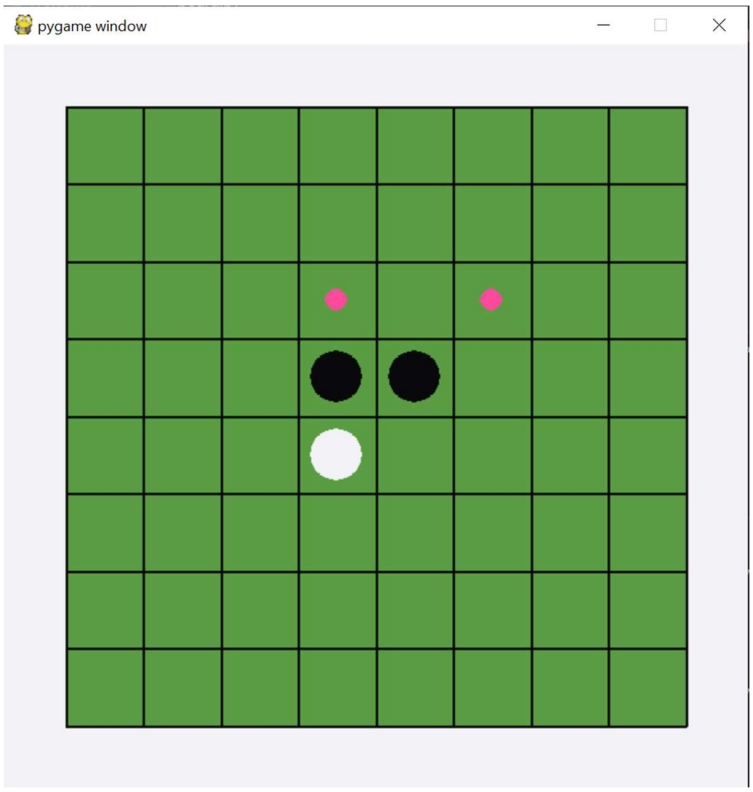


Figure 8 Game Window

Why this project is of A-Level Standard

In this Section I have detailed a few examples of why my project meets the A Level Standard.

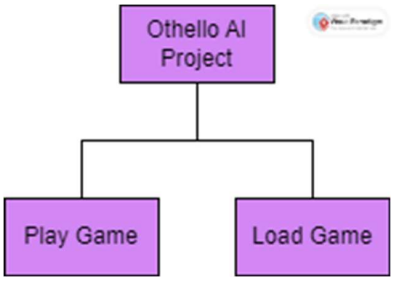
Group	Model	Algorithms	How And Why Used
A	Hash tables	Hashing	Custom hashing algorithm used to generate a unique integer value to represent a board state to allow for faster lookup in the Lookup Table and storage in the database.
A	Complex user-defined algorithms	Recursive algorithm	Recursive checkValid function used to check if a move is going to be valid.

A	Complex user-defined algorithms	Recursive algorithm	Recursive flipIfValid function used to flip pieces in a given direction when a move has been played.
A	Complex user-defined algorithms	Recursive algorithm for optimisation	Recursive Minimax algorithm to optimise moves played using alpha-beta pruning and a lookup table to maximise efficiency at large depths.
A	Complex user-defined algorithms	Evaluation algorithm	Complex algorithm to generate and count the number of 'stable' discs on a playing board and combine this with a weighting system to assign a numerical evaluation to a board state.
A	Parameterised SQL	Game data storage and retrieval	The *insert_data* subroutine takes the details about the match just played and parameterises them to be inserted into the table. This is to allow all matches played by the user to be stored in the database to be fetched for reviewing. The *get_data* function parameterises the user input of the key of the match they wish to review to fetch it's data from the table and the data is then used to review the match in the review system.
B	Multi-dimensional arrays		The *stableGenerater* algorithm uses multi-dimensional arrays to store the directions that need to be checked for each corner to validate that the piece is stable.
B	Text Files	Writing and reading from files	Used to persistently store the lookup table which is used in the *MiniMax* algorithm to save re-evaluation of a position if it has already been checked before. Also used in other locations such as in *OthelloDatabase.py* in subroutines such as *execute_read_query* to save any errors to a log file for evaluation by the user.
B	Simple client-server model	Calling Web Service APIs and parsing JSON/XML to service a simple client server model	User questions are submitted to a web-server-hosted Large Language Model using API calls and JSON responses are parsed and displayed to the user to aid them with their understanding of the Othello game and the AI system implemented into it.

Documented Design

Overview

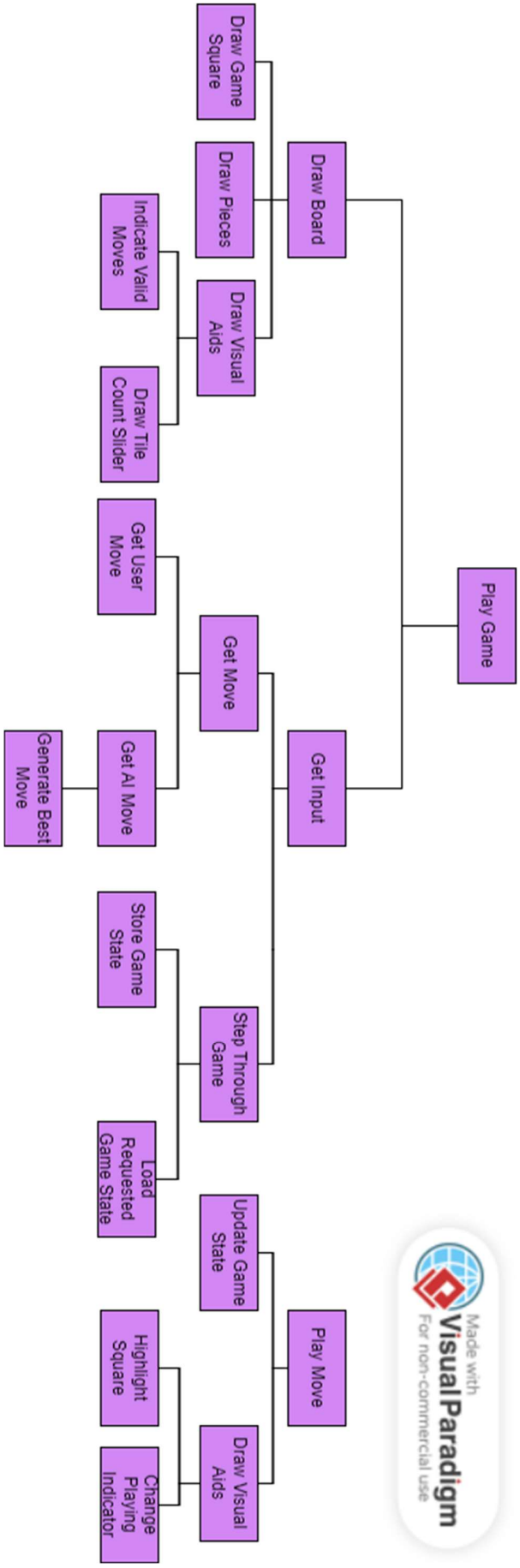
I will first create a high-level overview in the form of a systems diagram that will demonstrate how the project will operate. I can then break down individual components of this system by developing code for them and prototyping. Due to space restrictions on Visual Paradigm Online – the tool I am using to develop these diagrams – I have split the systems overview into separate diagrams.



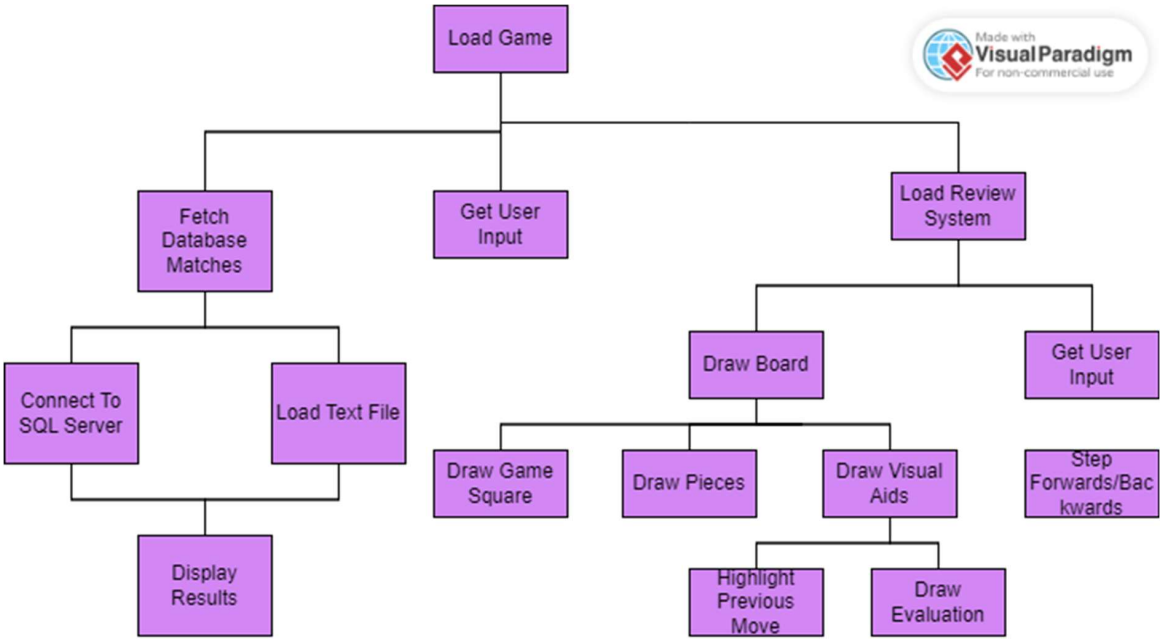
On the game starting, the user will be faced with a choice between loading a game and playing a game. If they pick to **Play Game**, they will need to setup the game first.

As part of the setup, board size can be decided for ease of use, the user should be able to choose the AI difficulty they want to play against in line with the objectives and choose if they wish to save the game for later review. If they wish to, the system should choose where to save the game either a SQL database or text file depending on if connection to the server can be established. Once the setup is complete, the main system of playing the game can begin.

This is broken down into the system for drawing the board, getting input from the user due to them making a move or choosing to click arrow keys to backtrack through the game. Finally, this move needs to be made and corresponding operations need to be run.



If the user chooses to load a game, the load game system needs to be run. This will be a separate system to playing the game, however, some features such as aspects of drawing the board will be the same as playing the game.



Drawing The Board

GUI Design

I asked my client what sort of design they would like for **Objective 2**[1]. They suggested that a slider visually indicating who has the higher number of pieces is likely a beneficial solution. I decided to also add the number of pieces of each type to the slider and designed the GUI in a visual design software. I produced two different possible example positions, one where white has more pieces and one where black has more pieces. The bar length should be proportional to the percentage of placed tiles that belong to that player.

Figure 9 A situation where white has 15/18 tiles.

Figure 10 A situation where white has 8/22 tiles.

This calculation for bar length is done by calculating the width of the whole slider, a value I've set at half the width of the entire board and multiplying it by the percentage of the tiles that are the corresponding colour. For example, for white, it is (*whiteTiles / totalTiles) * sliderWidth.*

[1] Objective 2 - Have a system to demonstrate to the user the number of white and black discs face up on the board in each position that updates every time a move is made.

Indicate Valid Moves

In order to indicate the valid moves on the board (**Objective 4**[2]), they must first be generated. I will begin by implementing the brute force method by checking every tile on the square and checking if it is valid. First, I will need to implement a method of checking if an individual tile is a valid location to play a piece and then do this method for the entirety of the board.

In order to generate a method of looking along the board in a straight line, I will use a vector comprised of the change in x and the change in y on the board. I can generate these using Nested For loops and have to ignore the case where the change in x and the change in y both equal zero.

```
FOR INT dx = -1 TO 1
        FOR INT dy = -1 TO 1
                IF dx != 0 OR dy != 0 THEN
                        # Check valid move in the direction: dx,dy
                ENDIF
        NEXT dy
NEXT dx
```

Figure 11 Pseudocode to generate direction vectors.

To check if the move is valid (**Objective 1.3**[3]), I will need a function that takes in 4 pieces of information:

- The current tile we are looking at (a tuple with the x and y coordinate).
- The direction in which we are looking (a tuple containing the x direction and the y direction).
- The colour of who is playing (1 for white, 2 for black).
- The current state of the board (A 2D array of all rows and columns).

The method I have chosen to use will start checking on squares where a piece has been played and move along the line, checking if all of the pieces are the opposite colour until it reaches a blank space, indicating that this blank space is a valid location to play. This is done recursively.

```
FUNCTION check_valid(location, direction, colour, board)
        curX <- location[0] + direction[0]
        curY <- location[1] + direction[1]
        # Check edge of board hasn't been reached
        IF curX < 0 OR curX == BOARDSIZE or curY < 0 or curY == BOARDSIZE THEN
                RETURN FALSE, (-1,-1)
        ELSE IF board[curY][curX] == colour THEN
                RETURN FALSE, (-1, -1)
        ELSE IF board[curY][curX] == (3-colour) THEN
                RETURN check_valid((curX,curY),direction,colour,board)
        ELSE
                # Prevent false positive when the colour is immediately next to a blank space
                IF board[location[1]][location[0]] == 3-colour THEN
                        RETURN TRUE, (curX,curY)
                ELSE
                        RETURN FALSE, (-1,-1)
                ENDIF
        ENDIF
END FUNCTION
```

Figure 12 Pseudocode to check if a given direction from a tile is a valid location to play.

[2] Objective 4 - Have a system to indicate to the player which moves are valid and legal in the current board state by adding a semi-transparent dot in the centre of the playing square.

[3] Objective 1.3 - Flip the correct discs in accordance with the rules when a player makes their move.

The method to flip the tiles when the move is made is similar in nature, however, returns an altered board instead of the location.

```
FUNCTION flip_if_valid(location,direction,colour,board):
        curX <- location[0] + direction[0]
        curY <- location[1] + direction[1]
        IF board[curY][curX] == 0 OR curX < 0 OR curX == BOARDSIZE or curY < 0 or curY == BOARDSIZE THEN
                RETURN FALSE, board
        ELSE IF board[curY][curX] == colour THEN
                RETURN TRUE, board
        ELSE
                change, board = flip_if_valid((curX,curY),direction,colour,board)
                IF change == TRUE THEN
                        board[curY][curX] = colour
                END IF
                RETURN change, board
        END IF
END FUNCTION
```

Figure 13 Pseudocode to flip the tiles on the board.

Backtrack Through Match

In order to meet **Objective 6**[4], I will need to allow the user to step forwards and backwards through the past states of the board. I plan on implementing a system where users can both use the arrow keys or use their mouse to click on arrows to step backwards and forwards through the match in accordance with the objective.

My intended method is going to be to store every board state played in the match so far in an array list. This is beneficial as the array of boards can be used with the databasing system to store the match at the end while also being used as a storage system you can step backwards and forwards through. Instead of the *drawBoard* function just drawing the board state that is passed to it, it will also take an integer parameter *pastDifference* specifying how many moves in the past the system should draw. A *pastDifference* of 0 indicates that the player wants to see the most recent board state but a value of 3 would mean the player wishes to see 3 moves ago. The *pastDifference* integer is constrained between 0 and the number of moves played in the game in total.

In order for the user to interact with this value, I have implemented a GUI feature displaying arrows.

The single chevron pointing left < increases *pastDifference* by 1 whilst the single chevron pointing right > decreases it by one. The double chevron pointing left << skips back to the beginning of the match by setting *pastDifference* to be equal to the number of moves played so far. Conversely, the double chevron pointing right >> skips back to the most recent board-state by setting *pastDifference* to 0. Additionally, pressing the left arrow key is the same as clicking < and the right arrow key is similarly the same as clicking >.

Minimax Algorithm

In order to develop the AI (Objective 5[5]), I have decided to implement the Minimax algorithm. The Minimax algorithm works using numeric values of each board state produced. It assumes one player is trying to maximise the overall score and one player is trying to minimise their score. In my game, white will try to maximise their score and black will try to minimise their

[4] Objective 6 - Allow the user to click on arrows or use the arrow keys to step forwards and backwards through the match they are currently playing or reviewing.

[5] Objective 5 - Write an evaluation function to provide a numerical evaluation of a given Othello position.

score. This is because as discussed, the chess.com evaluation this is based upon used positive numbers for white advantage and negative for black. Minimax algorithm assumes perfect play and looks at all possible future board states. For example, in a black position where there are 3 possible moves, it will generate a numeric evaluation of the board produced by the 3 moves and pick the one with the lowest evaluation (minimise). It does this by, for each move it could make, analysing the moves white can make in response. It will look at all the possible response moves white could make from each new position and assumes white will pick the one with the highest evaluation (maximise). It can repeat this process a set number of times to look x number of moves into the future where the depth of the function is x. This means that the max depth can be altered easily to adjust difficulty and Minimax is a very effective algorithm when numeric evaluations of board states are produced.

```
FUNCTION minimax(board, depth, maximisingPlayer)
        IF depth == 0 THEN
                RETURN heuristic value of board
        ELSE IF maximisingPlayer == TRUE
                eval <- -∞
                # each valid move in a given position
                FOR each move:
                        eval <- MAXIMUM( value, minimax(board.play(move),depth-1,FALSE)
                # board.play(move) will return the board state after the move is made
                NEXT move
                RETURN eval
        ELSE
                value <- ∞
                FOR each move:
                        eval <- MINIMUM( value, minimax(board.play(move),depth-1,TRUE)
                NEXT move
                RETURN eval
END FUNCTION
```

Figure 14 Pseudocode for Minimax algorithm.

Alpha-Beta Pruning

Alpha-Beta pruning is a method to optimise the Minimax algorithm by cutting off some of the possible moves when there is no chance it will be picked. For example, at one position you may be looking to choose the minimum value. The options already checked may be position A with evaluation +3. The program then moves on to attempt to evaluate position B by using the maximiser function at this location. If the algorithm finds a move from position B where the board becomes evaluation +5, you know that the evaluation of position B is going to be at least +5 so you can disregard position B as position A is guaranteed to be a lower evaluation at only +3. This cuts off unnecessary search time and optimises the algorithm by reducing the number of searches. In order to do this, the current maximum and the current minimum are stored in the form of alpha and beta respectively.

```
alpha <- -∞
beta <- ∞
FUNCTION minimax(board, depth, maximisingPlayer,alpha,beta)
        IF depth == 0 THEN
                RETURN heuristic value of board
        ELSE IF maximisingPlayer == TRUE
                eval <- -∞
                # each valid move in a given position
                FOR each move:
                        eval <- MAXIMUM( value, minimax(board.play(move),depth-1,FALSE,alpha,beta)
                        alpha = MAXIMUM(alpha,eval)
                        IF beta <= alpha THEN
                                BREAK
                        ENDIF
                # board.play(move) will return the board state after the move is made
                NEXT move

                RETURN eval
        ELSE
                value <- ∞
                FOR each move:
                        eval <- MINIMUM( value, minimax(board.play(move),depth-1,TRUE,alpha,beta)
                        beta = MINIMUM(eval, beta)
                        IF beta <= alpha THEN
                                BREAK
                        ENDIF
                NEXT move

                RETURN eval
END FUNCTION
```

Figure 15 Pseudocode for Alpha-Beta pruning.

Lookup Table (LUT)

A LUT is a method of optimising an algorithm by replacing a runtime computation with a pre-computed value. In my project, the Minimax function is the most costly algorithm due to its recursive nature. Because there are a finite number of Othello positions and because some positions appear frequently early in the game, it is wasting resources calculating the evaluation of a position if the algorithm has already calculated this value in the past. Therefore, when a board state is encountered, the algorithm should check if it has already seen this board in the past before beginning any computation. If it has, it should return this previously computed value. If it has not, it should calculate the value as normal and then store it in the LUT for future use. A LUT uses a simple array indexing operation to find the stored value but this index needs to be calculated. An obvious index to be used is just the full board-state string as this will always be unique for that board state but this would be highly inefficient due to the fact it is always 64 characters long. Because of this, it becomes necessary to perform an algorithm to map each board-state onto a unique value while minimising space used. This is detailed in Optimising Lookup. To allow for persistent data storage, the LUT should be saved between when the program is run. This could be done either using a SQL database or using a file system. I have chosen to store the LUT in a text file that can be written to after every move and read from at the beginning of each match. This is because a SQL server requires internet access, so the text file prevents the system from becoming redundant if the user is playing offline.

```
lookupTableFile <- FILE("lookuptable.txt")    # Provides reference to where the LUT is stored
lookupTable <- lookupTableFile.READLINES   # read file contents, stored in format: [BOARD, EVALUATION]
lut <- {}    # Dictionary
FOR INT count <- 0 TO lookupTable.LENGTH:
        lut[lookupTable[count][0]] <- lookupTable[count][1]
NEXT count
```

Figure 16 Pseudocode to generate LUT from the file it is stored in at the beginning of program run.

```
IF board IN lut THEN
        bestMove <- lut[board]
ELSE
        bestMove <- minimax(board,depth,FALSE,-∞,∞)     # Maximising player set to FALSE as the AI is playing black so is minimising
        lut[board] <- bestMove
END IF

playMove(bestMove,BLACK)
```

Figure 17 Pseudocode to determine whether the move needs to be calculated or not.

Optimising Lookup

In order to allow for direct access lookup, the board states need to be stored as efficiently as possible whilst still ensuring that each entry is unique. They were originally going to be stored as strings representing the board, however, these strings are 64 characters long and may look like this:

"0000000000000000000000000001200000021000000000000000000000000000"

This is the example string for the starting position of the match where 0 is a blank tile, 1 is a white tile and 2 is a black tile. Because this string could be read as a number, it could also be stored as an integer. However, I have realised it will be more efficient to treat this string as a base 3 number (ternary form) as only 3 digits are used: 0, 1 and 2. By treating the string as a base 3 number and converting it to an integer, it will allow for a unique integer value for every board state that will be smaller than the 64-digit integer currently used. This can be down with two functions. *toTernary* will take a decimal number (the index in the LUT) and convert it into ternary form (the state of the board). *toDecimal* will take a ternary number (the state of the board) and convert it into a decimal number (the index it will be stored at in the LUT).

```
FUNCTION toTernary(n)
        IF n == 0 THEN
                RETURN 0
        ELSE
                result <- ""
                WHILE n > 0 DO
                        remainder <- n MOD 3
                        n <- n // 3
                        result <- result + STRING(n)
                END WHILE
        result <- REVERSE( result )
        result <- result.LEFTFILL('0',64)
        RETURN result
END FUNCTION


FUNCTION toDecimal(n)
        n <- REVERSE( n )
        decimal <- 0
        FOR power <- 0 TO 63:
                decimal <- decimal + (n[power] * (3**power))
        NEXT power
        RETURN decimal
END FUNCTION
```

Figure 18 Pseudocode for the two conversion functions.

As integer comparison is faster for smaller numbers, there is a further optimisation that can be made to slightly improve look up speed. Because pieces are never removed in Othello, the starting 4 tiles will always exist. Due to the ternary to decimal conversion, these would produce the lowest possible decimal value when all 4 are white. Even though this is an impossible

board state to achieve as other tiles would need to be played in order to flip all to white, it is a simple cut off to the minimum the decimal value for a board can be. Therefore, it can be subtracted from the decimal value for a board.

The ternary value and decimal equivalent is shown for the starting square:

0000000000000000000000000001200000021000000000000000000000000000
250211104677393444

And the two values for the hypothetical minimum I will use initially:

0000000000000000000000000001100000011000000000000000000000000000
200156682785938776

This means the value 200156682785938776 can become a constant integer *LOOKUPSHIFT* used in calculation that is subtracted from each decimal value. The number can be changed if it is necessary for speed to further optimise the look-up process.

Evaluation Algorithm

The Minimax algorithm implemented above requires a numerical evaluation for each position it encounters in order to work. The evaluation algorithm needs to return a positive number if white is winning, a larger number implies white is winning by more. A negative number will show black as winning. This needs to be based upon Othello strategy as to what is a good move and what is not. Efficiency will be key in the evaluation algorithm as it will be performed an exponentially increasing number of times based upon the depth of the Minimax algorithm.

An advantage of the AI system is that it can compute all possibilities for multiple moves ahead. This is an ability most players generally do not have and therefore gives it an inherent advantage over any player. The evaluation algorithm needs to be good enough that when coupled with this advantage and placed in hard mode, it will be highly challenging to beat.

An Othello strategy guide written by a creator named Marcin on Bona Ludo – a blog for board game strategy and tips – provides helpful information.

https://bonaludo.com/2017/01/04/how-to-win-at-othello-part-1-strategy-basics-stable-discs-and-mobility/

Marcin describes the concept of a ‘stable’ and ‘unstable’ disc. A stable disc is one that cannot ever be flipped because there is no place where the opponent can play in the future that would cause it to be flipped. A corner disc is always stable as it is not possible ever to be flipped once it is played.

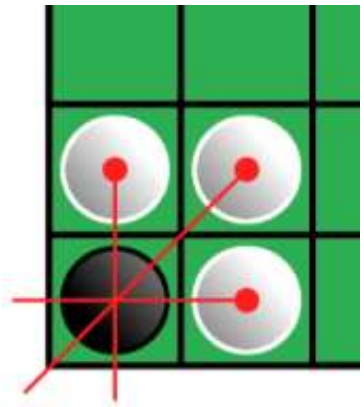


Figure 19 Impossibility of capturing a corner tile.

This means that corner discs are incredibly strong and should be sought after. A player having a corner disc allows for them to create more stable discs on the board. As long as a corner is captured, all discs along the edge of the board that are adjacent to this corner cannot be captured.

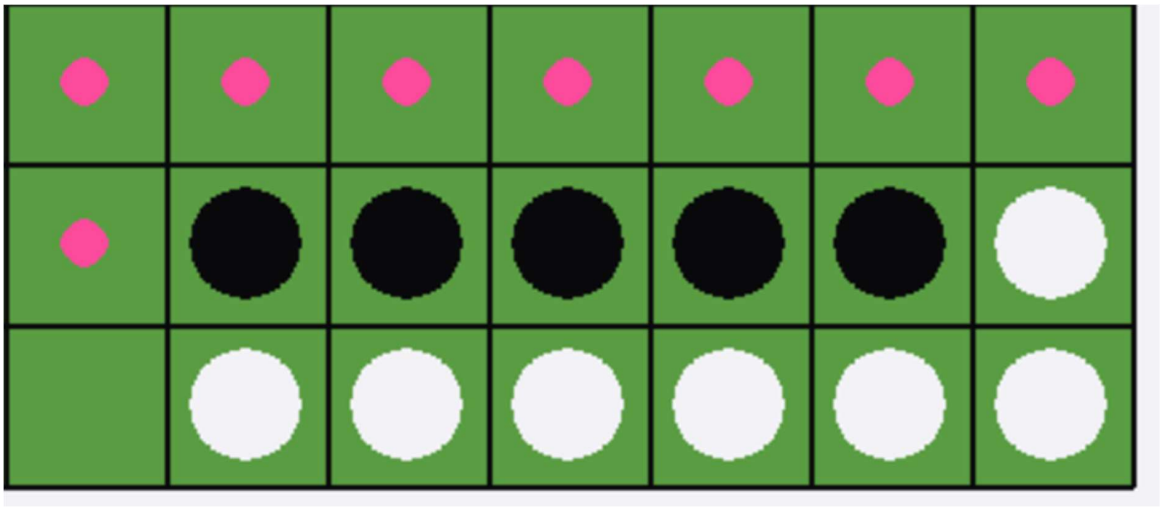


Figure 20 All white tiles can never be captured by black and also provide the benefit of being able to capture other black pieces.

Once these edges have been made stable, it is possible to work inwards on the board making more and more discs stable. In the below position, there are 17 stable white discs as for each disc there will never be a move in the future black can play to flip it. This position is completely winning for white as black only has unstable discs.

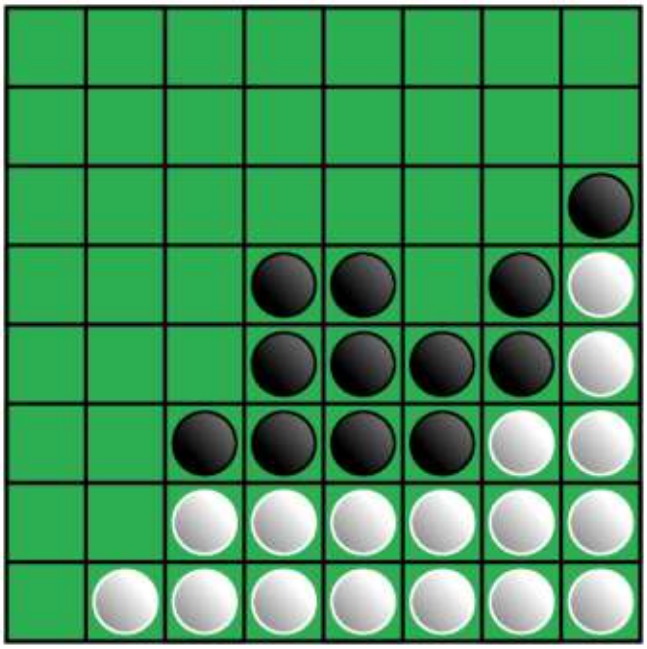


Figure 21 A position where all white tiles are stable.

Due to the power of stable discs, they are immensely beneficial for the player who has them. Therefore, I will need to add an algorithm to determine if a tile is stable and count the number of stable tiles each player has and use this information to adjust the evaluation of a position.

Because stable tiles can only be created with a corner tile (apart from a couple of exceptions which are so rare and hard to manufacture they can be discounted) the evaluation algorithm should have different priorities depending on the phase of the game. Before any corners have been captured, the AI should play in a manner that allows it to capture the first corner. After it has captured a corner, it should split its interests between capturing other corners and creating as many stable pieces around the corner as possible as this will allow it to win later in the game.

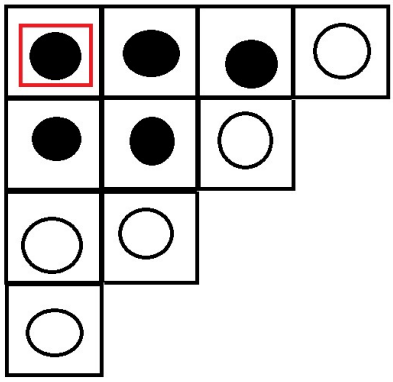
In order to do this, an algorithm needs to be implemented to check for stable pieces. For a piece to be stable, in every direction that an opposition tile or an empty space an opposition tile could be played, the piece needs to be defended from being flipped. For a piece to be defended fully, every tile in that direction must be stable and the same colour as it until it reaches the edge of the board. This means that the algorithm for finding the stable pieces needs to start in the corner and iterate out

along the two joining edges. Once reaching the end of stability along that edge, the algorithm needs to look one side further inwards and check for stability here. This process should slowly work inwards until no more stable edges are found.

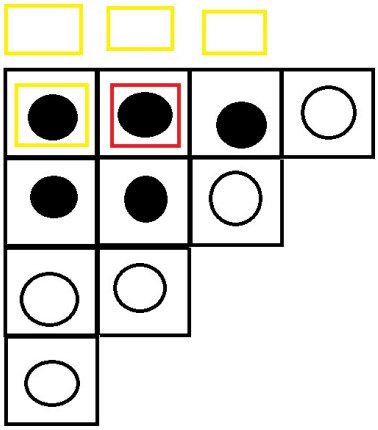
```
FUNCTION stableGenerater(board,player)
        stableCount <- 0
        corners <- [0,7,56,63]
        directions <- [[1,8],[-1,8],[1,-8],[-1,-8]]
        toCheck <- [[-9,-8,-7,-1],[-9,-8,-7,1],[9,8,7,-1],[9,8,7,1]]
        stable <- ARRAYLIST[INT]
        FOR index <- 0 TO 3:
                curCorner <- corners[index]
                IF board[curCorner] == player THEN
                        stableCount <- stableCount + 1
                        stable.APPEND(curCorner)
                        curPosition <- curCorner
                        xDirection <- directions[index][0]
                        yDirection <- directions[index][1]
                        DO
                                validFound <- False
                                curPosition <- curPosition + xDirection
                                DO
                                        stableFound <- False

                                        IF exists(curPosition,curPosition) THEN
                                                IF board[curPosition] == player THEN
                                                        checksPassed <- 0
                                                        FOR checkIndex <- 0 TO 3:
                                                                checking <- curPosition+toCheck[index][checkIndex]
                                                                IF checking IN stable OR !exists(curPosition,checking) THEN
                                                                        checksPassed <- checksPassed + 1
                                                                ENDIF
                                                        NEXT checkIndex
                                                ENDIF
                                                IF checksPassed == 4 THEN

                                                        stableCount <- stableCount + 1
                                                        stableFound <- TRUE
                                                        validFound <- TRUE
                                                        stable.APPEND(curPosition)
                                                ENDIF
                                        ENDIF
                                        curPosition <- curPosition + xDirection
                                WHILE stableFound == TRUE


                                curPosition <- curCorner


                                DO
                                        stableFound <- False

                                        IF exists(curPosition,curPosition) THEN
                                                IF board[curPosition] == player THEN
                                                        checksPassed <- 0
                                                        FOR checkIndex <- 0 TO 4:
                                                                checking <- curPosition+toCheck[index][checkIndex]
                                                                IF checking IN stable OR !exists(curPosition,checking) THEN
                                                                        checksPassed <- checksPassed + 1
                                                                ENDIF
                                                        NEXT checkIndex

                                                        IF checksPassed == 4 THEN
```

```
                                                        stableCount <- stableCount + 1
                                                        stableFound <- TRUE
                                                        validFound <- TRUE
                                                        stable.APPEND(curPosition)
                                                ENDIF
                                        ENDIF
                                ENDIF

                                        curPosition <- curPosition + yDirection
                                WHILE stableFound == TRUE
                                curCorner = curCorner + xDirection + yDirection
                        WHILE validFound == TRUE
                ENDIF
        NEXT index
END FUNCTION

FUNCTION exists(before, after)
        valid <- TRUE
        IF after < 0 OR after > 63 THEN
                valid <- FALSE
        ELSE IF after MOD 8 != before MOD 8 THEN
                valid <- FALSE
        END IF
        RETURN valid

END FUNCTION
```

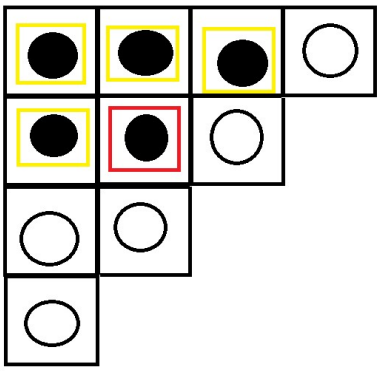
A diagram of how this function works is detailed below. This is a snippet of the board space we are checking. We first check that the corner is a tile of the player we are checking (black), which it is, so we add it to the array of stable pieces.



Next, we move in the x direction and check all the relevant directions are either off the board or stable, which they are, so the tile we are checking is added to the array of stable pieces.



This process is repeated in the x direction until a non-stable or non-black piece is found, which will occur at the white tile in this direction. The subroutine then resets the current position back to the corner piece and then starts checking in the y direction. Once this whole process is completed, if stable pieces are found, the corner is then set to be one piece diagonally towards the centre of the board and the process is repeated.



As all these pieces are all stable, this tile we are currently checking is also stable. The same process of checking in the x and y direction is conducted and if stable pieces are found in either of these directions (which they won't be in this example) the corner is moved in again. This entire process is conducted on every corner. Another reason why I have designed the algorithm this way is so that it will double count and therefore value entire connected rows at a higher level. If the whole of the top row is owned by one player, each stable piece is counted twice. This is designed this way as owning an entire row or column on the edge of the board is the most powerful position in Othello so should be favoured strongly by the algorithm.

Othello is a more defensive game where it is challenging to attack the first corner but a lot easier to allow the other player to make a mistake allowing you to capture the corner. This means the AI should avoid making these mistakes while also giving itself the best possible chance for capturing the corners.

One way to avoid letting the opponent capture corners is to avoid playing in the 'X-squares'. These are the tiles diagonally inwards from the corners as these can be easily captured by playing in the corner.

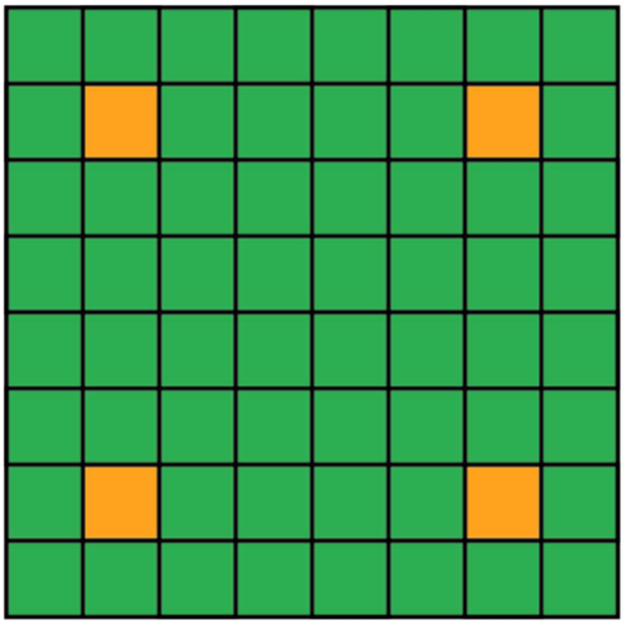


Figure 22 The X-squares.

The other squares that make it easier to capture corners (although still harder than X-squares) are the other adjacent tiles to the corner.

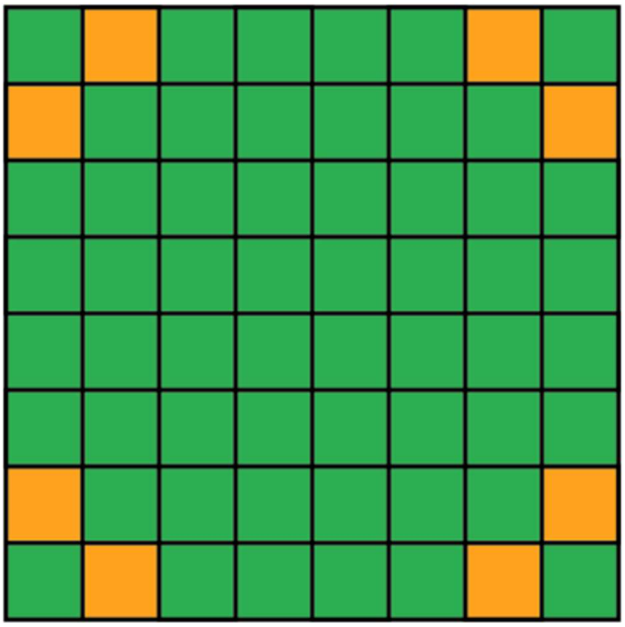


Figure 23 Adjacent tiles to corners allowing for easier capture of the corner.

To implement this into my code, I have assigned each tile on the board a weight based on the positional advantage it provides. The central 4 tiles allow for more options for play so are strong tiles to capture but anything adjacent to the edges and corners is negatively weighted. This is implemented as an 8x8 matrix containing the weights and the initial values I have chosen are shown below. The weights can be refined as I play the algorithm and test it.

```
boardWeights = [30 , -5 , 0 , 0 , 0 , 0 , -5 , 30 ,
               -5 , -10 , 0 , 0 , 0 , 0 , -10 , -5  ,
                1 , 0 , 1 , 1 , 1 , 1 , 0 , 1  ,
                1 , 0 , 1 , 2 , 2 , 1 , 0 , 1  ,
                1 , 0 , 1 , 2 , 2 , 1 , 0 , 1  ,
                1 , 0 , 1 , 1 , 1 , 1 , 0 , 1  ,
                -5 , -10 , 0 , 0 , 0 , 0 , -10 , -5  ,
                30 , -5 , 1 , 1 , 1 , 1 , -5 , 30   ]
```

Database

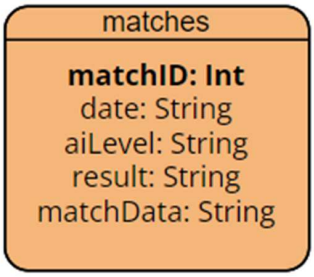
In order to achieve the goal of a database in order to store the previous matches played, **Objective 8**[6], it will be necessary to design a table suitable for this purpose. Each match will need a primary key as a unique identifier which should be an autoincrementing integer named **matchID**. This will allow users to request matches from the database by the matchID number. To help users recognise which match they are looking for in the database, it may be beneficial to store the date the match was played. I will store this as a String, as the DateTime module in Python produces a user-friendly string which is easily readable.

```
>>> datetime.datetime.now().ctime()
'Wed Jun 14 11:42:41 2023'
```

I will store this under the attribute **date**. Additionally, I wish to store the difficulty of AI the user was playing against with the match as it may be beneficial for the user to look for only matches they played against a more challenging AI in order to improve. This will be stored as a string. Example values may be easy, medium and hard. This will be the attribute **aiLevel.** Additionally, sorting by matches the user won or lost will be beneficial for the user so the result of the match will be stored as a string of either Win, Loss or Draw in the table under the attribute **result.** Finally, the attribute **matchData** will store a

[6] Objective 8 - Allow the user to review matches played.

string that will represent the moves made during the match. This should not be shown to the user and only used by the program to load the match into the reviewing software. These attributes produce the following entity:



Now that the design of the table is decided, I need to choose how I will actually convert the information about the match played into a single string to be inserted into *matchData* and stored (Objective 8.2)[7]

Given we have a system already from Optimising Lookup to convert a given board into a numerical value by treating the board state as a ternary value, it seems wise to reuse this system. A character can be used in-between each new board state in order to separate which value is which. I have decided to use the character 'n'. I can then write two functions, one to convert from the stored data to the board states and one to convert back the other way.

```
# Takes in all the board states from a match and converts it into data to be stored in the table
FUNCTION boardsToData(boards) # Array of all board states
        data <- "" # Stores the data which will be stored in the table
        FOR index <- 0 TO LENGTH(boards) - 1:
                data <- data + toDecimal(boards[index])
                data <- data + 'n'
        NEXT index
        RETURN data
END FUNCTION


# Takes in the data from the table and converts it into all the board states to be processed
FUNCTION dataToBoards(data) # String of data
        boards <- ARRAYLIST[STRING] # Stores the boards in their decimal form
        previous <- ""
        FOR index <- 0 TO LENGTH(data) - 1:
                IF data[index] == 'n' THEN
                        boards.APPEND(previous)
                        previous <- ""
                ELSE
                        previous <- previous + data[index]
                END IF
        NEXT index

        FOR index <- 0 TO LENGTH(boards)-1:
                boards[index] <- toTernary(boards[index])
        NEXT index


        RETURN boards
END FUNCTION
```

With this system now working, it is possible to store and load past matches from a persistent database. I have done this using MySQL as I proposed in my Software Decisions. However, after some initial testing and research, I have decided to not stay accurate to my design in the Overview section specifically about giving the user the choice of where they wish to save the game. I feel like this detail should be abstracted away from the user for an improved experience so will just be using MySQL as the method of storage due to highly infrequent errors with this system meaning that there is no need for an alternative. MySQL is configured to be able to be run without internet connection meaning errors of this nature will not occur and the match data will always be stored.

[7] Objective 8.2 - Once a match is complete, store it in a database along with the time it was played.

Review System

With the game able to store and load past matches, I now need to write the review system where the user can step through the match and view the best possible move in each situation. In order to not overwhelm the user with too much information, it is likely best to only show the best possible move and not all the moves in order of strength; however, I will ask my client which they would prefer before creating the system. I first need to decide how to display the best move in the current situation. I think I will use a method similar to how I've shown the most recent move but just change the colour of the outline of the box. The two highlights are shown below. Following client feedback, I have made the highlighting of the best move slightly thicker as the pinkish colour doesn't stand out as well as the yellow for the most recent move but keeps with the colour palette chosen. The user will always be able to tell which box highlights a recent move and which highlights the best move as the most recent move will always be around a piece, but the best move will be an empty tile.

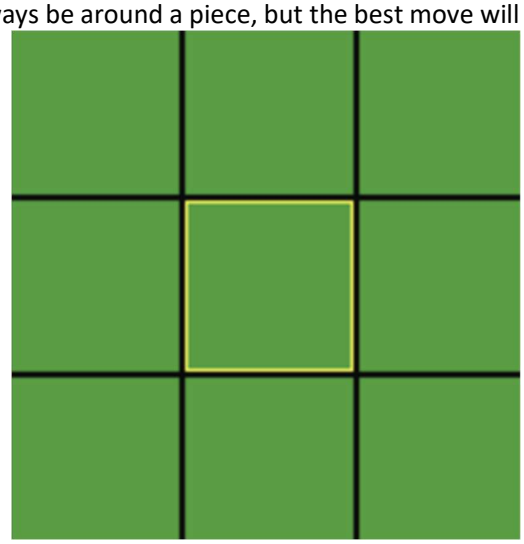


Figure 24 Highlighting the most recent move.

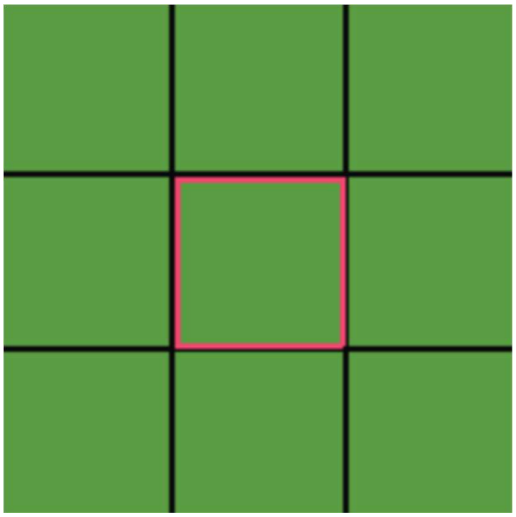


Figure 25 Highlighting the best move.

While loading the match data from the database, I plan to run the move evaluation function on all positions in that match and store the best move in a table to allow for faster access for the user. To do this, I can utilise multithreading. This is where functions are run in parallel on different CPU threads in order to increase the speed and efficiency of multi-tasking. This would allow for the user to be able to continue interacting with the program whilst the evaluation algorithm is running in the background.

After some consideration and testing, I have decided how I will implement multithreading into the program. I will load the match into the review system and will have one thread running the review system maintaining the board and checking for inputs about moving forward and backward through the game states. When a new game state is loaded, a new thread will be created finding the best position to play at a depth *d* which I can begin at a value of 3 so it will run quickly. Once this thread has finished running it will return the best position according to looking a depth of 3 into the future. Once this is returned, d

will be incremented by 2 making it, for example, 5. Then, a new thread is created evaluating the position at depth 5. This process is repeated looking deeper and deeper into the future providing a more and more accurate calculation of the best moves in the position.

As this process is continuous in the given position, once the user changes the board state they are looking at, the actively running thread will be immediately terminated to free up processing space and the cycle of evaluating the board at increasing depths will begin again in the new position.

Technical Solution

Main Menu

For the menu I have decided to use a text-based interface for simple operation and navigation. If the user wishes to play against the AI, the file which runs the game against AI is loaded and once this is finished, the match data is stored in the database.

```
if answer == "1": # Play Othello against AI
    import OthelloAI   # Load Othello game against AI
    summary = OthelloAI.summary()
    DB.insert_data(summary[0],summary[1]) # Insert the match data into the database
```

If the user wishes to review a past match, the keys and details of all previous matches are fetched from the database and shown to the user. Input from the user is then validated to be a correct key then the review system is loaded and passed the match data.

```
31      elif answer == "2": # Review a past match
32          pastMatches = DB.get_vals() # Fetch past matches from the database
33          matchKeys = []
34          for item in pastMatches:
35              print(item) # Display match details to the user
36              matchKeys.append(str(item[0]))    # Display to the user all the possible past matches
37
38          key = -1
39          print("Enter the key of the match you want to review")
40          key = input()
41          while key not in matchKeys:    # Get a valid matchKey from the user
42              print("Does not exist")
43              key = input()
44
45          matchData = DB.get_data(key)    # Retrieve the matchdata of the match chosen
46          import OthelloReview    # Load the review system
47          OthelloReview.review(matchData)    # Pass the matchdata to the review system
```

For playing against another player, the relevant program is similarly run with results being stored into the database.

```
51      elif answer == "3": # Play against another Player
52          import OthelloPVP # Load Othello game against another player locally
53          summary = OthelloPVP.summary()
54          DB.insert_data(summary[0],summary[1]) # Insert the match data into the database
```

If the user requests more information about the AI system, an initial message is sent explaining the system and then an API call is made to the RESTful Bard API setting up a dynamic API key for exchange (in this snippet a static key is used for the specific hour the call was being made, the key would no longer be valid after this hour). A request is then made with a setup message, priming the Large Language Model to respond appropriately to any messages from the user. These messages are passed to the LLM and any errors in the API exchange are written to a log file.

```
    elif answer == "4": # Information about the AI

        print("""This AI works using an algorithm called MiniMax.\n
This means that the algorithm assumes that you are a perfect player and will try to
minimise the maximum score you can get in any position.\n
It calculates a numerical score to show who has the advantage with a positive number meaning
a white advantage and negative meaning black advantage.\n
The bigger the number, the bigger the advantage.\n
If you have any further questions, you can ask the ChatBot Assistant, would you like to do this?""")
        askAI = input("y/n\n")
        if askAI == "y":
            setupMessage = """You are now a chatbot being used within a python program.
This program is a recreation of the board game Othello and uses an AI that uses the MiniMax algorithm.
Your role is to answer any questions from the user about the rules of the game or about how the AI system for the game works.
All your responses must be within this context."""

            # Setup the API for communication using the API key
            bard = Bard(token='fwixqCpb1S1nJ6m0X1hK-m3HBstQILyZDIgS-GZLtMNNcUvdSYIeRwDm4ZrApTOOmnIbiw.')

            # Send the LLM a message detailing the environment it is being used in to help it respond appropriately
            temp = bard.get_answer(setupMessage)
            print(temp['content'])

            message = input("Please enter your question or type \'exit\' to quit\n")
            while message != "exit":
                if message:
                    try:
                        # Make an API call to the BardAPI using the message to customise the response
                        response = bard.get_answer(message)
                        print(response["content"])
                    except Exception as e: # If an error occurs, write the error details to a log file
                        print("Unfortunately the ChatBot Assistant is not currently active, sorry for the inconvenience")
                        errorFile = open("errorLog.txt","w+")
                        errorFile.write(e)
                        errorFile.close()
                        break
                message = input("")
```

Database Management Program

In order to execute SQL queries and receive results from the database, I have created a function *execute_read_query()* which takes a SQL query that expects a result and executes it then returning the result. Any errors are logged to a file.

```
# Exectures a SQL query which has a return value and returns that result
def execute_read_query(connection,query):
    cursor = connection.cursor()
    result = None
    try:
        cursor.execute(query)
        result = cursor.fetchall()
        return result
    except Exception as e:
        errorFile = open("errorLog.txt","w+")
        errorFile.write(e)
        errorFile.close()
        print(e)
```

Another subroutine *execute_query()* executes a similar process just for SQL queries that are not returning anything so does not need to return the result.

```
# Exectures a SQL query which has a return value and returns that result
def execute_read_query(connection,query):
    cursor = connection.cursor()
    result = None
    try:
        cursor.execute(query)
        result = cursor.fetchall()
        return result
    except Exception as e:
        errorFile = open("errorLog.txt","w+")
        errorFile.write(e)
        errorFile.close()
        print(e)
```

The database and table are then created using these subroutines and relevant SQL queries.

```
create_database_query = "CREATE DATABASE IF NOT EXISTS othello"
create_database(connection,create_database_query)    # Creating the database if it does not already exist in the SQL

create_table = """
CREATE TABLE IF NOT EXISTS matches (
    matchID INT AUTO_INCREMENT,
    date TEXT,
    aiLevel TEXT,
    result TEXT,
    matchData TEXT,
    PRIMARY KEY (matchID)
) ENGINE = InnoDB
"""
execute_query(connection,create_table) # Creating the table if it does not already exist with matchID as the primary key
```

The database file also includes the *toTernary* and *toBinary* subroutines discussed in Optimising Lookup.

```
# Function to convert a board state that has been stored to the 0,1,2 - Blank,White,Black representation
def toTernary(n):
    n = int(n)
    n += LOOKUPSHIFT # Correct by the look up shift value
    if n == 0:
        return 0
    else:
        result = ""
        while n > 0:
            n,remainder = divmod(n,3)
            result += str(remainder)
    # Pad the result with 0s to ensure the string is 64 characters long and follows the standard board format
    return result[::-1].zfill(64)


# Function to convert a board state from the 0,1,2 representation to decimal
def toDecimal(n):
    n = str(n)[::-1]
    decimal = 0
    for power in range(64):
        decimal += int(n[power]) * (3**power)
    return decimal - LOOKUPSHIFT  # Shifts by the look up shift value to reduce storage size
```

For storage and retrieval of match data, each board state to be stored has to be run through the appropriate subroutine to convert it into the right format.

```
# Takes in all the board states from a match and converts it into data to be stored in the table
def boardsToData(boards):
    data = ""
    for item in boards:
        data += str(toDecimal(item))
        data += 'n'
    return data

# Takes in the data from the table and converts it into all the board states for processing
def dataToBoards(data):
    boards = []
    print(data)
    boards = data.split("n")
    boards.remove('')
    for index in range(len(boards)):

        boards[index] = str(toTernary(boards[index]))
    return boards

```

The database also needs functionality for retrieving data from the table to display previous matches to the user, retrieving matchData for a given matchID, and inserting the data from a completed match into the table. When inserting the data, information such as the result of the match and the time at which it was played is calculated and also stored to help the user distinguish between matches they have played.

```
# Gets relevant information from the table to display to the User a list of previous matches
def get_vals():
    query = """SELECT matchID,date,aiLevel,Result FROM matches"""
    result = execute_read_query(connection,query)
    return result


# Gets the match data for a match with a given key and returns the converted result
def get_data(key):
    query = """
    SELECT matchData
    FROM matches
    WHERE matchID = {matchKey}""".format(matchKey=key)
    result = execute_read_query(connection,query)[0][0]

    result = dataToBoards(result)
    #print("RESULT:",result)
    return result


# Converts the board states to data to be stored and stores it in the table
def insert_data(matchData, mode):
    lastBoard = matchData[-1]
    if lastBoard.count("2") != lastBoard.count("1"):
        winner = "Black" if lastBoard.count("2")>lastBoard.count("1") else "White"
    else:
        winner = "Draw"

    data = boardsToData(matchData)

    insert_vals = """
    INSERT INTO
        `matches` (`date`,`aiLevel`,`result`,`matchData`)
    VALUES
        ('{date}','{gameMode}','{win}','{matchInfo}');
        """.format(date=datetime.datetime.now().ctime(),gameMode=mode,win=winner,matchInfo=data)
    execute_query(connection,insert_vals)
```

Play Against the AI

When the program to play against the AI is first run, the first step is determining whether the player wishes to play as White, Black or a randomly chosen colour (black or white) and also asking the user which difficulty they wish to play against.

```
# Ask the user if they want to play as white, black or a random colour
answer = "-1"
options = ["1","2","3"]
while answer not in options:
    answer = input("Would you like to play as:\nWhite: 1\nBlack: 2\nRandom: 3\n")

# Select a random colour if selected
if answer == "3":
    import random
    answer = str(random.randint(1,2))
    if answer == "1":
        print("You are White")
    else:
        print("You are Black")

# Set the AI colour to the opposite of the player
AIplayer = str(3-int(answer))
player = answer


# Ask which difficulty they wish to play
options = ["1","2","3","4","5"]
answer = "-1"
while answer.lower() not in options:
    answer = input ("""Which difficulty do you wish to play on:\n
                    Beginner: 1\nEasy: 2\nMedium: 3\nHard: 4\nExpert: 5\n""")

# Set the difficulty string to be stored in the database
# Set the depth of the AI based on the difficulty
if answer == "1":
    difficulty = "Beginner"
    depth = -1
elif answer == "2":
    difficulty = "Easy"
    depth = 0
elif answer == "3":
    difficulty = "Medium"
    depth = 1
elif answer == "4":
    difficulty = "Hard"
    depth = 3
elif answer == "5":
    difficulty = "Expert"
    depth = 5

```

Next, the board is drawn for the first time. The *drawBoard* takes the current state of the board and the number of moves in the past the player is looking at. If the number of moves in the past is not passed to the board, it is assumed to be 0. The function first clears the screen and then begins drawing the grid, the pieces on the grid and slider to show the number of pieces on the board. It then highlights the most recently played piece and finishes by drawing the arrows and the quit button. The sizes of all these things are generated dynamically using mathematical equations that use the current width and height of the window being played in. This allows the game window to be resized to any size and still draw everything correctly.

```
# Function to draw the game window and board
def drawBoard(board,pastDifference=0):
    # Clear the gamewindwow by painting the screen and surface white
    screen.fill(FINW)
    surface.fill((0,0,0,0))

    # Draw the green playing board
    playingBoard = pygame.draw.rect(screen, ASPARAGUS, (wBuffer, hBuffer, wBoard, hBoard))

    # Draw the grid lines on the board
    for lineNum in range(9):
        pygame.draw.line(screen, FINB, (wBuffer, hBuffer + lineNum * (hBoard / 8)),
                         (wBuffer + wBoard, hBuffer + lineNum * (hBoard / 8)), max(1, hBoard // 200))
        pygame.draw.line(screen, FINB, (wBuffer + lineNum * (wBoard / 8), hBuffer),
                         (wBuffer + lineNum * (wBoard / 8), hBuffer + hBoard), max(1, wBoard // 200))


    # If currently looking at past board state, set the board to draw to be the past board
    if pastDifference != 0:
        board = boards[-(pastDifference+1)]

    # Iterate through the pieces of the board and draw them
    for count,item in enumerate(board):
        if item != "0":
            drawPiece(count,item)


    # Drawing the slider at the top of the board

    white, black = mm.countTiles(board)
    # Find the percentage of pieces that are black or white
    wProportion = white / (white + black)
    bProportion = 1 - wProportion
    sliderWidth = wBoard // 2
    sliderBuffer = int(min(height,width)*0.01)
    # Draw the slider
    pygame.draw.rect(screen,FINB,(width//4,sliderBuffer,
                                  (width//2) * wProportion,hBuffer - 2 * sliderBuffer),1)
    pygame.draw.rect(screen,FINB,(width//4 + (width//2) * wProportion,sliderBuffer,
                                  (width//2) * bProportion,hBuffer - 2 * sliderBuffer))


    # Load the font to display the slider text at the right size
    font = pygame.font.SysFont('didot.tcc', int((hBuffer - 2 * sliderBuffer)*0.8))
    if white > 0:
        # Render white text if there are tiles on the board
        whiteText = font.render(str(white), True, FINP)
        whiteRect  = whiteText.get_rect()
        whiteRect.center = ((width//4 + 0.5*((width//2) * wProportion)),
                            sliderBuffer + 0.5*(hBuffer - 2 * sliderBuffer))
        surface.blit(whiteText,whiteRect)

```

```
    if black > 0:
        # Render black text if there are tiles on the board
        blackText = font.render(str(black),True,FINP)
        blackRect = blackText.get_rect()
        blackRect.center = ((width//4 + (width//2) * wProportion) + 0.5*((width//2) * bProportion),
                            sliderBuffer + 0.5*(hBuffer - 2 * sliderBuffer))
        surface.blit(blackText,blackRect)


    # Highlight the most recent move in the current position as long as it is not the starting position
    if boards.index(board) != 0:
        highlightRecent(positionsPlayed[boards.index(board)-1])



    # Render the arrows to allow the user to click forward and backward through the game
    global arrowsHeight,arrowsWidth, arrowXCentre,arrowYCentre
    arrowXCentre = width - wBuffer*1.75
    arrowYCentre = height - hBuffer*0.5

    arrows = font.render("<< <  > >>",True,(0,0,0))
    arrowsRect = arrows.get_rect()
    arrowsRect.center = (arrowXCentre,arrowYCentre)
    surface.blit(arrows,arrowsRect)

    arrowsHeight,arrowsWidth = arrows.get_height(),arrows.get_width()


    # Render the quit text as a button for the user
    global quitHeight,quitWidth,quitXCentre,quitYCentre

    quitXCentre = wBuffer*1.75
    quitYCentre = height - hBuffer*0.5

    quitButton = font.render("quit",True,(0,0,0))
    quitRect = quitButton.get_rect()
    quitRect.center = (quitXCentre,quitYCentre)
    surface.blit(quitButton,quitRect)
    quitHeight,quitWidth = quitButton.get_height(),quitButton.get_width()

    # Render the surface and update the screen
    screen.blit(surface, (0, 0))
    pygame.display.update()
    return playingBoard

```

The procedure used within *drawBoard* to draw all the pieces, *drawPiece*, takes the position and type of tile to be drawn and calculates the appropriate location and size of the circle based upon window sizing before drawing the shape.

```
# Procedure to draw either a piece or an indicator that there is a valid move
def drawPiece(position, colour):
    """
    colour 1 is white
    colour 2 is black
    colour 3 is pink
    """

    # Calculate the x and y location of where the circle is going to be drawn
    x = (position%8)
    y = position//8
    x = int((x*wInterval) + wBuffer + (wInterval//2))
    y = int((y*hInterval) + hBuffer + (hInterval//2))
    if colour == WHITE:
        # Draw a white circle
        pygame.draw.circle(screen, FINW, (x, y), int(min(wInterval,hInterval) * 0.35))
    elif colour == BLACK:
        # Draw a black circle
        pygame.draw.circle(screen, FINB, (x, y), int(min(wInterval,hInterval) * 0.35))
    else:
        # Slightly smaller semi-transparent circle to be used for valid move indicator
        pygame.draw.circle(surface, FINP, (x, y), int(min(wInterval,hInterval) * 0.15))
        screen.blit(surface, (0, 0))

```

Once the initial game state is drawn, the main loop begins. If the current player – *curPlayer* – is the AI player, their move is calculated based upon the difficulty chosen by the player.

```
if curPlayer == AIplayer and inPresent:

    # Generate all possible AI moves
    AILocations = mm.findAllPieces(curPlayer,board)
    AIValidPositions = mm.findAllValid(curPlayer,AILocations,board)

    if len(AIValidPositions) != 0: # If there is a location where the AI can play
        if depth == -1:
            # Beginner difficulty so pick random location to play
            randomIndex= random.randint(0,len(AIValidPositions)-1)
            position = AIValidPositions[randomIndex]

        elif depth == 0:
            # Easy difficulty so pick the move that maximises the number of AI tiles
            position = mm.easyAI(board,curPlayer)
        else:
            # Medium or above so use MiniMax algorithm with the depth set by difficulty
            val, position = mm.minimax(board,depth,curPlayer == WHITE,1,len(boards),float("-inf"),float("inf"),[])


        board = mm.flipAll(curPlayer,position,board) # Flip all the pieces
        boards.append(board) # Add the board to the list of baord states
        positionsPlayed.append(position) # Add the location played to a list of all past locations

        # Draw the most recent played piece
        drawPiece(position,curPlayer)
        highlightRecent(position)

        # Add a short constant delay between playing the piece and flipping the pieces
        # This allows the user to better see what is happening
        time.sleep(PLAYDELAY)

        # Update the board
        playingBoard = drawBoard(board,pastDifference)
```

Once the AI has played or if it cannot play, *curPlayer* is flipped to being the users colour. It is then checked if the user cannot play. If the user cannot play and the AI cannot play, the match is finished so the *end* subroutine should be called. However, if the user cannot play and the AI can play, the AI is allowed to play again and again until the user has a valid move they could play.

```
# AI cannot play or has just played so change current player to the PLAYER
curPlayer = str(3-int(curPlayer))
playerLocations = mm.findAllPieces(curPlayer,board)
playerValidPositions = mm.findAllValid(curPlayer,playerLocations,board)

while len(playerValidPositions) == 0:

    #If the player cannot play, check whether the AI also cannot play and if so, the match is finished
    AILocations = mm.findAllPieces(str(3-int(curPlayer)),board)
    AIValidPositions = mm.findAllValid(str(3-int(curPlayer)),AILocations,board)
    if len(AIValidPositions) == 0:
        end(board)
        break

    playingBoard = drawBoard(board,pastDifference)

    curPlayer = str(3-int(curPlayer))
    time.sleep(2) # Small delay to allow user to realise they cannot play


    if depth == -1:
        # Beginner difficulty so pick random location to play
        randomIndex= random.randint(0,len(AIValidPositions)-1)
        position = AIValidPositions[randomIndex]

    elif depth == 0:
        # Easy difficulty so pick the move that maximises the number of AI tiles
        position = mm.easyAI(board,curPlayer)
    else:
        # Medium or above so use MiniMax algorithm
        val, position = mm.minimax(board,depth,curPlayer == WHITE,1,len(boards),float("-inf"),float("inf"),[])


    # Draw the piece and update the board
    board = mm.flipAll(curPlayer,position,board)
    positionsPlayed.append(position)
    boards.append(board)
    drawPiece(position,curPlayer)

    time.sleep(PLAYDELAY)

    # Draw the options for the player
    curPlayer = str(3-int(curPlayer))
    drawOptions(curPlayer,board)
    playerLocations = mm.findAllPieces(curPlayer,board)
    playerValidPositions = mm.findAllValid(curPlayer,playerLocations,board)

playingBoard = drawBoard(board,pastDifference)
drawOptions(curPlayer,board)
```

The *end* subroutine calculates who has won the match and then displays the result to the user on a small pop-up window using TKinter. It also closes the file containing the look up table in order to prevent any potential corruption.

```
127 # Procedure to be run when the match finishes
128 def end(board):
129     # Print the scores of the match
130     white, black = mm.countTiles(board)
131     endText = ""
132     if white < black:
133         endText = ("Black wins {b}-{w}".format(b=black,w=white))
134     elif black < white:
135         endText = ("White wins {w}-{b}".format(b=black,w=white))
136     else:
137         endText = ("Draw: {w}-{b}".format(b=black,w=white))
138 
139     # Close the transposition table file
140     mm.closeTable()
141 
142     # Create and display the pop-up label of the score
143     window = Tk()
144     lbl=Label(window, text=endText, fg='blue', font=("Helvetica", 16))
145     lbl.place(x=5, y=5)
146     window.title('')
147     window.geometry("200x50+10+10")
148     window.mainloop()
149 
```

Additionally, while the main loop is running, the *drawOptions* subroutine is sometimes called to display to the user the possible locations in which they can play. This subroutine utilises subroutines from the Minimax program and finds all the valid moves and then displays them to the user.

```
151 # Procedure to find all the valid moves for a colour and draw them as options
152 def drawOptions(colour,board):
153     locations = mm.findAllPieces(colour,board)
154     validPositions = mm.findAllValid(colour,locations,board)
155     for item in validPositions:
156         drawPiece(item,3)
157     pygame.display.update()
158 
```

When it is the user's turn, any mouse presses are registered and processed. The location of the click is used to calculate what has been pressed by the user. Firstly, it is checked if the user has clicked the board. If they have, the numerical value of the tile they have clicked is calculated and then checked if it is a valid move to be played. If it is, the move is played but if not, nothing changes.

```
    for event in pygame.event.get():

        if event.type == pygame.MOUSEBUTTONDOWN:

            # If the board has been clicked and the current board state is the most recent board
            if playingBoard.collidepoint(pygame.mouse.get_pos()) and inPresent:

                # Find the position of the press
                boardPos = (wBuffer, hBuffer)
                relativePos = (pygame.mouse.get_pos()[0] - boardPos[0],
                               pygame.mouse.get_pos()[1] - boardPos[1])
                curX = int(relativePos[0] // wInterval)
                curY = int(relativePos[1] // hInterval)
                position = curX + curY*SIZE

                # Check if the user has clicked a valid position
                locations = mm.findAllPieces(player,board)
                validPositions = mm.findAllValid(player,locations,board)

                if (position in validPositions):
                    # Play the move clicked by the user
                    board = mm.flipAll(player,position,board)
                    boards.append(board)
                    positionsPlayed.append(position)
                    playingBoard = drawBoard(board,pastDifference)
                    curPlayer = AIplayer
```

If the click was not on the board, there are two possible locations that would need to be processed. If the press is on the arrow keys, the program needs to calculate which arrow option was pressed and change the board as appropriate. If the press was on the quit button, the program should be ended.

```
            else:
                #arrows
                #<< < > >>
                # 9 characters
                xPos,yPos = pygame.mouse.get_pos()
                if  arrowXCentre-(arrowsWidth//2)  <= xPos <= arrowXCentre+(arrowsWidth//2)
                and arrowYCentre-(arrowsHeight//2)  <= yPos <= arrowYCentre+(arrowsHeight//2):
                    # arrows have been clicked, decide which arrow it is

                    xDistance = xPos - (arrowXCentre-(arrowsWidth//2))
                    clicked = xDistance // (arrowsWidth//9)


                    if 0 <= clicked <= 1:
                        pastDifference = len(boards) -1  # Set the difference to the maximum
                    elif 3 <= clicked <= 3:
                        pastDifference += 1 # Increase the difference, stepping backwards
                    elif 5 <= clicked <= 5:
                        pastDifference -= 1 # Decrease the difference, stepping forwards
                    elif 7 <= clicked <= 8:
                        pastDifference = 0 # Set the difference back to the present

                    # Restrict the difference to not cause an error
                    pastDifference = min(pastDifference,len(boards)-1)
                    pastDifference = max(pastDifference,0)

                    # Redraw the board in the new state
                    if pastDifference != 0:
                        inPresent = False
                        playingBoard = drawBoard(board,pastDifference)
                    else:
                        inPresent = True
                        playingBoard = drawBoard(board,pastDifference)
                        drawOptions(curPlayer,board)

                # Check if the quit button is pressed
                elif quitXCentre-(quitWidth//2) <= xPos <= quitXCentre+(quitWidth//2)
                and quitYCentre-(quitHeight//2) <= yPos <= quitYCentre+(quitHeight//2):
                    # Close and save the transposition table then quit the program
                    mm.closeTable()
                    running = False
```

Another event that could occur is the user resizing the window. When this happens, the position of each tile is recalculated, and the necessary values are updated in order to redraw the board.

```
        # Check if the game window size has been changed
        elif event.type == pygame.VIDEORESIZE:

            # Save the new width and height of the window
            # Update the size of the screen and surface to match this
            width, height= event.w,event.h
            screen = pygame.display.set_mode((width, height),pygame.RESIZABLE)
            surface = pygame.Surface((width, height), pygame.SRCALPHA)


            # Calculate new board widths and buffers
            wBuffer = int(width * 0.1)
            hBuffer = int(height * 0.1)
            hBoard = height - 2 * hBuffer
            wBoard = width - 2 * wBuffer


            # Recalculate the positions of each playing square
            positions = []
            hInterval = hBoard / 8
            wInterval = wBoard / 8
            for row in range(8):
                for col in range(8):
                    positions.append(((wBuffer + wInterval // 2 + wInterval * col),
                          (hBuffer + hInterval // 2 + hInterval * row)))

            # Draw the updated board
            playingBoard = drawBoard(board,pastDifference)
            drawOptions(curPlayer,board)

        # Check if the window close button is pressed
        elif event.type == pygame.QUIT:
            # Close and save the lookup table and quit the program
            mm.closeTabe()
            running = False

```

Minimax Program

In order to run the Minimax algorithm, I first had to create some helper functions. This includes standalone subroutines such as counting the number of tiles on the board and finding all the pieces of a certain colour on the board.

```
# Helper function to return the number of white and black tiles respectively
def countTiles(board):
    return board.count(WHITE) , board.count(BLACK)


# Function to find an array of all the index locations of a piece of a certain colour
def findAllPieces(colour,board):
    locations = []
    for boardIndex in range(64):
        if board[boardIndex] == colour:
            locations.append(boardIndex)
    return locations

```

From an array of all the locations of the pieces of the colour whose turn it is to play and the current board state, *findAllValid* iterates through every direction for every piece and checks if there is a valid location to play in that direction using the *checkValid* recursive subroutine. These algorithms were designed in the Indicating Valid Moves section.

```
# Function to find all valid locations for the current player to play
def findAllValid(colour,locations,board):
    valid_locations = []
    for item in locations:
        for dx in range(-1,2):
            for dy in range(-1,2):
                if dx == 0 and dy == 0:
                    pass
                else:
                    valid, pos = checkValid(item,(dx,dy),colour,board)
                    if(valid):
                        valid_locations.append(pos)
    return valid_locations

# Recursive function to move in a direction checking if a move is going to be valid
def checkValid(location,direction,colour,board):
    # We are checking if the colour passed in is able to play
    # location is the current tile to be looked at
    # direction is a tuple (dx,dy)

    curX = (location%8) + direction[0]
    curY = (location//8) - direction[1]



    if (curX < 0) or (curX >= SIZE) or (curY < 0) or (curY >= SIZE):
        # If we are off the edge of the board, it is not valid
        return False, -1
    elif board[curY*8 + curX] == colour:
        # If we find a tile that is the same colour,
        # there is not going to be a valid move in this direction
        return False, -1
    elif board[curY*8 + curX] == str(3-int(colour)):
        # If it is the opposite colour, there could be a valid move this way so recurse
        return checkValid((curX+curY*8),direction,colour,board)
    else:
        # If a blank tile is found
        if board[location] == str(3-int(colour)):
            # If the previous tile is the opposite colour, this is a valid location to play
            return True, curY*8+curX
        else:
            # If the previous tile is the same colour,
            # there have been no opposite colour tiles to flip so not valid
            return False, -1
```

The subroutines to flip all the pieces work in a similar manner, taking the location of where the piece has been played and iterating through every direction recursively checking if it is suitable to flip the pieces in that direction. If it is found to be valid, each level of the recursive routine flips the piece it is checking and by the time the first stack frame is reached, a fully flipped *board* variable is returned ready for the next direction to be checked.

```
264 # Function to iterate through all directions and check if they can be flipped
265 def flipAll(colour,location,board):
266     board = board[:location] + colour + board[location+1:]
267     for dx in range(-1,2):
268         for dy in range(-1,2):
269             if dx == 0 and dy == 0:
270                 pass
271             else:
272                 # Call the recursive function flipIfValid in the direction specified
273                 temp, board = flipIfValid(location,(dx,dy),colour,board)
274     return board
275
276
277
278
279 def flipIfValid(location,direction,colour,board):
280
281     curX = (location%8) + direction[0]
282     curY = (location//8) - direction[1]
283
284
285     if curX < 0 or curY < 0 or curX >= SIZE or curY >= SIZE:
286         # If we are off the edge of the board, it is not valid
287         return False, board
288     elif board[curY*8 + curX] == '0':
289         # If we find a blank tile, this direction is not valid
290         return False, board
291     elif board[curY*8 + curX] == colour:
292         # If we find a tile that is the same as the tile we are checking,
293         # this is a valid direction to flip so pass True back to the previous call
294
295         return True, board
296
297     else:
298
299         # If the tile is the opposite colour,
300         # call the subroutine on the next tile in this direction
301         change, board = flipIfValid(curY*8+curX,direction,colour,board)
302         # If the subroutine finds this is a valid direction, change will be True
303
304         if change:
305             # Change the board variable so that the current tile is flipped
306             board = board[:curY*8 + curX] + colour + board[1+curY*8 + curX:]
307
308         # Return whether we are changing and the new board version
309         return change,board
310
```

During a later stage of production, I decided I wanted to attempt to optimise this process in order to run faster for the higher depth Minimax calls from the review system. I did this by restructuring and rewriting the procedure to be exclusively iterative and not involve recursion. This also means that the subroutine can be run in a thread without a risk of maximum recursion depth being reached as multiple call stacks would potentially be created.

```
# A version of the above flipAll function that is more efficient time wise
# This version is used in the review system by MiniMaxUnlimited()
def flipAll2(colour,location,board):
    # First place the piece in the location specified
    board = board[:location] + colour + board[location+1:]
    other = str(3-int(colour))
    for dx in range(-1,2):
        for dy in range(-8,9,8):
            if dx == 0 and dy == 0:
                pass
            else:
                cur = location+dx+dy
                new = cur+dx+dy
                try:
                    while board[cur] == other and 64>cur and cur>0:

                        if (cur % 8 == 0 and location % 8 != 0)
                        or (cur % 8 == 7 and location % 8 != 7 )
                        or new > 63 or new < 0:
                            break
                        cur = new
                        new = cur+dx+dy

                    if board[cur] == colour and board[cur-dx-dy] == other:
                        # If we reach the same colour and the previous tile
                        # Is the opposite colour, valid direction
                        cur = location+dx+dy
                        # reset cur to the starting location
                        while cur != new:
                            # Iterate through every tile in this direction
                            # Flip the piece in each location
                            board = board[:cur] + colour + board[cur+1:]
                            cur = cur + dx + dy

                except:
                    # Except clause to more efficiently handle index errors that
                    # indicate the end of the board rather than repeated comparison operation
                    pass
    return board


```

The Minimax algorithm I am using is the same as designed in the Minimax algorithm section. It utilises the optimisations of alpha-beta pruning and a lookup table to maximise efficiency.

```
def minimax(board, depth, maximisingPlayer,count,moveCount,alpha=float("-inf"),beta=float("inf"),table=[]):
    if count == 1:
        # Convert the board state to Decimal to optimise lookup
        decBoard = toDecimal(board)
        if decBoard in lut:
            #If the board is found in the lookup table, return the appropriate value
            return lut[decBoard][1],lut[decBoard][0]


    if depth == 0:
        # Base Case where the evaluation of the position is returned
        return evaluation(board)
    elif maximisingPlayer:
        # White is being checked
        val = float("-inf")

        # Find all valid moves for White
        locations = findAllPieces(WHITE,board)
        validPositions = findAllValid(WHITE,locations,board)


        if len(validPositions) == 0:
            # If there are no valid moves found for White, find all valid moves for black
            locations = findAllPieces(BLACK,board)
            validPositions = findAllValid(BLACK,locations,board)


            if len(validPositions)== 0:
                # If black also has no valid moves, someone has won so
                # count up the tiles to determine the winner and return the
                # appropriate value of + or - infinity
                white, black = countTiles(board)
                if white > black:
                    return float("inf")
                elif white < black:
                    return float("-inf")
                else:
                    # Return 0 if a draw is reached
                    return 0
            else:
                # Black has valid moves but White doesn't so skip White's turn
                # and call the algorithm for Black
                return minimax(board,depth,False,count+1,moveCount+1,alpha,beta)

        for pos in validPositions:

            # For every valid move White has, call the MiniMax algorithm recursively
            positionVal =  minimax(flipAll(WHITE,pos,board),depth-1,False,count+1,moveCount+1,alpha,beta)

            # If this current postion is the numerically best thus far, save the move and the evaluation
            if positionVal >= val:
                bestLocation = pos
                val = positionVal

            # Update the alpha value
            alpha = max(alpha,val)

            # Stop checking positions if Beta cutoff is reached
            if beta<= alpha:
                break

        # If this is the first layer of the call, return the evaluation of the board and
        # the best possible move in the situation, if not just return the evaluation
        if count == 1:
            # If there is only one valid move return that one
            if len(validPositions) == 1:
                return val, validPositions[0]

            # Write to the transposition table:
            # the state of the board, the best move and the corresponding evaluation
            towrite = str(toTernary(board))+" "+str(bestLocation)+" "+str(val)+" 1\n"
            transpositionTable.write(towrite)
            return val, bestLocation
        else:
            return val
    else:
        # Black is being checked
        # The samge system as White is being used but is mirrored to be applicable to Black
        val = float("inf")


        locations = findAllPieces(BLACK,board)
```

```
        validPositions = findAllValid(BLACK,locations,board)
        if len(validPositions) == 0:
            locations = findAllPieces(WHITE,board)
            validPositions = findAllValid(WHITE,locations,board)
            if len(validPositions)== 0:
                white, black = countTiles(board)
                if white > black:
                    return float("inf")
                elif white < black:
                    return float("-inf")
                else:
                    return 0
            else:
                return minimax(board,depth,True,count+1,moveCount+1,alpha,beta)
        for pos in validPositions:

            positionVal =  minimax(flipAll(BLACK,pos,board),depth-1,True,count+1,moveCount+1,alpha,beta)


            if positionVal <= val:

                bestLocation = pos
                val = positionVal
            beta = min(beta,val)
            if beta<= alpha:
                break

        if count == 1:
            if len(validPositions) == 1:


                return val, validPositions[0]
            towrite = str(toDecimal(board))+" "+str(bestLocation)+" "+str(val)+" 2\n"
            transpositionTable.write(towrite)

            return val, bestLocation
        else:
            return val
```

The base case of the Minimax algorithm uses the function *evaluation()* which takes in the board state and returns the numerical evaluation associated with it.

```
# Function to generate the numerical evaluation of a board
def evaluation(board):
    # Initialise evaluation at 0
    curEval = 0


    # Adjust the evaluation based on the number of white and black corner pieces
    wCorners = cornerCount(board,WHITE)
    bCorners = cornerCount(board,BLACK)
    curEval += (wCorners-bCorners)* cornerWeight


    # If a corner has been taken, run the stable piece checker function
    if wCorners > 0 or bCorners > 0:
        curEval += (stableGenerater(board,"1") - stableGenerater(board,"2")) * stableWeight

    # Commented version of the globally defined boardWeights for ease of understanding
    """boardWeights = [30 , -5 , 4 , 3 , 3 , 4 , -5 , 30 ,
                   -5 , -10 , 0 , 0 , 0 , 0 , -10 , -5  ,
                    4 , 0 , 1 , 1 , 1 , 1 , 0 , 4  ,
                    3 , 0 , 1 , 2 , 2 , 1 , 0 , 3  ,
                    3 , 0 , 1 , 2 , 2 , 1 , 0 , 3  ,
                    4 , 0 , 1 , 1 , 1 , 1 , 0 , 4  ,
                    -5 , -10 , 0 , 0 , 0 , 0 , -10 , -5  ,
                    30 , -5 , 4 , 3 , 3 , 4 , -5 , 30   ]"""


    # Add or subtract the board position evaluation if there is a white or black tile there
    for index,score in enumerate(boardWeights):
        if board[index] == WHITE:
            curEval += score
        elif board[index] == BLACK:
            curEval -= score

    return curEval

```

The evaluation function also takes into account the number of stable pieces on the board (discussed in Evaluation Algorithm) and multiplies the difference in the number of pieces by a fixed value of *stableWeight* to help calculate the overall score. The function to count the number of stable pieces for a certain colour is shown below.

```
# Function to count the number of stable pieces on the board for a given player
def stableGenerater(board,player):
    stableCount = 0
    corners = [0,7,56,63]
    directions = [[1,8],[-1,8],[1,-8],[-1,-8]]
    toCheck = [([-9,-8,-7,-1],[-9,-8,7,-1]),([-9,-8,-7,1],[9,-8,-7,1]),
               ([9,8,7,-1],[-9,8,7,-1]),([9,8,7,1],[9,8,-7,1])]
    stable = []
    for index in range(4):
        # For every corner on the board
        curCorner = corners[index]
        if board[curCorner] == player:
            # If the player we are checking has the corner,
            # the corner is stable so check in both directions
            stableCount += 1
            stable.append(curCorner)
            curPosition = curCorner
            xDirection = directions[index][0]
            yDirection = directions[index][1]


            # While a valid stable tile exists from this corner, keep checking
            validFound = True
            while validFound:
                curPosition = curCorner
                validFound = False
                # Start by checking the x direction
                curPosition += xDirection
                # While a valid stable tile exists in the x direction, keep checking
                stableFound = True
                while stableFound:
                    stableFound = False

                    if exists(curPosition,curPosition):
                        # If we are checking a valid location

                        if board[curPosition] == player:
                            # If every tile in every appropriate direction is stable,
                            # this tile is stable
                            checksPassed = 0
                            for checkIndex in range(4):
                                checking = curPosition + toCheck[index][0][checkIndex]
                                if (checking in stable) or (not exists(curPosition,checking)):
```

```
                                    checksPassed += 1

                            if checksPassed == 4:
                                stableCount += 1
                                stableFound = True
                                validFound = True
                                stable.append(curPosition)
                        # Move onto the next tile
                        curPosition += xDirection

                # Reset the starting position to the corner
                curPosition = curCorner



                curPosition += yDirection

                # While a valid stable tile exists in the x direction, keep checking
                stableFound = True
                while stableFound:
                    stableFound = False

                    if exists(curPosition,curPosition):
                        # If we are checking a valid location
                        if board[curPosition] == player:
                            print(curPosition)
                            # If every tile in the appropriate direction is stable,
                            # the tile is stable
                            checksPassed = 0
                            for checkIndex in range(4):
                                checking = curPosition + toCheck[index][1][checkIndex]
                                if (checking in stable) or (not exists(curPosition,checking)):

                                    checksPassed += 1

                            if checksPassed == 4:
                                stableCount += 1
                                stableFound = True
                                validFound = True
                                stable.append(curPosition)

                    # Move onto the next tile
                    curPosition += yDirection
                # Change the corner to be one tile diagonally closer to the centre
                curCorner = curCorner + xDirection + yDirection
                if (curCorner - xDirection) in stable and (curCorner - yDirection) in stable
                and board[curCorner] == player:
                    stable.append(curCorner)
                    stableCount += 1
                else:
                    validFound = False

    return stableCount

```

The difficulty of the AI is determined by the depth passed to the Minimax algorithm. However, for lower difficulties, the Minimax algorithm is still too effective. For beginner AI, a random move is played. For the easy AI, I am using a simple algorithm which picks the move that will flip the most tiles possible.

```
# The AI function for the Easy difficulty AI
# Plays the move that maximises the number of tiles flipped
def easyAI(board,curPlayer):
    # Find all valid moves
    locations = findAllPieces(curPlayer,board)
    validPositions = findAllValid(curPlayer,locations,board)

    # Create an empty array of integers to store tile counts
    tileCounts = [0] * len(validPositions)

    for index in range(len(validPositions)):
        # Iterates through every valid move and calculates
        # the number of tiles of each colour and stores
        # the difference (white-black) in the array
        white,black = countTiles(flipAll(curPlayer,validPositions[index],board))
        tileCounts[index] = white-black

    if curPlayer == WHITE:
        # If the AI is playing as white,
        # play the move that maximises the value of white-black
        return validPositions[tileCounts.index(max(tileCounts))]
    else:
        # If the AI is playing as black,
        # play the move that minimises the value of white-black
        return validPositions[tileCounts.index(min(tileCounts))]
```

Review System

When the review system is initially called, it is passed all of the board states of the match from the database. It then iterates through these to find which move has been played and add this to the *positionsPlayed* array to allow for easier processing later on. Once this process has been completed, the primary thread continues with running the *main()* function whilst a new thread is created to generate the best move at a low depth in every position. This low depth best move is used to be shown to the user initially whilst moves at a greater depth are being checked.

```
def review(reviewBoards):
    global boards
    boards = reviewBoards
    global pastDifference
    pastDifference = len(boards)-1


    # Generate positionsPlayed based on what changes between board states
    global positionsPlayed
    prev = boards[0]
    for item in boards[1:]:
        for index in range(len(item)):
            if prev[index] == "0" and (item[index] == "2" or item[index] == "1"):
                positionsPlayed.append(index)
        prev = item

    # Begin calculating a prelimenary best move in each position at depth 1
    bestMovesThread = threading.Thread(target=lambda: genBestMoves(boards,1))
    bestMovesThread.start()

    # Run the main board interface at the same time
    main()
```

The *main()* function is very similar to the main loop in the Play Against the AI program in its handling of mouse clicks or arrow key presses to step forwards and backwards through the match. When the user is looking at a state of the board, a procedure similar to iterative deepening is run. First a thread being created to run Minimax at a certain depth and once this thread is detected to have been finished, a new thread is created at a greater depth. When the user switches between positions, the running thread is killed, and a new one is created to run Minimax in the new board state.

```
            # Highlight the best move
            highlightBest(bestMoves[len(boards)-pastDifference-1])
            # If we are still looking at the same board state as before
            if prevPastDifference == pastDifference:
                # If the running thread has terminated
                if not bestMoveThread.is_alive():
                    #Increase the depth
                    depth = depth + 2
                    # Update the evaluation display
                    changeEvaluation()
                    pygame.display.update()
                    # Start a new thread at the greater depth
                    bestMoveThread = threading.Thread(target=lambda:
                                                      genBestMove(boards[len(boards)-pastDifference-1],depth))
                    bestMoveThread.start()
            else:
                # If the board state has changed

                # Update the previous board state is now the current one
                prevPastDifference = pastDifference

                # Update the evaluation display for the new board state
                changeEvaluation()
                pygame.display.update()

                # Reset the depth and begin the first thread
                depth = 2
                bestMoveThread = threading.Thread(target=lambda:
                                                  genBestMove(boards[len(boards)-pastDifference-1],depth))
                bestMoveThread.start()
```

The initial subroutine that is called in the thread from the *review()* procedure iterates through every board state and calls Minimax for each one of them.

```
# Procedure to generate a preliminary best move in every position
# with a limited depth for speed
def genBestMoves(boards,depth):
    global bestMoves,evaluations
    for index,board in enumerate(boards[:len(boards)-1]):
        curPlayer = boards[index+1][positionsPlayed[index]]
        val, position = mm.minimaxUnlimited(board,depth,curPlayer == WHITE,1,
                                            len(boards),float("-inf"),float("inf"),[])
        bestMoves.append(position)
        evaluations.append(val)
```

The subroutine that does this in an individual position is below.

```
# Function to generate the best move in a single position that is called
# With increasing depth for more and more accuracy
def genBestMove(board,depth):
    global bestMoves,evaluations
    index = boards.index(board)
    curPlayer = boards[index+1][positionsPlayed[index]]
    val, position = mm.minimaxUnlimited(board,depth,curPlayer == WHITE,1,
                                        len(boards),float("-inf"),float("inf"),[])
    bestMoves[index] = position
    evaluations[index] = val
    return position
```

Testing

Testing Plan

First, I am going to test the whole system trying out each of the features of the program and playing through full matches to test moves are working correctly and the system links together well. During this process of playing multiple full matches, I am comprehensively testing all subroutines related to the game such as generating moves and flipping relevant tiles when a move is played. After that, I will test some of the behind-the-scenes subroutines used by the Minimax algorithm with specific board states to ensure they are counting discs correctly. Additionally, I will do a specific test on the subroutine for generating valid moves in a board state and the subroutine for flipping the tiles, but this will already have been thoroughly tested as I play through multiple matches anyway.

The link for my testing video that is referenced by timestamps in the following table is:

https://www.youtube.com/watch?v=BIp2ebCgQig

Area	Number	Objective	Evidenced
GUI	1.1	Have an 8x8 board that can contain black or white tiles.	0:20-2:45, 5:23-6:25, 6:45-7:40
	1.2	Display the winner of the match when the game is finished.	2:43-2:48
	1.4	Have an option to allow the player to restart the match.	2:56-3:01
	2	Have a system to demonstrate to the user the number of white and black discs face up on the board in each position that updates every time a move is made.	0:20-2:45, 5:23-6:25, 6:45-7:40
	3	Have a system to visually indicate to the player the most recently placed tile by the computer by outlining the square the computer played on.	0:20-2:45, 5:23-6:25, 6:45-7:40
	4	Have a system to indicate to the player which moves are valid and legal in the current board state by adding a semi-transparent dot in the centre of the playing square.	0:20-2:45, 5:23-6:25, 6:45-7:40
	8.1	Before launching the game, give the user the choice between playing a match or reviewing one.	0:06-0:10, 2:59-3:03, 5:04-5:10, 6:34-6:41,
	8.3	Display to the user a list of all previous matches played that are available for review.	3:02-3:12
	11	Allow the user to select the difficulty AI they wish to play against at the beginning of the match.	0:11-0:16, 5:11-5:21

Functionality	1.3	Flip the correct discs in accordance with the rules when a player makes their move.	0:20-2:45, 5:23-6:25, 6:45-7:40
	1.5	Setup the board in the correct position (D4W, D5B, E4B, E5W) every time the match starts.	0:20-0:23, 5:22-5:27, 6:44-6:48
	6	Allow the user to click on arrows or use the arrow keys to step forwards and backwards through the match they are currently playing or reviewing.	0:36-0:59, 3:13-4:58, 6:11-6:24
	8.2	Once a match is complete, store it in a database along with the time it was played.	3:02-3:12
	8.4	Allow the user to input the key of a match they wish to review and load that match into the review system.	3:02-3:12
	8.5	Write a match review system where the user can step through the match, seeing the evaluation and best move in each position.	3:13-4:58
	9	Allow the player to select the location of their move by clicking on the appropriate square with their mouse /trackpad/touch screen.	0:20-2:45, 5:23-6:25, 6:45-7:40
AI	5	Write an evaluation function to provide a numerical evaluation of a given Othello position.	3:13-4:58
	7	Allow the user to request information explaining how the AI evaluation function works so they can better understand the feedback they are given.	8:12-8:57
	10	Have an AI that can perform at different difficulties. The difficulty for a given match will be selected by the user before the match begins.	Expert AI: 0:20-2:45, Easy AI: 5:23-6:25

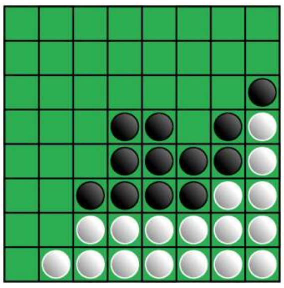
I also carried out more testing of each difficulty mode, trying my best to beat each mode. I beat beginner and easy mode relatively easily. I then drew to medium mode before beating it in a rematch. I could not beat hard or expert mode. Given I believe I am a somewhat strong player at Othello after my research into tactics and frequent practice during this project, I believe the hard and expert AI are very strong. The link to these matches is below:

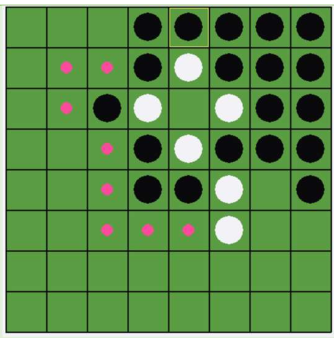
https://www.youtube.com/watch?v=x14VhxeiJbw

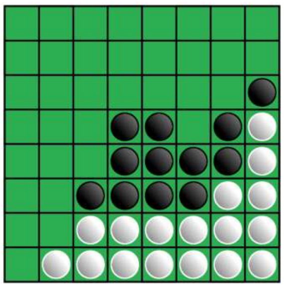
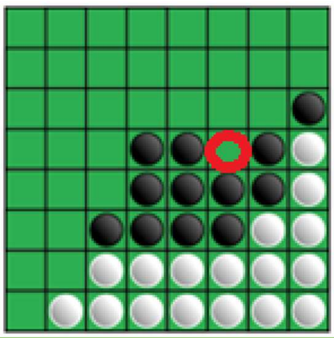
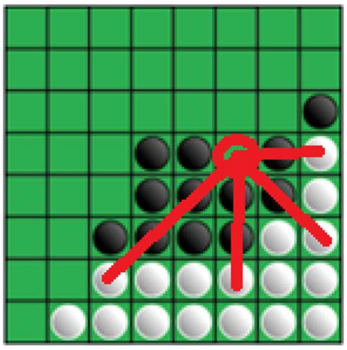
Testing key algorithms

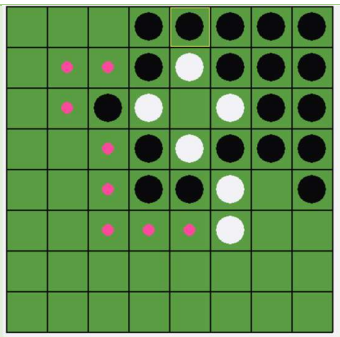
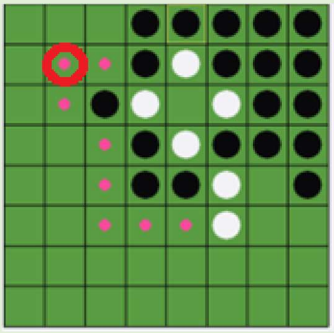
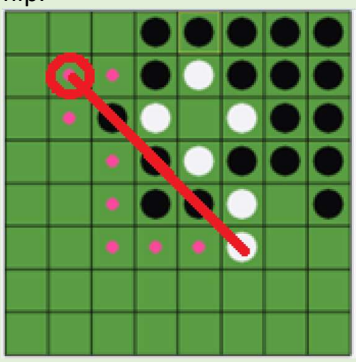
Sub-routine	**Testing Data**	**Manually Calculated Expected Result**	**Algorithm Calculated Result**

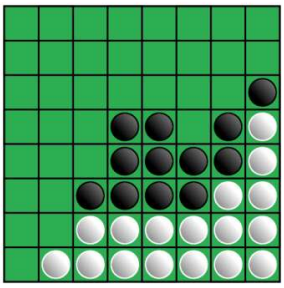
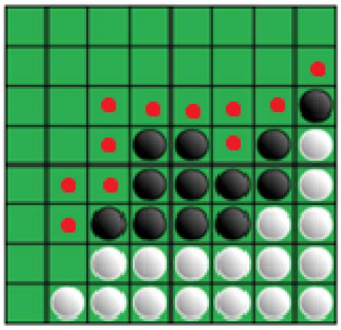
cornerCount	10001111110000000001111
	11111100000000000000000
	000000000000000002
		For white: 2
		For black: 1
			`>>> board ="100011111100000000011111111100000000000000000000000000000002"`
			`>>> cornerCount(board,"1")`
			`2`
			`>>> cornerCount(board,"2")`
			`1`
			`>>>`
			The correct number of corner tiles is counted for both white ("1") and black ("2") at 2 and 1 respectively.

stableGenerater	
	00000000000000000000000
	20002202100022221002222
	110011111101111111
		For white: 17 as every tile is stable.
		For black: 0 as no tile is stable.
			`>>> board = '000000000000000000000000200022021000222210022221100111111011111111'`
			`>>> stableGenerater(board,"1")`
			`17`
			`>>> stableGenerater(board,"2")`
			`0`
			The correct number of stable tiles is counted for both white ("1") and black ("2") at 17 and 0 respectively.

	
	00022222000212220021012
	20002122200022102000001
	0000000000000000000
		For white: 0 as no stable
		For black: 13 – 5 in top row, 3 tiles on the right in the second row, 2 tiles on the right in the third row, 2 tiles on the right in the fourth row, 1 tile on the right in the fifth row.
		5 + 3 + 2 + 2 + 1 = 13
			`>>> board = '0002222200021222002101220002122200022102000001000000000000000000'`
			`>>> stableGenerater(board,"1")`
			`0`
			`>>> stableGenerater(board,"2")`
			`13`
			The correct number of stable tiles is counted for both white ("1") and black ("2") at 0 and 13 respectively.

flipAll	
	Position 29 for White
	00000000000000000000000
	20002202100022221002222
	110011111101111111
	 
		Positions 30,37,38,45,36,43 should flip.
		 
		00000000
		00000000
		00000002
		00022111
		00021111
		00212111
		00111111
		01111111
			`>>> board = '0000000000000000000000020002202100022221002222110011111101111111'`
			`>>> result = flipAll("1",29,board)`
			`>>> print(result)`
			`0000000000000000000000020002211100021111002121110011111101111111`
			`>>> print(result[30],result[37],result[38],result[45],result[36],result[43])`
			`1 1 1 1 1 1`
			The resulting board state matches the expected result and to highlight this I have further shown that every position that should have flipped has flipped to a 1.

	
	Position 9 for White
	0002222200021222002101220002102222000021020000010000000000000000
	 
		Positions 18, 27, 36 should flip.
		 
		00022222
		01021222
		00110122
		00011222
		00021102
		00000100
		00000000
		00000000
			`>>> board = "00022222000212220021012200021222000221020000010000000000000000"`
			`>>> result = flipAll("1", 9, board)`
			`>>> print(result)`
			`00022222010212220011012200011222000211020000010000000000000000`
			`>>> print(result[18],result[27],result[36])`
			`1 1 1`
			`>>>`
			The resulting board state matches the expected result and to highlight this I have further shown that every position that should have flipped has flipped to a 1.

findAllValid	
	0000000000000000000000020002202100022221002222110011111101111111
		Positions 15, 18, 19, 20, 21, 22, 26, 29, 33, 34 and 41 are valid moves.
		 
			`>>> board = "00000000000000000000002000220210002222100222211001111110111111 1"`
			`>>> correctMoves = [15, 18, 19, 20, 21, 22, 26, 29, 33, 34, 41]`
			`>>> calculated = findAllValid("1",findAllPieces("1",board),board)`
			`>>> set(correctMoves) == set(calculated)`
			`True`
			`>>>`
			The set of moves that were expected (correctMoves) is identical to the set of moves that were calculated (calculated).

Evaluation

General Reflection of Overall System

My initial project statement was 'I am designing an AI for the board game Othello (sometimes called Reversi) that a user can play against. The solution will feature a visual representation of the Othello board and allow a user to interact with it to input their moves.' I think I have comprehensively met this statement. The AI is very strong when on the hardest difficulty and unbeatable (in every match thus far) but able to perform at lower difficulties for a full range of users. The Othello board is very well replicated in my opinion and the user interaction to input their moves works well.

Overall, I think the AI system is very strong and effective and as this was my primary goal, I am happy with the project as I met it well. The main area of improvement would be in the user interface system. Given I did not set out to make the project exclusively use a graphical interface with clickable buttons, it was not something I developed during the project. However, during the process of playing many games against the AI to test its strength, I did feel that an improved system for interaction in menus would be highly beneficial. As such, this would be my main area of potential extension on the project and area of improvement to the overall user experience.

Objective Based Evaluation

Area	Number	Objective	Evaluation	Client Feedback	Possible Extension
GUI	1.1	Have an 8x8 board that can contain black or white tiles.	This objective was achieved well as it was relatively simple. The harder part was the scaling of the board and upon reflection, I should have made it so that the board remains perfectly square even when the window is resized to a non-square size as the square board looks a lot better than a stretched rectangular board.	*Q: What do you think about the playing board and discs and are there any changes you would make?* A: I think the game looks very similar to the original board game and the pieces are clear. Not sure of any changes I can think of.	Test whether always drawing the board as a square is more effective when rescaling.
	1.2	Display the winner of the match when the game is finished.	I think there was room for improvement of this area. I settled on a pop-up text window to show the winner and the score. Given I was aiming to provide the user with 'visual gratification' a more colourful or potentially animated pop up may have been relatively simple to implement but more effective.		Develop a more visually attractive graphical element to display the winner of the match.
	1.4	Have an option to allow the player to restart the match.	This objective was met as after each action the user completes, they are able to ask to play again. Potentially converting the text interface to a graphical user interface may have been a potential improvement but I don't think	*Q: Do you think that the main menu interface could be improved?* A: I think each choice is very clear and well phrased.	Develop a full GUI system for the main menu interface including a restart button.

			this adds that much to the system.	*Q: Would you prefer a graphical interface with buttons you could press instead of typing?* A: I think the text works fine, but it is easier to click a button than type the number so maybe yes.	
	2	Have a system to demonstrate to the user the number of white and black discs face up on the board in each position that updates every time a move is made.	I am very happy with the design and functionality of the sliding bar system that displays the number of pieces of each colour. I think the design looks sleek and effectively conveys all necessary meaningful information.	*Q: What do you think about the system for indicating the number of each colour discs on the board?* A: I like the design. It looks pretty refined and works well.	
	3	Have a system to visually indicate to the player the most recently placed tile by the computer by outlining the square the computer played on.	Upon reflection, I think my clients feedback about not making the system to highlight the most recent move too garish may have pushed me into making the system too subtle. However, I think the only potential change would be making the highlighting square a few pixels thicker which I could easily change but decided not to because of the client interview initially. This could be a change I make following my client interview I am going to have to evaluate the product.	*Q: Do you have any thoughts about the system for highlighting the most recent played move?* A: I think it works well most of the time but sometimes when there's a lot of pieces on the board it gets hard to spot which piece is highlighted and I have to look around the board for a couple of seconds to find it.	Tweak and refine the size and colour of the highlighting of the tiles to find a better cross between overly garish and subtlety.
	4	Have a system to indicate to the player which moves are valid and legal in the current board state by adding a semi-transparent dot in the centre of the playing square.	I specified the dot would be semi-transparent and, although it is, the effect is fairly limited. However, the system highlights every single valid move completely accurately – proven by lots of testing – so I feel I have achieved this objective	*Q: What do you think about the method of highlighting the possible moves?* A: The system works well. It is very clear and useful for playing the game.	
	8.1	Before launching the game, give the user the choice	This objective has been achieved as the user is clearly asked if they wish to play		Develop a full GUI system for the main menu

		between playing a match or reviewing one.	against the AI or review a past match from the main menu interface. However, converting this into a graphical user interface may have improved client experience.		interface for better client experience.
	8.3	Display to the user a list of all previous matches played that are available for review.	I achieved this objective but feel I could have some improvements. For example, as more and more matches are played, the list of past matches will just keep growing so some method to filter by date played may have been helpful.	*Q: Would you like to see a method of filtering the list of matches from the database?* A: I hadn't thought about that but that could be helpful especially as I play more and more matches and the list gets longer.	Implement a filtering system for the list of previous matches from the database. This could include filtering by date, outcome (if the player or AI won), AI difficulty or player versus player matches only.
	11	Allow the user to select the difficulty AI they wish to play against at the beginning of the match.	I achieved this objective but, again, this could potentially be improved by using a graphical user interface for ease of use by the client.		Develop a full GUI system for the main menu interface for better client experience.
Functionality	1.3	Flip the correct discs in accordance with the rules when a player makes their move.	Having tested the system in many example board states, the flipping system works 100% of the time. I evaluated this process during the production of the NEA identifying potential inefficiency so iterated on this design to improve it. I feel the system is in the best state it can be in python but, if a different programming language was used, I think it may be possible to make it more efficient using binary logic.		Write and compile a faster flipping function in a different programming language to be imported and used by the primary python program.
	1.5	Setup the board in the correct position (D4W, D5B, E4B, E5W) every time the match starts.	The board is always setup in this state, so I met the objective quite easily. Something that could be fun to explore would be investigating different potential starting positions and giving the user the option to choose between starting positions.	*Q: Would you be interested in being able to play Othello from different starting positions?* A: Again, that's something I hadn't thought about but that could be quite fun. You could also use it to give a player an advantage by giving them a	Give the user the ability to choose the setup at the start of the match if they wish to deviate from standard rules.

				corner or something like that.	
	6	Allow the user to click on arrows or use the arrow keys to step forwards and backwards through the match they are currently playing or reviewing.	I think I met this objective well; one potential improvement could be to add a key bind to skip to the start or end of the match. Whilst you can do this with the keypad by clicking the arrows on the screen, it is not currently possible to do with the keyboard so adding this feature could improve the usability of the system.	*Q: Would you like the ability to skip to the start or end of the match using the keyboard?* A: I don't really use the keyboard controls when playing the match, but I do in the review system and that could be helpful as long as I can remember which keys are the right ones.	Specify a keyboard key to be used to skip to the start or end of the match.
	8.2	Once a match is complete, store it in a database along with the time it was played.	I think I met this objective well. Matches are always stored with enough information to uniquely identify the match. The specific date and time played is the key factor in distinguishing between matches for the user.		
	8.4	Allow the user to input the key of a match they wish to review and load that match into the review system.	I met this objective well with each match having a unique numerical key that can be entered to review it. I am not sure how I can improve so will ask the client for feedback.	*Q: Do you think the way of entering the key of the match to review could be improved?* A: I'm not sure how it could be improved to be honest, sorry.	
	8.5	Write a match review system where the user can step through the match, seeing the evaluation and best move in each position.	The match review system functions well. Upon evaluation of my product, instead of just showing potentially multiple different good moves, it may have been helpful for the client to display the evaluation of each unique move so the user can see both bad moves and good moves.		Add small text in each tile that is a potential move showing the evaluation of that move.
	9	Allow the player to select the location of their move by clicking on the appropriate square with their mouse /trackpad/touch screen.	This system works well both with touchscreen and with mouse presses which are handled the same by PyGame. As an extension to my system, I could potentially implement a method for the user to place tiles with the keyboard using	*Q: Would you like to be able to place pieces using just your keyboard?* A: I don't think I would personally use it that much given I always have	Allow the user to navigate the playing board using the keyboard and let them place a piece by pressing a key such as 'enter'.

			the w, a, s and d keys for example.	a mouse, but I can see how some people would benefit from being able to do that.	
AI	5	Write an evaluation function to provide a numerical evaluation of a given Othello position.	I think I have met this objective well. The only issue with it is that in the initial game it suggests black has a slight advantage, but the game should be more balanced. In the middle and end game, the evaluation seems accurate and often results in a win for the AI if played on expert.		Do further research into how to evaluate the strength of early game positions and implement this into the AI system.
	7	Allow the user to request information explaining how the AI evaluation function works so they can better understand the feedback they are given.	This objective was challenging to condense sufficient information into a short amount of text clearly and succinctly. To resolve this, I implemented the API calls to the Large Language Model to allow for the user to ask more in-depth questions. This works well but has the drawback of requiring an API key which changes regularly so distributing the system would be more challenging. However, for a user who knows how to update the API key, the system is effective.		Try to find a Large Language Model which can be used easily without an API interface or with a static API key, however, this proved challenging when I initially researched potential options.
	10	Have an AI that can perform at different difficulties. The difficulty for a given match will be selected by the user before the match begins.	The range of AI difficulties provided works well with Beginner and Easy being simple to beat without much thought for me – given I have played and practiced playing Othello a lot during this project. Furthermore, medium is beatable with considered thought and hard is a significant challenge to beat. Expert difficulty is very challenging, and I am yet to personally beat it. Therefore, I feel I have succeeded in the task of creating an AI that is strong enough to be seemingly unbeatable to me and successfully created intermediary difficulties. I think the step up in difficulty between easy and medium is quite significant, but I will be interested to see how	*Q: How challenging did you find the AI system to beat, and do you think adjustments should be made to each difficulty?* A: Both beginner and easy were very easy to beat as they often gave away corners or didn't take them if I made a mistake. Medium was a lot more challenging and often took corners but sometimes played moves that let me take the corner a couple of moves later. I couldn't beat hard difficulty and	Further increases to the efficiency of components of the program could allow me to increase the depth of the minimax algorithm on expert mode leading to an even stronger AI system without compromising processing time too significantly. Currently 95% of searches take less than 3 seconds.

			challenging my client finds the different difficulties.	similarly expert won every time.	

References

Resources Used

- Semi-transparent circles in PyGame: https://stackoverflow.com/questions/51229740/how-to-draw-a-semi-transparent-circle-in-pygame
- Colour palette generator: https://coolors.co/
- MySQL database basics: https://realpython.com/python-sql-libraries/#understanding-the-database-schema
- Database and GUI design: https://www.visual-paradigm.com/

Bibliography

Anhlee. (2023, April 18). *Pinterest*. Retrieved from Pinterest: https://www.pinterest.co.uk/pin/778700591791665334/

Draughts.org. (2023, April 4). Retrieved from Play Computer: https://draughts.org/play/computer

LiChess. (2023, April 18). Retrieved from LiChess: https://lichess.org/

Othello Instruction Sheet. (2023, April 17). Retrieved from Mattel Global Consumer Support: https://service.mattel.com/instruction_sheets/T8130-Eng.pdf

Stockfish Analysis. (2023, 04 25). Retrieved from Chess.com: https://www.chess.com/