Final Project: CMPT 310

Jasper Quan: 301255149
Fang Chi (Amanda) Chang: 301278767

Implementation

For the game *Reversi* we decided to make a class that encompasses all board objects, as well as all the functions that are needed for the game. We chose the board to be a 2d-array, since the board is a static size and we are just manipulating the values stored in each index (-1-empty, 0-black, 1-white).

Initial startup of the program initializes a 8x8 board of reversi, with the initial 4 middle tiles preset to their designated color. The main functions that are called throughout the program are *play(), update(), valid_moves(), gameOver()*, this is excluding the A.I. heuristic functions.

GameOver() is a function that checks each turn whether the game is over or not, there's only two cases that would cause a game to be over; the board is filled completely, or both players cannot make a valid move.

Play() is a function that takes in 3 arguments, 2 of which are the grid coordinates of the tile you want to play on, the 3rd is a boolean value that represents whose turn it is. Since there are only two players, *True* or *False* is just a simple way to switch turns.

Update() is similar to *Play()*, taking in the same 3 arguments. The function takes the position that is played, and based off the color, check all 8 directions. If a tile of opposite color is found, continue in that direction until either you encounter your own color or an empty tile. If you encounter an empty tile, then the direction has not been "bordered". If you run into your own color, that means everything between my current location and the one I just encountered have tiles of opposite color, therefore I should flip them to be my own. There are some restrictions based on which column and row we are currently updating, for instance if I was on the top row I do not need to check the tiles above me since there are no tiles.

Valid_moves() uses the exact same logic as *Update()*, however it only takes in 1 argument, which is to tell it which player is asking it for valid moves. The function goes through the whole board instead of just a single position check, and does the same check as *Update()*, but instead of flipping the bit, it adds its current position into a vector of pairs of integers, which represents the row and column that is to be chosen. Once it checks all tiles it returns that vector to the caller.

We have implemented the Monte Carlo Tree Search (MCTS) heuristic as one of the AI that is playing the game. The AI picks its next move based on the results of the random playouts it gives at each valid move. The score that we implemented was to add the absolute value (MCTS score - Evaporation score). It is obvious that if this value is positive, MCTS had won the game and if the value is negative MCTS had lost the game. In our implementation we had MCTS play 500 random playouts and chose the move with the highest score.

For our second AI we chose the “Evaporation Strategy” (we named this AI Eva in short), the concept behind this strategy is that the easiest way to increase your relative mobility compared to your opponent is to have fewer pieces on the board. We had to test at which point in the game is it best to switch over from conquering the fewest number of pieces possible to the largest number of pieces possible. If there was a point where two or more moves results in the same number of pieces being conquered, we pick a random move.
In our implementation there is an if statement which evaluates (space_left() > n), if this is true we pick the minimum number of pieces conquered possible, if this is not true we pick the maximum.

We let out two AIs (Eva and MCTS) play against each other, MCTS always does 500 random play outs before deciding its next move, but for Eva we modified the N value to be 32, 16, 48 and 58 and we observed how varying N affected how well Eva plays against MCTS. Our results are shown below.

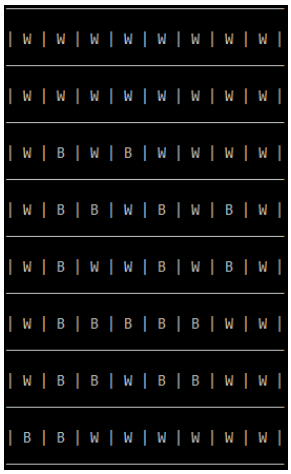
When running our code, it will output the board after each turn, and the position played. It will keep playing until the game is over and output the results.

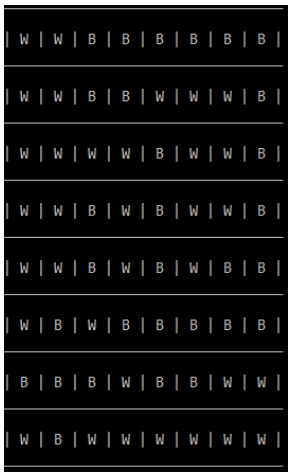
Value of N	Last Position Played	Results (Eva - MCTS)	Board
32	(8,1)	26 - **38**	
32	(1,2)	20 - **44**	
32	(1,8)	28 - **36**	

32	(8,1)	22 - **42**	
32	(2,7)	**41** - 23	

Value of N	Last Position Played	Results (Eva - MCTS)	Board
16	(1, 1)	32 - 32	
16	(8, 1)	**40** - 24	
16	(4,1)	**51** - 13	

16	(2, 7)	24 - **40**	
16	(1,1)	25 - **39**	

Value of N	Last Position Played	Results (Eva - MCTS)	Board
48	(1,3)	**44** - 20	
48	(4,1)	**64** - 0	
48	(1,3)	**34** - 30	

48	(8, 2)	**34** - 30	
48	(2, 1)	**38** - 26	

Value of N	Last Position Played	Results (Eva - MCTS)	Board
56	(7, 8)	**35** - 29	
56	(8,8)	**48** - 16	
56	(8,8)	**34** - 30	

56	(7, 8)	**52** - 12	
56	(2, 7)	**38** - 26	

Value of N	Final Standings (Eva win - MCTS win - draw)
32	1 - 4 - 0
16	2 - 2 - 1
48	5 - 0 - 0
58	5 - 0 - 0

Findings

From these records it is obvious that the most optimal *n* was 48 and 56. We noticed when we switched strategies too early (when N = 16 and 32) Eva had lost more often. New players often make this mistake when first playing this game, since the objective of the game is to capture as many of the opponent's pieces as you can. From our results we can see that capturing many of the opponent's pieces can backfire and lead to a loss. Eva performed much better when N = 48 and 58, which is when 75% and 90% (respectively) was occupied. This is surprising since, the only outside information we are giving to Eva is depending on how many pieces are placed on the board, Eva performs a certain strategy. With this information alone, it is enough to beat MCTS algorithm. In comparison, *Evaporation Strategy* won more total games when *n* = 48 than *Pure Monte Carlo Search*, so we conclude that Evaporation Strategy is the better strategy.