Analysis

Project Summary

In this project, I will be attempting to design and develop a successful solution to the problem of creating digital version as well as an AI for the board game known as “Othello”. Othello itself is a 2-player strategy board game that functions by the placement of black/white tokens on an 8x8 chequered grid. Certain placements result in the conversion of an opponent’s counters to the colour opposite. The ultimate aim being to control as much of the board with one’s coloured tiles by the end of the game.

Given that Othello is a 2-player board game, it cannot be played by a single person unless both sides are played by the same player. This creates a problem when a player has no board and no willing opponent to play against. Hence, I propose the creation of a digital version of the board game with an AI to replicate an opponent (and to provide a board).

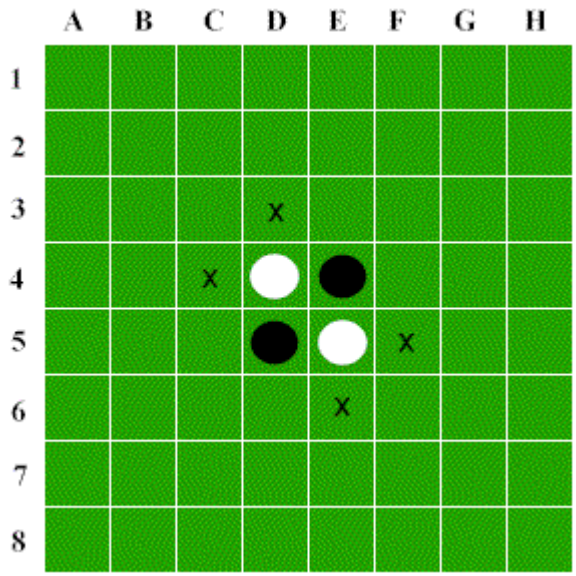
There are many different possible solutions that could be used to provide a suitable AI. One of these, a game tree, is a method for developing gaming AIs that relies on “Brute Forcing” the best possible set of moves many turns in advance in an attempt to beat the player. However, this method brings many technical challenges including the inevitably long processing time that is needed to generate the tree and search through it. This could lead to a long waiting time for the algorithm to reach a decision which would not be sufficiently appropriate for the end user. There are ways of minimising this with methods such as limiting search depth. These solutions will be further expanded on with more detail further into the document.

It will also be necessary to develop a suitable GUI for the end user. This user interface will need to actively mimic a real-life Othello board but also give the user a way to alter basic settings about how the game will be played e.g. the determination of who goes first. It also may be useful to implement a mechanism to allow for different player abilities. However, further investigation into the exact nature of the end user requirements is necessary in order to establish the necessity for this proposed feature.

Finally, I propose that the system could be made more beneficial and useful as an alternative to the standard board game by offering a basic means of supporting multiplayer games. However, due to complications and time constrains this suggestion needs to be further researched to ensure that this can be achieved in the amount of time that I have for development.

Background Research

The game that I have chosen for this project "Othello" was invented in the 1960s in Japan by Mr. Goro Hasegawa. A game that was inspired from Go and possibly Reversi. The game itself is based on a chequered grid of 8x8 squares. Tokens with a white and black side are used to populate the board. At the start of the game both players place 4 stones in the centre 4 squares in a diagonal black and white pattern. Black always goes first, and in order to make a move a player must place a token where it forms a line between it and a token of the same colour but between them must be only token/s of the opposite colour. If this is true, when the player places the tokens all the white token/s that are in this line are flipped so that they are now the opposite colour. For example, in the picture to the left, if black were to place a token at 3D then the white token at 4D would then have to be flipped so it becomes a black token. This process is followed until the point at which a player can no longer make a move, at which point the player must pass their turn and if the other player also then passes after that, the game is over. The winner is decided by the person who has the largest amount of their coloured tokens on the board. This end result can be a draw with each player ending up with an equal number of tokens on the board.

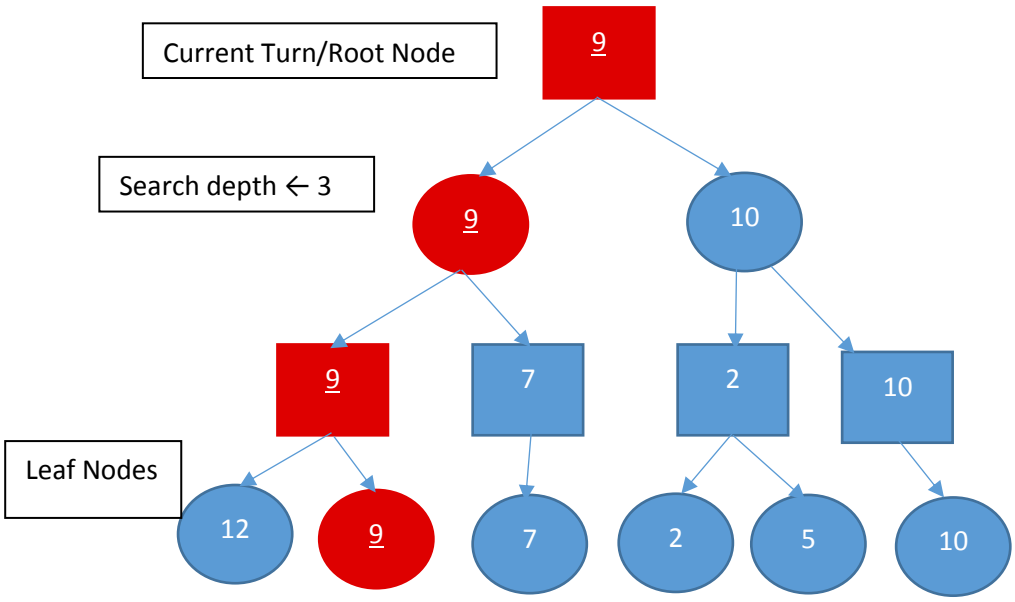


The problem of trying to implement an effective AI to Othello has been attempted since 1980 and grids of 4x4 and 6x6 have already been fully solved – meaning all possible moves that can be possibly made in an Othello game by both players in these sizes (a 6x6 Othello complete map of all possible moves can now be created in less than 4 days by most programs) have been generated and analysed to see if the game can be won by making a certain number of moves. Both versions of the game had no distinct "perfect solution" however it was noted that white gained a significant statistical advantage over black on average. However, 8x8 Othello is much more difficult due to a significant increase in possibilities (with around 10^54 nodes in its game tree). Therefore, it is necessary to find alternate solutions to the problem that are realistic. One of these techniques in Othello computing is "Disk-square tables". This is the practise of assigning each different square on an Othello board a number based on its strategic value. For example, a side piece would be considered much more valuable to the computer than a centre piece, this is because side pieces can be difficult to take and would allow for the possibility of taking a large number of tokens from the opposing player due to their position causing very large lines of enemy tokens to become available for later stages of the game. Another method is to take into account the mobility available to the player. Many Othello players will not aim for taking large percentages of tokens in the early-mid game but will instead look too minimise their tokens so that they can start taking large amounts of tokens later due to their prior positioning in the mid-late game. This would be considerably difficult to implement as the program would need some knowledge on different board configurations, although it would result in a much more sophisticated AI which could potentially beat most human players.

In order to implement any of these techniques within a computer AI, it is necessary to create data structures and an appropriate search algorithm to store and evaluate the possible moves that could be made by the human player and the computer. A common strategy to achieve this, is to use a game tree. A game tree is a method to represent all of the possible moves and board states that could occur from each moves. When using game trees, the tree is often stored as a series of linked nodes; where the nodes represent a board state and where a link represents a possible move that could be made.

In order for any AI to make a decision based on the value of one move over another, it is necessary to perform a search through the game tree in order to find the move that will give the computer the greatest advantage over the other player.

An example of a commonly used algorithm for artificial intelligence in games (especially in board games) would be Minimax. This algorithm is designed to function in a zero-sum game (meaning every time a player moves to gain a score, the opposing player will lose the same amount), and is also under the assumption that both players are undertaking the same overall strategy (to control the maximum amount of the board by the end of the game) as well as them not being able to benefit from changing their strategy (if either player chooses to not work towards winning the majority of the board they will always have a very high risk of failure). If all of these assumptions are true then Minimax can be applied.



In the diagram to the left, we can see an example of a game tree from a hypothetical game zero-sum game. The square represents player 2's turn and the circle being player 1's turn. The root node (the top shape) represents the current turn. When executing the algorithm, the first step is to work out the score of all of the leaf nodes (bottom shapes that are at the end of the current search depth) because the intermediate scores are technically irrelevant as the end scores are what will determine the outcome of the overall game (this statement becomes much more accurate whenever the search depth is increased). When the scores are calculated for all leaf nodes, the program will calculate the value of the turn above the previous level. In this case given that it is a square (indicating that it is player 2's turn), algorithm will find the move that will minimise the score of player 1. This is done because the algorithm will assume that the opponent is playing perfectly and thus will choose the move that will reduce the score of the opponent by the most. For example, on the far-left square node on the second level, we can see that it has chosen 9 instead of 12 because player 2 is attempting to minimise score of the opponent thus the algorithm is assuming that player 2 will make such a move. On alternate turns, the program will do the opposite (given that it is player 1's turn) and will choose the move that will maximise their potential score. This algorithm is repeated up till

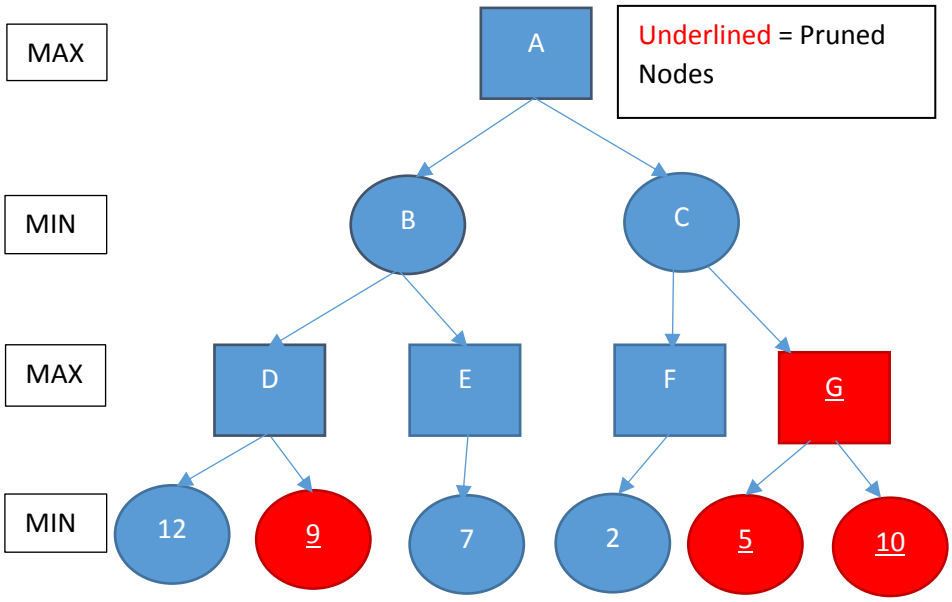
the root node where a score is calculated (in this case 9) and a path is also calculated (which has been underlined).

Unfortunately, a problem with game trees and Minimax is that the tree data structure that will be used to store the possible moves can get very large in certain scenarios during the game (especially during the mid-game, where the range of possible moves can suddenly rise dramatically at a given turn. Thus, to be able to keep waiting times short, I need to be able to find a means to make the search through the data structure to be made more efficient. A way to do this is to implement an algorithm known as “Alpha Beta Pruning”. The idea around the creation of this algorithm is that there are some moves that can be made, that a rational player would never make. Meaning that some nodes in a game tree can be “pruned” so that the search algorithm does not even need to look at these nodes as doing so would be pointless.

In this algorithm, each node has two variables attached to them one is named α and the other β. With α representing the maximum lower bound of possible solutions (so on initialisation this is set to the lowest negative number possible, usually represented by -∞). Whereas β represents the minimum higher bound of possible solutions (so on initialisation it is typically represented by ∞). In theory, this means that a node can only be considered as a valid path if its value falls within the range $\alpha \leq N \leq \beta$. As the search continues a pattern will eventually take place where the bounds for α and β come closer together until they converge. However, should this continue so that there is no longer any overlap between the them, the children of node that is currently being evaluated will be pruned.

To demonstrate this, I will again use a game tree as well as 2 variables: alpha (α) and beta (β). In this tree, there are 2 players: player 1 (known as the maximiser) and player 2 (known as the minimiser). With the square representing player 1’s move and the circles representing player 2’s turn. Each node in this tree has an α and β value.

In this case after root initialisation, the search takes the left most branch down to a node if it is not a leaf node, it will inherit the α and β values of its parent node. This is repeated until it reaches the leftmost leaf node (the nodes at the bottom of the tree with an associated value to them).



When this condition is met, the parent node of the leaf (D) will be checked to see if it is a max or minimiser. In this case it is a maximiser, so it will evaluate the leftmost leaf first by comparing its value with its current α and given that 12 > -∞ and is thus worth more than the current value for α, node D's α will be set to 12. Now, to decide if the algorithm will even look at D's other child leaf node, the algorithm will move to the node above it (a minimiser) and will change its β value to 12, not its α, this is because minimisers can only make decisions regarding β and thus can only take a value for its β from its child nodes. After this it will look at another branch to see if D node will ever be reached in gameplay. When the branch is taken, there is only 1 leaf node (thus E node is irrelevant as only 1 move can possibly be made) and it has a value of 7 and given that E's α value is still -∞, it will take it. Now the minimiser of B can take the value of E as it is the lowest value it can guarantee to occur, so β at B node becomes 7. Finally, the algorithm has enough information to see if it wants to prune a node (meaning not look at it) so given that the β value of B node is greater than the α value of D node, the minimiser node at B will not want to make a move to D as it can guarantee a worse score for the maximise by making a move to E node instead, thus there is no need to look at the leaf node with value 9 and so it can be pruned.

After this the value for α is passed up to the root node and I again compared with its value for α and given that 7 > -∞, it replaces it. So now player 1 can guarantee at least a score of 7 should the other branch prove worse off. Now the same process can be repeated for the next branch of the search. So the search chooses the path to C node (inheriting the root node's value for α and β) and again goes for the left most path to Node F and finally reaches the leaf node with value 2. Then it does not take the value for α, however the node above it C will take it as 2 < ∞. Now, given that the value for α at the root node is greater than the value for node C's β, I can now prune G node as it wouldn't matter what value G's child leaf nodes had because the minimiser would be able to guarantee a value that is less than or equal to 2 which is much less than what the root node can currently guarantee. Thus, the algorithm has searched through the tree without having to search though the while game tree.

Identification of end users.

Othello's recommended age group is around 7+ this is because the rules are simple enough for young children to understand but the overarching strategy of the game and the potential for complex decision making makes the game also accessible to a much older audience. So overall, we can say that Othello itself would often be classified as a family game. This has implications for the expectations that I will have for my program, including a range of selectable difficulties that can offer a suitable match to the ability of the user.

With this in mind, the user interface will also have to be altered so that it is simple enough to be operated by a child. To do this, each menu will be fitted with large buttons with contrasting texts to making each selection process easy for a person who may find it difficult to control the pointer. Another addition is to minimise the amount of buttons/settings necessary to allow for a fully completed product. This is vitally important as it will minimise confusion and promote accessibility to the wider family audience.

The following table is taken from Debra Levin Gelman's Design for Kids.

Table 2—Design guidelines for developmental categories

2–4 year olds	4–6 year olds	7–9 year olds	10–12 year olds
Highlight only things with which a child can interact.	Break up instructions into manageable chunks.	Clearly show goals and purpose.	Provide contextual Help after a failure.
Use few colors.	Allow invention and self-expression.	Offer a clear set of rules, with opportunities to interpret and expand them.	Emphasize self-expression and accomplishment.
Use icons that are highly representational.	Provide immediate positive feedback.	Let kids earn and collect awards.	Invite silliness or irreverence.

An important factor that needs to be taken into account when designing a game that includes an age range involving young children is the development categories. These categories describe the appropriate cognitive development stages in children. This is very important as the various needs for very small age ranges in young children can be very specific in their needs. In the case for Othello AI, the youngest age that the game can aim for would be for 7+. This can be explained by examining the table above by *Debra Levin Gelman* that depicts design guidelines for various child development stages, here we can draw several important points that may prove to be needed to successfully create a game that can appeal to our younger target audience:

- Have a clearly defined and simple rule set, whilst allowing the ability for the user to customise to form a variation of this rule set.
- Allowing the user to have a sense of accomplishment after completing a task set forward by the game.

- Possibly allowing the user to express themselves individually, by allowing to collect possible awards or achievements.
- Providing a means to give help to a struggling player or help to a player who may have made a very poor choice during a game.

Each of these points has some degree of merit to being able to be implemented into Othello AI's overall game design.

Interview

To investigate the needs of potential users I decided to interview a family member who plays Othello. The following is a transcript of this interview:

What control do you think the player should have over establishing the order of play?

Well I think it should have 2 options. You should have a manual choice or you give the computer random choice. So, some people may prefer going first, some people may prefer going second, the aggressive move to go first or the defensive move to go second, some people prefer both ways. But then to add a little bit of uncertainty and have a setting where the computer decides who goes first. To take it out your hands

What features/informative tools do you feel the app should offer to help support the player (especially new ones) while playing?

I would like to be able to rotate the board 360 degrees around so that you can view from every angle. Making it so you don't have to keep the same side of the board facing towards you for this game, so it may give you a better perspective. In perhaps a beginner's mode, a hint button. So, it highlights available moves, especially when there is a lot of counters, you may not be able to see the options so maybe for beginners there is a hint button. And, when you hover over a square, what effect that move would make could perhaps be highlighted at a beginner's level.

How wide ranging should the AI's difficulty be to make it acceptable in an app in your opinion?

Well there should be levels, there should be degrees of difficulty. Because you want someone who has never played it before to be able to play it and progress through the levels to maybe perhaps a professional level. So maybe with hints and help at the first level and as hard as it can be at the top you can progress through the game. Making it interesting to any level of player from the outset.

What degree of customisation, if any, would you like to see over the standard format of Othello?

It might be interesting to be able to have different colour backgrounds as opposed to the standard green of Othello, perhaps different colour counters. Different size grids than the 8x8 smaller perhaps.

I'd want to see if there was some form of effect if you cleared a long roll, some feature like that. That's about it.

Analysis of interview

From this interview we can establish a list of several suggestions that can be utilised to help me solve the problem. One, offering the user to decide the method of which the order of play is established, either by a "manual" method where the user can pick which colour they choose to play as or by implementing a way to randomise the this outcome where a player doesn't mind to play with both. Second, the mention of the concept of a "hint" button that could be used to help brand new players learn to make beneficial moves could perhaps be implemented. The ability to use such a button a limited amount of turns to highlight all possible moves, as well as to be able to hover over a square to show the effects this move would have on the board may be possible. Third, (although previously noted for importance before) the interviewee did stress the need to have a wide range of available difficulties to account for Othello's wide reaching audience, this of course is an essential part to our program and could be easily implemented with an AI but with also the addition of the pre-mentioned "hint button" for the lowest of difficulties. Fourthly, the interview did demonstrate that a basic degree of customisation could also be important enough to consider. Examples include utilising different shaped 6x6 or even 4x4 grids to offer a quicker game for the user to play, or even the ability to change colours of tokens, although customisation should be stressed to be of lower priority to the rest of the mentioned features.

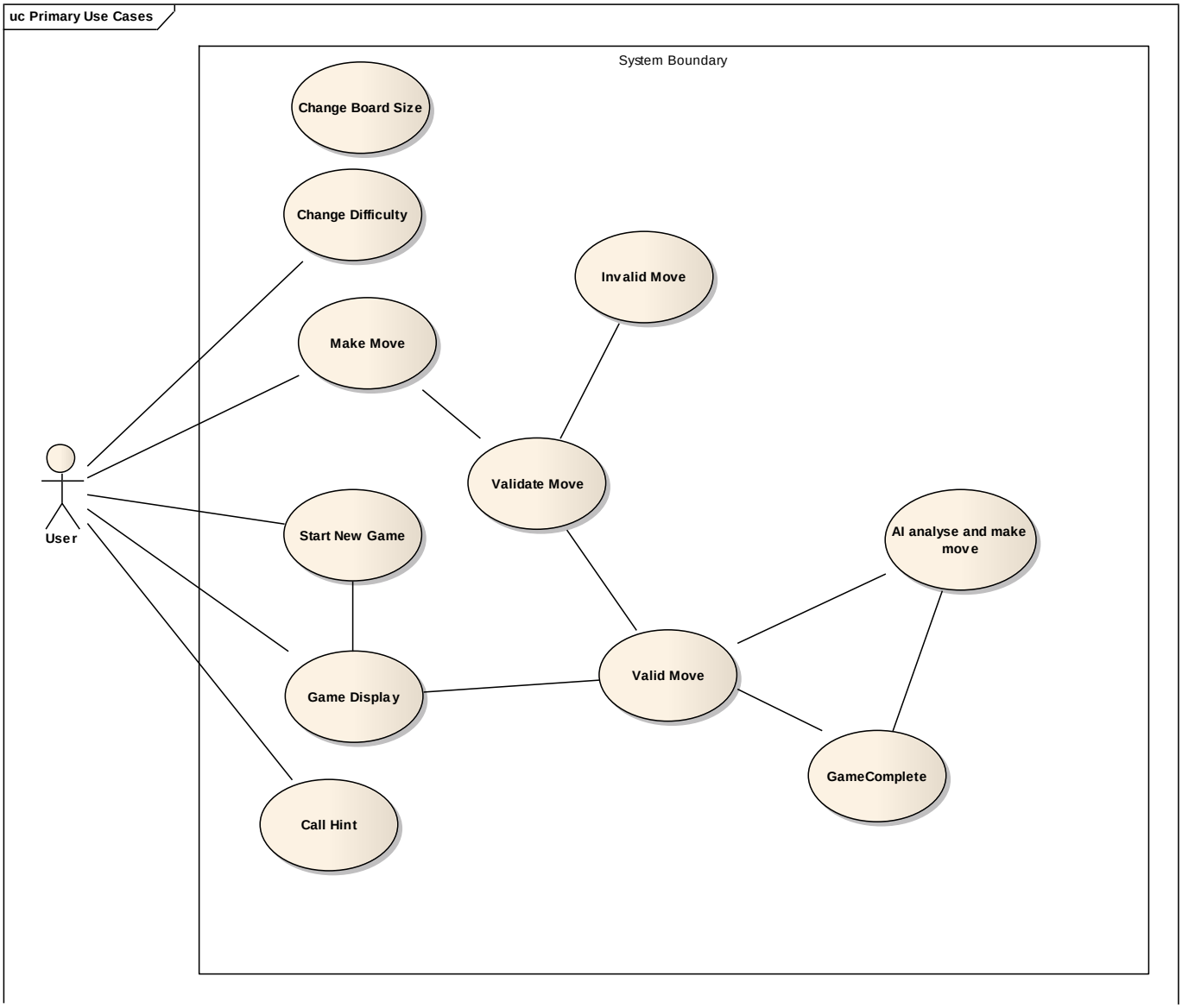
Interesting features mentioned in this interview can be summarised in the following bullet points:

- An option to choose a manual or random method of selecting the initial order of play.
- A range of difficulties to split up the abilities of different users so that they can find a level that they enjoy to play.
- A possible "hint" button that is shown in the lower levels of difficulty to visualise where the possible moves that the user can make.
- A few basic methods of customisation including being able to alter the size of the board and the colour of the tokens (low priority compared to other suggestions, will be subject to availability of time)
- Possible rotation of the board (again low priority, however the effect of rotating the board even by 180^{o} may still be negligible as the board will most likely be visible from a bird's eye view, making it in most cases unnecessary).

Project Objectives & Use Cases

Main Program

The following diagram details the main, high-level, use cases required.



Start new game

From the perspective of the user this case means that by activating it, the program will ask for their chosen colour, then after a few seconds the game will begin and allow the user to make a move. For this to work from a technical perspective, the program needs to be able: to generate a new board (as well as committing this to memory), initialise its graphical representation on the user interface and create the first game tree/data structure for the AI to utilise.

In a summary of objectives, the program must be able to do the following at this stage:

- Either ask the user to select a colour before the game launches, or offer a random mode of selection for the selection of colours.
- Construct the first data structure that the AI will use to search through later.
- Can render a graphical interface onto the screen.
    - Which should include (but not limited to), the board, a hint button and a scoreboard to show the number of tokens taken on each side and an object to display the player of the current turn.
- Construct the necessary objects from the necessary classes within the program.

Change Difficulty

When the user initialises this use case they will expect to be able to limit/enhance the AI's ability to analyse and play the game. By the end of the project, the finished program must be able to offer at least 2 selectable difficulties to the user. In practise when this event is called before the start of the game, I will expect to alter the difficulty of the AI by limiting its search depth. What this means is that it will limit the amount of turns the AI will analyse ahead of the current turn with the use of a game tree.

In a summary of objectives, the program must be able to do the following at this stage:

- At least 2 selectable difficulties must be present within the game.
- Any change in difficulty must affect the creation of the main data structures that the AI has to search through thus limiting/enhancing the AIs ability to perform its algorithm.
    - When in relation to a game tree, the search depth must vary due to change in difficulty.

Make (user) a move

When the user makes a move, from their perspective they will choose a square on the board in the GUI and then the program will display changes made to the board. The program should also be able to pass the players turn if they have no legal move to make.

To achieve this the program will need to be able to record the position that the player has chosen on the board and call a function to validate the move made. Finally, the program will need to update the current board state and update the AI game tree.

In a summary of objectives, the program must be able to do the following at this stage:

- Before the user makes a move, the program must check to see if the player can make a move.

Validate movement

Whenever a player makes a move, they will expect the program to prevent them from making an illegal move (defined as a move that breaks the rules of Othello). To do this I expect the creation of a method that will compare the position given by the user to the current board and work out if it breaks the 3 rules of placement of Othello being:

- The space chosen must be empty.
- The space chosen must be adjacent to another token (a token can be adjacent diagonally).
- And by placing a token at this location, at least one token from the opponent is switched to being a token of the current players colour.

In a summary of objectives, the program must be able to do the following at this stage:

- When the user makes a move, a validation function must be called to analyse the move made.
- The finished program must be able to detect an illegal move made by the player and prevent them from making it.
    - The program must be able to check the square chosen by the player to see if it is occupied.
    - The program must be able to search through all adjacent spaces to detect for a token of the opposite colour
    - The validation section of the program also should be able to detect all of the tokens of the opposite colour that will be turned over and ensure that at least 1 is changed to keep consistent with the rules of Othello.
- The validation function must also be able to handle moves that are made close to the edge of the board.

Valid Move

When a valid move is made the user will expect at least 2 or more other squares on the board to switch and become the token of their assigned colour. For this to be achieved the valid move must be simulated with a function to determine what effect it has on the board and then data structure storing the current board must be updated with the changes that were found to occur.

In a summary of objectives, the program must be able to do the following at this stage:

- The program must have a function to simulate a valid move to find out all the effects that it has on the current board.
- The program must be able to utilise a data structure to store the current board.

Game Display

When this case is activated the user would expect to see a GUI depicting the main menu of the game where they will have the ability to alter the difficulty of the next game they play. When a new game

is started, the GUI should switch to a new form that shows a graphical representation of the game board that the user can interact with to make a move, along with a scoreboard to show the current number of tokens on each side, and an object to show which player has the current turn. They will also expect the scoreboard showing the current number of tokens for each player to change and for the object displaying the player of the current turn to switch to showing that it is the opponents turn. In order for this to function I will need to create a form upon starting the game that will render the starting board and will change whenever the appropriate class messages it with the necessary changes caused by either the user or AI making a move.

In a summary of objectives, the program must be able to do the following at this stage:

- On start-up, the program should be able to generate a GUI showing a main menu giving the user the ability to change difficulty and initiate a new game.
    - When a new game is started the GUI should switch to a different form with a board and a text box to display the current score next to the board.
- The graphical representation of the board will also be used as the user interface for making moves by simply clicking on the square that the user wants to place a token at.
- The program must be able to update the scoreboard at the end of each turn to reflect the current score of each player (determined by the number of tokens on each player side).
- The program must be able to update the graphical board to show the board that is currently being held within memory.
- The program must be able to update a text box at the end of each turn to show the player who has the current turn.

Game completion

Upon the game finishing (when neither player can make a valid move), the user will expect the program to display the winner of the match (or be able to tell if the match is a draw). In order to do this, the program will count the number of each token and discern which player has more tokens and display the winner with either a message box or form, and in the case of a draw, be able to display similarly that the game ended as a draw with no winner.

In a summary of objectives, the program must be able to do the following at this stage:

- It must be able to display a message box or form to show the winner of the match.
- To be able to check for a win condition on each turn.

Change Board Size

Before the game, the program will be able to offer a degree of customisation. Specifically, the product should allow the user to alter the size of the board, picking from a selection of 3 different sized grids: 8x8 (set as the default), 6x6 and 4x4. The program needs to be able to adjust its user interface as well as its mechanism to store the game board whenever this configuration is altered.

In a summary of objectives, the program must be able to do the following at this stage:

- It must be able to offer a setting for the user to choose a different board size from a selection of an: 8x8 grid (set as the default), 6x6 grid and a 4x4 grid.

- Whenever this setting is changed, the data structures used to store the board as well as the user interface should alter to accommodate the new size of the board.

Call Hint

During the lowest difficulty, the user should have the ability to click on a hint button on the user interface. By pressing this button, it will highlight to the player all the possible moves that can be made on the user interface for that turn only. However, this button will have a limited amount of uses giving new players to begin to develop their own skills so they can find it easier to tackle harder difficulties without the button.

In a summary of objectives, the program must be able to do the following at this stage:

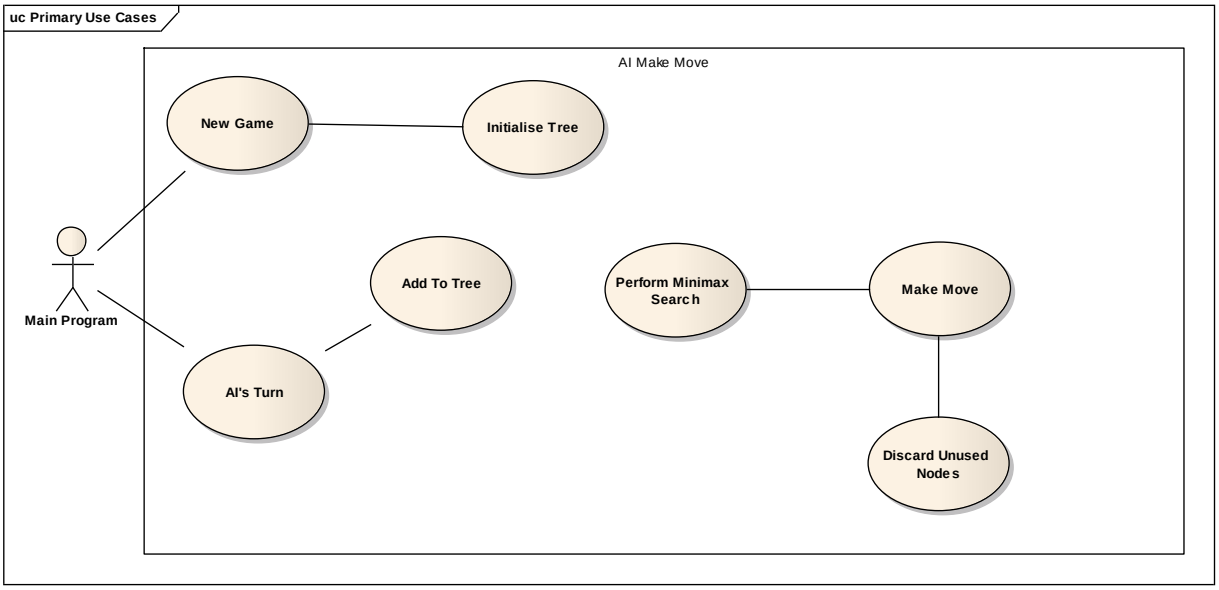
- When the difficulty is set to its lowest, a hint button should appear within the user interface during a game.
- When this button is clicked, the user interface should highlight all the possible moves the player can make on the screen for them for that turn only.
- This button will have a limited amount of uses each game, and a player will only be able to use this button 3 times per game.

AI Analyse and Make Move

When the user finishes their move, they will expect the computer to mimic a human player move. To perform this, I expect my program to be able to utilise a game tree and an efficient search algorithm to find the optimum path a couple turns ahead in order to make the best move. I also expect the algorithm to choose one move over another by calculating an arbitrary value for each move and then be able to compare this value with other nodes in the game tree to find the best move to ensure a greatest chance of victory for the computer.

Artificial Intelligence

The Following is a more detailed use-case diagram to analyse in much greater depth into the “AI Analyse and Make Move” use-case.



(New Game has already been addressed in the previous diagram– and is here just for reference)

Initialise Tree

When the main board is created, the program needs to initialise the data structure that will be used to store the game tree.

In a summary of objectives, the program must be able to do the following at this stage:

- When a new game is created, the program must be able to initialise the data structure that will be used to store the game tree.

Add to Tree

At this stage, the program will be needing to update the pre-existing game tree. To do this the program first checks to see if it is the first move of the game, if this is true then the program will create the first root node. If either this is not the case or if the previous step has been done, the program then needs to be able to find the nodes that are currently leaf nodes and generate their children by simulating all possible board placements up until the search depth is reached.

In a summary of objectives, the program must be able to do the following at this stage:

- On the AI’s first turn, the program should be able to create the first “root” node and generate the rest of the tree to the required search depth (being 2).
- At ever other turn for the AI where it can still make a move and/or the game has not been won/lost, the program must be able to generate the next child nodes of the current leaf nodes and attach it to the game tree.

Perform Minimax Search

When the tree has been fully updated (when the search depth has been reached), the program will then begin to decide on the optimum move it can make on the current turn. To do this it will begin a search though the game tree to find the choice that will both bolster the best chances of obtaining a high score many moves into the future. The type of search that will be used is a Minimax algorithm along with Alpha Beta Pruning to minimise the need to search through certain nodes. For this to function, whenever a leaf node is found for its search, a static evaluation must be performed where an arbitrary value is generated for it that represents its "score". This will be worked out by combining: the number of tokens controlled by the AI, and the number of strategic positions that the Ai also controls.

In a summary of objectives, the program must be able to do the following at this stage:

- The finished program need to be able to perform a full Minimax search each turn to find the best possible move.
    - This algorithm will also be supported by Alpha, Beta pruning to minimise the search time.
- Each node in the game tree should have an associated "value" that will be calculated by several factors such as:
    - Number of tokens belonging to the AI.
    - Number of strategic positions (such as corner pieces and side pieces, with corner pieces having a greater value than normal side ones).
- The Value of strategic position should decrease as the board is increasingly filled, this should be done according to a factor which should be determined by experimentation.

Make Move

When a path is chosen and the AI makes a move, it becomes necessary to pass on the designation as the "root node" to the chosen child that now reflects the current board state. This is done so that it is easier for the program to trim parts of the tree that are no longer connected.

In a summary of objectives, the program must be able to do the following at this stage:

- When the AI makes a move, the previous "root" node will have its "root" designation revoked.
    - The selected child that was moved to will now inherit the designation of "root node".

Discard Unused Nodes

At the end of every turn for the AI, all the nodes that are now no longer connected to the new root node, and thus are no longer possible to reach, must be deleted from the data structure. This will then free up space for the generation of new nodes on the next turn.

In a summary of objectives, the program must be able to do the following at this stage:

- At the end of the turn for the AI, all nodes that can no longer be reached will be deleted.

Modelling

Prototypes

Here is some pseudocode to demonstrate the use of the Alpha Beta Pruning algorithm (improved Minimax):

The following shows some pseudocode that demonstrates an implementation of the Minimax algorithm with alpha beta pruning:

```
FUNCTION Search (α, β, Tree, Node):
    IF Node has NO children:
        RETURN Node.Value
        #Returns value of leaf node
    IF Node is Maximiser:
        Children = Tree.FindChildren (Node)
        #Find all children
        FOR EACH Child:
            Value = Search (α, β, Tree, Child)
            IF Value > α:
                α = Value
                IF α >= β:
                    #Prune Rest of Search
                    RETURN α
        RETURN α
    IF Node is Minimiser:
        Children = Tree.FindChildren (Node)
        #Find all children
        FOR EACH Child:
            Value = Search (α, β, Tree, Child)
            IF Value < β:
                β = Value
                IF β <= α:
                    #Prune Rest of Search
```

RETURN β

RETURN β

This short recursive function is designed so that it will return the best value that the player can hope to achieve, assuming both players played perfectly. It works by working itself down the tree until it accounts the leaf nodes, then it begins to alter β and α values as it moves up until it finds the optimum α (for the players actual turn) and giving them their best score assuming the scenario of perfect play from both sides.

To demonstrate the algorithm that can be used to validate a move made by the player, I have created this prototype (written using the python language with object orientated style):

```
class BoardState:
    def __init__(self):
        self.listPresenseBoard = [0 for i in range(0,7)]
        self. listColourBoard = [0 for i in range (0,7)]
        listByte = [1, 2, 4, 8, 16, 32, 64, 128]
        pListBase = self.listPresenseBoard
        cListBase = self.listColourBoard
        cListBase[3] = 8
        cListBase[4] = 16
        pListBase[3] = 24
        pListBase[4] = 24
        self.listPresenseBoard = pListBase
        self.listColourBoard = cListBase



    def GetToken(self, intX, intY):
        if (x > 7 or x < 7) or (y > 7 or y < 7 ):
            return "o"
           #out of bounds
        if (self.listPresenseBoard[3] << 7-intY) // 128 == 1:
            if (self.listColourBoard[3] << 7-intY) // 128 == 1:
                return "w"
            #white
            else:
                return "b"
            #black

        else:
            return "e"
        #empty
```

```
class Game:

    def __init__(self, Board):
        objectBoard = Board

    def ValdateMove(self, intX, intY, stringColour):
        #is the space occupied?
        if objectBoard.GetToken(intX, intY) != "e":
            return False
        IsValidMove = False
        for x in range(intX-1, intx+1):
            for y in range (intY+1, intY - 1):
                if x == intX and y == intY:
                    pass
                else:
                    if stringColour == "b" and objectBoard.GetToken(x, y) == "w":
                        #furthersearch
                        Xvector = x - intX
                        Yvector = y - intY
                        if CheckVector(intX, intY, Xvector, Yvector, "b") == True:
                            IsValidMove = True
                    elif stringColour == "w" and objectBoard.GetToken(x, y) == "b":
                        Xvector = x - intX
                        Yvector = y - intY
                        if CheckVector(intX, intY, Xvector, Yvector, "w") == True:
                            IsValidMove = True
                        #furthersearch
                    else:
                        pass
        if IsValidMove == True:
            return True
        else:
            return False

    def CheckVector(self, intX, intY, Xvector, Yvector, Colour):
        newPos = objectBoard.GetToken(intX+Xvector, intY+Yvector)
        if newPos == "e" or newPos == "o":
            return False
        elif newPos == Colour:
            return True
        else:
            return self.CheckVector(self, intX+Xvector, intY+Yvector, Xvector, Yvector, Colour)

Board = BoardState()
CurrentGame = Game(Board)
```

This prototype utilises the data structure that will be used to store the individual boards, this is done by a single list of integers where the integers are bit maps (8 bits long, equal to the length of the board). Furthermore, this also demonstrates an algorithm which can be used to validate a player move by searching around the chosen square to see if the Othello rules of placement have been violated.