CS 201 Spring 2019 Portfolio Project Proposal

Catherine Kosek

Project Choice: Option 2 – Custom Game – Othello

**Questions and Clarifications on Instructions**

* I would like to design an Othello game for my project. As this is not one of the suggested options, I first need approval to move forward with this game. If you are not familiar with the game, I describe gameplay and rules beginning on the next page.
* Othello must be played by two people, but the instructions for the portfolio specify that “single” must be an option for game mode. For this game, can the “single” game mode be omitted so that the only options are “double” and “playing against the computer”?
* How intelligent should the AI for “playing against the computer” be? I detail my suggested AI for “easy” and “regular” gameplay mode in this proposal, but Othello computers exist that can beat even the most dedicated players through game trees and the Minimax algorithm. My research indicates an AI of this capacity is only necessary to beat very experienced players, and my “regular” gameplay mode should present a sufficient challenge to a casual player. Are my current AI goals enough for this project, or should I pursue a more ambitious computer mode?
* The other games all are of “arbitrary board size”. A standard Othello board size is an 8x8 grid. Should there be an option for the user to select a board of a different size?
* I have included discussion and pseudocode for the activities within the program that I think will be the most difficult to implement. Any feedback or suggestions on the design of these components is much appreciated.

**Rules and Specifics of the Game**

* The board is an 8x8 grid.
* There are two players, one with black pieces and one with white pieces.
* The game begins with two pieces of each color diagonal from each other in the center four grid locations, as shown below.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  | w | b |  |  |  |
|  |  |  | b | w |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

* The player with black pieces moves first.
* The player with black pieces must place a black piece on the board such that there is a horizontal, vertical, or diagonal line between the new piece and an existing black piece, and there must be one or more contiguous white pieces in between the two black pieces. Legal moves for a black piece are denoted with asterisks below.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  | \* |  |  |  |  |
|  |  | \* | w | b |  |  |  |
|  |  |  | b | w | \* |  |  |
|  |  |  |  | \* |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

* The moves for a black piece denoted by an asterisk and shown below are not legal because there are no white pieces between the black pieces, or the white pieces are not contiguous.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  | w | b |  |  |  |
|  |  |  | b | w |  |  |  |
|  |  |  | \* |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  | \* |  |  |
|  |  |  | w | b |  |  |  |
|  |  |  | b | w |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | \* |  |  |  |  |
|  |  |  | w |  |  |  |  |
|  |  |  |  | b |  |  |  |
|  |  |  | w | w |  |  |  |
|  |  |  | b | w |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| \* |  |  | w | b |  |  |  |
|  |  |  | b | w |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

* After the black piece is placed, all white pieces in between the two black pieces are “flipped” to white. A before and after example is shown below, with the move and effected piece indicated by asterisks.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  | \* |  |  |  |  |
|  |  |  | w | b |  |  |  |
|  |  |  | b | w |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  | b |  |  |  |  |
|  |  |  | b\* | b |  |  |  |
|  |  |  | b | w |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

* Now the turn of the player with the black pieces is over. It is now the turn of the player with the white pieces.
* The player with the white pieces makes move similarly to the black pieces. The player with white pieces must place a white piece on the board such that there is a horizontal, vertical, or diagonal line between the new piece and an existing white piece, and there must be one or more contiguous black pieces in between the two white pieces.
* Players continue making moves back and forth in this way.
* If a player cannot make a move, the player’s turn is skipped.
* If both players cannot make a move, the game is over, regardless of if the board is filled.
* If the board is filled, the game is over.
* Once the game is over, the winner is the player with the most pieces of their color on the board.
* It is possible for there to be equal numbers of black and white pieces at the end of the game. This is considered a tie, and the game is still over.

**Activities of the Program**

What will I need to make this program do to successfully play this game?

* Display
  + menu options, board, scores, instructions, etc.
* User Input
  + Read user input
  + Check input validity
  + React appropriately
* Gameplay
  + Check to see if game is over
    - Are there any valid moves left for the player whose turn it is?
    - If not, are there any valid moves left for the other player?
    - Either skip turn or end game.
  + Place piece on the board
    - Add new piece
    - Flip any pieces that need to be flipped by addition of new piece
  + Keep track of score
    - Keep track of wins across consecutive games
    - Keep track of current score, the number of pieces each player has on the board
  + Easy Mode Against Computer
    - For each valid move, calculate the number of pieces that would be flipped
  + Regular Mode Against Computer
    - For each valid move, compare the value assigned to that square to find the highest value move.

**Concept: User Interface Design**

* ASCII interface
* Primary Menu

OTHELLO

Welcome to Othello!

At any point, enter “main” to return to this screen.

Enter “next” to move to the menu.

* + Input Options (persist throughout program):
    - main – display primary menu
    - back – display previous menu
    - \*any invalid input\* - display error message containing list of valid inputs for that menu
  + Input Options (this menu):
    - next – move to secondary menu
  + Invalid Input Display

OTHELLO

That input is invalid.

Enter “next” to proceed to the menu.

* Secondary Menu

OTHELLO

Please choose game mode.

Enter “double” to start a two-player game.

Enter “computer” to play as a single player against the computer.

* + Input Options:
    - double – display gameplay instructions
    - computer – display computer menu
  + Invalid Input Display

OTHELLO

That input is invalid.

Enter “main” to return to the home screen.

Enter “double” to start a two-player game.

Enter “computer” to play as a single player against the computer.

* Computer Menu

OTHELLO

You have selected to play against the computer.

Please select a level of difficulty.

Enter “easy” or “regular”.

* + Input Options:
    - easy – display gameplay instructions
    - regular – display gameplay instructions
  + Invalid Input Display

OTHELLO

That input is invalid.

Enter “main” to return to the home screen.

Enter “easy” or “regular” to select a level of difficulty.

* Gameplay Instructions

OTHELLO

The board will be displayed as below.

8 \* \* \* \* \* \* \* \*

7 \* \* \* \* \* \* \* \*

6 \* \* \* \* \* \* \* \*

5 \* \* \* w b \* \* \*

4 \* \* \* b w \* \* \*

3 \* \* \* \* \* \* \* \*

2 \* \* \* \* \* \* \* \*

1 \* \* \* \* \* \* \* \*

a b c d e f g h

The numbers and letters along the side of the grid are used as identifiers for rows and columns.

An asterisk \* represents any empty square.

A w represents a square with a white piece played on it.

A b represents a square with a black piece played on it.

Players take turns placing pieces, with black going first.

A piece of one color must be placed on the board such that there is a horizontal, vertical, or diagonal line between the new piece and an existing piece of the same color, and there must be one or more contiguous pieces of the opposite color between these two pieces.

To place a piece, enter the coordinates of a valid square. For example, to place a black piece in the sample board above in the square above the top-left white piece, enter “d6”.

Enter “next” to start gameplay.

* + Input Options:
    - next – display gameplay for mode selected earlier
  + Invalid Input Display

OTHELLO

That input is invalid.

Enter “main” to return to the home screen.

Enter “next” to start gameplay.

* Gameplay

OTHELLO

Wins: Player One – 0 Player Two – 0

Score: black – 2 white - 2

Current Turn: Player One (black)

8 \* \* \* \* \* \* \* \*

7 \* \* \* \* \* \* \* \*

6 \* \* \* \* \* \* \* \*

5 \* \* \* w b \* \* \*

4 \* \* \* b w \* \* \*

3 \* \* \* \* \* \* \* \*

2 \* \* \* \* \* \* \* \*

1 \* \* \* \* \* \* \* \*

a b c d e f g h

* + Input Options:
    - \*coordinates\* - redisplay gameplay, with changes made from placing new piece
  + Invalid Input Display – coordinates indicate a space that is not a valid move

OTHELLO

That is not a valid move.

A piece of one color must be placed on the board such that there is a horizontal, vertical, or diagonal line between the new piece and an existing piece of the same color, and there must be one or more contiguous pieces of the opposite color between these two pieces.

Enter “main” to return to the home screen.

Enter coordinates of a valid square to place a piece.

* + Invalid Input Display – not coordinates

OTHELLO

That input is invalid.

Enter “main” to return to the home screen.

To place a piece and take your turn, enter coordinates, like “d6” or “g2”.

**Implementation Psuedocode: Check for Valid Moves**

This action must be performed at the beginning of every turn to determine if a turn should be skipped or if the game is over. Stores a list of coordinates of squares that could be played as a valid move into an array.

/\* This method looks at every square on the board. If the square on the board has a piece of the color for the current turn, it calls the findMoves method 8 times, one for each direction a connection could be made between the piece played and an existing piece. The findMoves method returns a coordinate, if it exists, of a square where a piece could be placed as a valid move. These coordinates are stored in an array of validSquares.

turnColor is the color of the piece that is about to be placed.

\*/

checkForMoves(turnColor)

array validSquares

for every square on the board

if pieceColor == turnColor

validSquares += findMoves(x + 1, y, 1, 0, turnColor, false) // right

validSquares += findMoves(x + 1, y + 1, 1, 1, turnColor, false) // right up

validSquares += findMoves(x, y + 1, 0, 1, turnColor, false) // up

validSquares += findMoves(x - 1, y + 1, -1, 1, turnColor, false) // left up

validSquares += findMoves(x - 1, y, -1, 0, turnColor, false) // left

validSquares += findMoves(x - 1, y - 1, -1,-1, turnColor, false)// left down

validSquares += findMoves(x, y - 1, 0, -1, turnColor, false) // down

validSquares += findMoves(x + 1, y - 1, 1, -1, turnColor, false) // rt down

// add some sort of check to make sure null values aren’t added to validSquares, or remove nulls

/\* This method is initially called on a piece next to a piece of the turn color. From there, it moves across the board in the direction indicated by xChange and yChange, stopping if an empty square, the outside edge of the board, or a square of the same color is reached.

This method is searching for an empty square with at least one square of the opposite color between the existing piece and the empty square.

If a piece of the same color or the outside edge of the board is reached before an empty square is reached, there is not a valid move in that direction, so no coordinate is returned.

If an empty square is adjacent to the existing piece, it is not a valid move because there are no pieces of the opposite color between the existing piece and the empty square. The parameter opposite is set to true once at least piece of the opposite color is visited.

If an empty square is found and opposite is true, meaning at least one piece of the opposite color exists between the existing piece and the empty square, then the coordinates of the empty square are returned.

Use of the paramenters xChange and yChange allow the direction of movement to be determined without having to create a separate method for each direction.

\*/

findMoves(x, y, xChange, yChange, turnColor, opposite)

squareColor = getColor(x,y)

if squareColor == turnColor OR squareColor == outOfBoard OR (squareColor == empty AND opposite == false)

return null

else if squareColor == empty AND opposite == true

return x,y

else // squareColor == opposite color

findMoves(x + xChange, y + yChange, xChange, yChange, turnColor, true)

**Implementation Pseudocode: Flip Pieces**

For a move to be valid, at least one piece of the opposite color must be flipped. Therefore, pieces are always flipped after every turn.

/\* This method, used for readability purposes, calls the flipPieces method 8 times, one for each direction a connection could be made between the piece played and an existing piece, and flips pieces that should be flipped.

x and y represent the coordinates on the board, and turnColor represents the color of the piece being played

\*/

flip(x, y, turnColor)

flipPieces(x + 1, y, 1, 0, turnColor, false) // moving right

flipPieces(x + 1, y + 1, 1, 1, turnColor, false) // moving right and up

flipPieces(x. y + 1, 0, 1, turnColor, false) // moving up

flipPieces(x - 1, y + 1, -1, 1, turnColor, false) // moving left and up

flipPieces(x - 1, y, -1, 0, turnColor, false) // moving left

flipPieces(x - 1, y - 1, -1,-1, turnColor, false) // moving left and down

flipPieces(x, y - 1, 0, -1, turnColor, false) // moving down

flipPieces(x + 1, y - 1, 1, -1, turnColor, false) // moving right and down

setColor(x, y, turnColor)

/\* This method is initially called on a piece next to the piece placed on the board. From there, it moves across the board in the direction indicated by xChange and yChange, stopping if an empty square, the outside edge of the board, or a square of the same color is reached.

If an empty square or the outside of the board is reached before a piece of the same color is found, no pieces should be flipped, so false is returned.

If a piece of the same color is reached, then all pieces previously called in the recursive stack should be flipped, so true is returned.

The parameter flip is false when this method is initially called, but it is set through the recursive calls. It determines whether pieces should be flipped, and it is returned by default to carry the same value, whether true or false, throughout the recursive stack.

Use of the paramenters xChange and yChange allow the direction of movement to be determined without having to create a separate method for each direction.

\*/

flipPieces(x, y, xChange, yChange, turnColor, flip)

squareColor = getColor(x,y)

if squareColor == empty OR squareColor == outOfBoard

return false

else if squareColor == turnColor

return true

else // squareColor == opposite color

flip = flipRight(x + xChange, y + yChange, xChange, yChange, turnColor, flip)

if flip == true

setColor(x, y, turnColor)

return flip

**Concept: Check to Skip Turn or End Game**

* A player might not be able to make any moves on the board, either because the board is full or because the remaining open squares are not valid moves.
  + In this situation, their turn is skipped. The other player takes another turn, and if moves for the original player become available after the other player’s turn, the original player takes their turn like usual.
* If neither player can make any moves on the board, the game is over. The board does not have to be full for this happen, the remaining open squares might just not be valid moves.
* To determine if a turn should be skipped, run checkForMoves method. If the array returned is empty, there are no valid moves for this player.
* The skipped turn should be noted in the system, and checkForMoves should be run again, but for the second player.
* If only the first player’s turn is skipped, then display a message informing the first player and proceed with the second player’s turn.
* If both players do not have valid moves, the game is over. Display a message with the final score and prompt the user to either quit or play again.

**Concept: Play-Against-Computer AI**

* After selecting to play against the computer, the user can choose between “easy” or “regular” gameplay modes.
* Easy – Maximum Disc Strategy
  + This strategy is the most intuitive and mimics how a new player would play the game. The strategy is simple: make the move that flips the maximum number of disks.
  + However, this method of gameplay is ineffective. Flipping more disks, especially early in the game, simply offers your opponent more options for strategic gameplay. Also, disks flipped early in the game and in the center of the board are not stable, and therefore do not matter.
  + Providing an easy mode of gameplay allows a new user to learn the game and experiment with different strategies.
  + Implementation of this AI would take advantage of the already existing checkForMoves and findMoves methods. The findMoves method would keep track of the number of pieces of the opposite color found by keeping track of the number of recursive calls. This value would be returned along with the x and y coordinates of the square with the valid moves. The maximum value could easily be found from within the validSquares array.
* Regular Gameplay – Weighted Square Strategy
  + This strategy recognizes that certain squares are more strategically valuable than other squares, so each square has a value assigned to it. The different types of squares, in order from least valuable (worst) to most valuable (best) are:
    1. X-Squares – squares diagonally adjacent to a corner
    2. C-Squares – squares vertically or horizontally adjacent to a corner
    3. Danger Zone – squares in the rows just inside the edge rows
    4. Internal Squares – squares in the center of the board
    5. Internal Edges – center two squares of each edge
    6. Internal Corners – squares diagonally adjacent to the X-squares
    7. Edges – squares between the internal edge squares and C-squares
    8. Corners – four corner squares, the most desirable in the board

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 2 | 7 | 5 | 5 | 7 | 2 | 8 |
| 2 | 1 | 3 | 3 | 3 | 3 | 1 | 2 |
| 7 | 3 | 6 | 4 | 4 | 6 | 3 | 7 |
| 5 | 3 | 4 | 4 | 4 | 4 | 3 | 5 |
| 5 | 3 | 4 | 4 | 4 | 4 | 3 | 5 |
| 7 | 3 | 6 | 4 | 4 | 6 | 3 | 7 |
| 2 | 1 | 3 | 3 | 3 | 3 | 1 | 2 |
| 8 | 2 | 7 | 5 | 5 | 7 | 2 | 8 |

* + Of all squares available to play on, the square with the largest value assigned to it is chosen.
  + Like the Maximum Disk Strategy, implementation of this AI would take advantage of the already existing checkForMoves and findMoves methods. When a square that allows a valid move is found, the weighted value at that coordinate is also identified and stored along with the coordinate. Once all valid moves are found, the weighted values of each move are compared, and the highest value is the move taken.