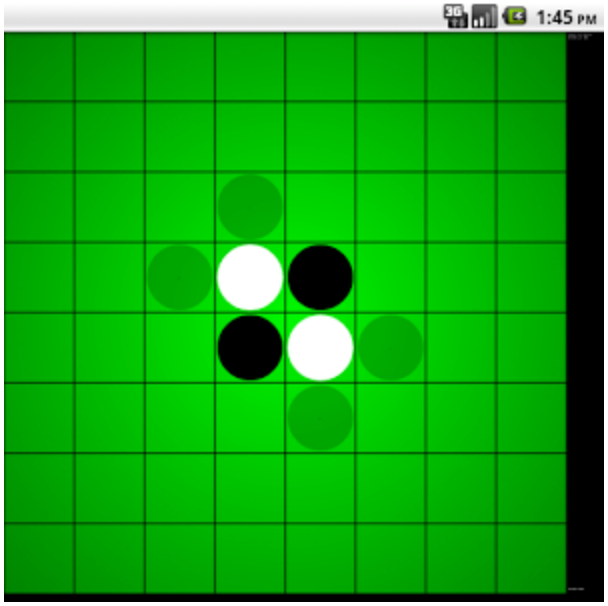
Kent Sommer

4511W Final Project
Reversi AI

Topic Paper

Reversi - Basics of the game:

Reversi is a strategy based board game for two players. There are 64 game pieces that are black on one side and white on the other, each player is assigned a colour (black or white). The players take turns placing a piece on the board (any piece that is in a straight line bounded by the piece just placed and another piece of the current player are flipped to the current player's colour). The line can be horizontal, vertical, or diagonal. It should be noted, however, that you must place a piece that creates a bound between the newly placed piece and one of that players already existing pieces (see picture)



Goal: To implement an AI to play Reversi at a respectable level
Domain:

1. **64 pieces**
2. **64 board spaces (8 x 8)**
3. **Placement requirements (see basics of the game description)**
4. **Not as simple as place to get greatest immediate gain (most number of pieces on the board).**

Questions and Research:

My question was "is minimax going to be the best algorithm to use for a Reversi AI?" As time has gone on I have modified the question somewhat to include "do some heuristics work better than others for Reversi?" The answer to both questions as you will see shortly is yes.

Let me first begin with the basic search algorithm. At the start of the semester when I was thinking of ideas for my final project, I landed on Reversi. I had previously written an AI to play 11 Men's Morris and found it interesting. Because of how much fun I had had working on a player agent for a board game, I chose Reversi. At the time, I was unsure of how to approach the problem, nothing we had learned yet seemed like it would really apply or work well for Reversi. Then one day, minimax was presented and I knew immediately this would be the algorithm I wanted to test. I used a code base to handle the player input and board drawing (this saved me a lot of time that I was able to use to research heuristics). Minimax was surprisingly harder for me to implement that I had thought it would be. I started with one function and ended up using two (one to make a call to the value function, and the recursive value function itself).

```
def miniMaxValue(board, maxply, best):
        #Implements MiniMax for Reversi AI
        possibleMoves = getValidMoves(board, computerTile)
        if maxply == 0 or isTerminal(board):
                score = getAltScoreOfBoard(board) #[computerTile]
                return score

        #try each move
        for x, y in possibleMoves:
                move = [x,y]
                #print(move)
                dupeBoard = getBoardCopy(board)
                makeMove(dupeBoard, computerTile, x, y)
                value = -1 * miniMaxValue(dupeBoard, maxply-1, best)
                #print(value)
                if best is 0 or value > best:
                        best = value
        return best
```

```
def miniMax(board, maxply):
        possibleMoves = getValidMoves(board, computerTile)
        best = None

        #try each move
        for x, y in possibleMoves:
                move = [x, y]
                dupeBoard = getBoardCopy(board)
                makeMove(dupeBoard, computerTile, x, y)
                value = -1 * miniMaxValue(dupeBoard, maxply, 0)
                #update best
                if best is None or value > best[0]:
                        best = (value, move)
        return best
```

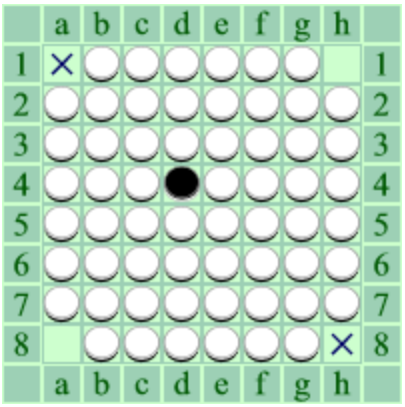
The line: "value = -1 * miniMaxValue(dupeBoard, maxply, 0)" is important because the value for the opponent (whoever is playing the AI) needs to get the value that is the opposite of whatever the board value is (this is computed recursively). Most of my initial frustration came from improper recursive calls which ended up giving completely wrong values for minimax and in turn led to abysmal play. I separated minimax into two functions to more easily wrap my head around the recursion concerning the depth limit. At its core, MiniMax will do a depth-first search of the whole game tree (if the depth limit is large enough). Heuristic values are assigned to leaf "nodes" in the game tree and are backed up to the root.

My next search algorithm was MiniMax with Alpha Beta, which, I am sad to report after a few hours of debugging I can't get to properly work. It will return valid moves, however, it does not play the same as basic MiniMax (it should). In fact, it plays much much worse (see code on next page).

```
def alphaBetaValue(board, maxply, alpha, beta):
        possibleMoves = getValidMoves(board, computerTile)

        if maxply == 0 or isTerminal(board):
                return getAltScoreOfBoard(board)[computerTile]

        #try each move
        for x, y in possibleMoves:
                move = [x, y]
                dupeBoard = getBoardCopy(board)
                makeMove(dupeBoard, computerTile, x, y)
                if beta is not None:
                        opp_alpha = -1 * beta
                else:
                        opp_alpha = None
                if alpha is not None:
                        opp_beta = -1 * alpha
                else:
                        opp_beta = None
                value = -1 * alphaBetaValue(dupeBoard, maxply-1, opp_alpha, opp_beta)
                #update alpha
                if alpha is -inf or value > alpha:
                        alpha = value
                #prune!! ;)
                if (alpha is not None) and (beta is not None) and alpha >= beta:
                        return beta
        return alpha


    def alphaBeta(board, maxply):
            possibleMoves = getValidMoves(board, computerTile)
            bestScore = 0
            bestMove = []

            #try each move
            for x, y in possibleMoves:
                    move = [x, y]
                    print("Current Move is: ", move)
                    dupeBoard = getBoardCopy(board)
                    makeMove(dupeBoard, computerTile, x, y)

                    if bestScore is not 0:
                            opp_beta = -1 * bestScore
                    else:
                            opp_beta = inf
                    value = -1 * alphaBetaValue(board, maxply, -inf, opp_beta)
                    print("Value: ", value)
                    #update best
                    if bestScore is 0 or value > bestScore:
                            if bestScore is 0:
                                    print("best is 0")
                            (bestScore, bestMove) = (value, move)
                            print("BestScore is:", bestScore)
            print(bestMove)
            print(bestScore)
            return (bestScore, bestMove)
```

Heuristics:

This was probably the most fun part for me to research and do work on. The first and most basic heuristic is to simply maximize the number of pieces on the board. In general this seems like it might actually be a good strategy. Everytime you play, you find the place that flips over the most of the opponents pieces so as to give you the most possible pieces of your colour on the board. This strategy becomes useless rather quickly. Consider the board below, it might look there is no way for white to lose this game, however, white has no valid moves left. Every turn from here on out white will have to pass and black will be able to play. Black ends up winning with a score of 40 to 24.



Once I researched more about Reversi gameplay and strategies, I discovered there are a few things that should be taken into account to provide a better heuristic.

The first heuristic is piece parity which I talked about above (maximizing pieces should be taken into account, it just cannot be fully relied upon).

The second heuristic is mobility. Mobility is a tactic to restrict the possible moves of the opponent while increasing your own. The number of moves the other player can do are decreased and this helps keep them from being able to gain control of the game. There are two different types of mobility that should be accounted for as well: actual and potential. Actual is the number of moves the other player has at that given state. Potential is how much potential moves that person might have over the next few turns (moves that aren't legal for the player now, may become legal later). Calculating potential mobility can be done by counting the number of empty spaces next to

at least one of the other players pieces. Although this is a fairly crude way to do it, it proved to be fairly effective for my needs.

The third heuristic is corners captured. This one is fairly self explanatory, but what it seeks to do is weight corners much heavier since once a piece has been placed on a corner, it cannot be flipped. Capturing a corner provides stability to that "region" of the game board which has a large effect on the outcome of the game. Weights are assigned to not only corners captured but also potential corners, and unlikely corners. A potential corner is one which could be "captured" in the next move, while an unlikely one is just the opposite.

The last heuristic (but not least) is stability. The measure of stability of a piece is simply a representation of how vulnerable it is to being flanked. Pieces are placed into three categories: stable, semi-stable, and unstable. Stable pieces are those that cannot be flanked at any point from a given start game state. Semi-stable simply means there is a low chance that they could be flanked in the future, but it is possible. Unstable pieces are those which can be flanked by the other player currently.

My goal for my heuristic function was to incorporate and weight each of these to produce a useful heuristic that performed better than the simple maximize pieces strategy. To do this I scaled each of the heuristic values from -100 to 100 and then weighted the overall result to produce a final board score for a given game state. (see code sample on next page)

Optional Heuristic:

A more simple heuristic would be using a static weights function over the whole game board. This implicitly captures the importance of each place on the board, and encourages play towards the corners.

4	-3	2	2	2	2	-3	4
-3	-4	-1	-1	-1	-1	-4	-3
2	-1	1	0	0	1	-1	2
2	-1	0	1	1	0	-1	2
2	-1	0	1	1	0	-1	2
2	-1	1	0	0	1	-1	2
-3	-4	-1	-1	-1	-1	-4	-3
4	-3	2	2	2	2	-3	4

```
    #Mobility
compT = compFT
playT = playFT
if compT > playT:
    m = (100 * compT)/(compT + playT)
elif compT < playT:
    m = -(100 * playT)/(compT + playT)
else:
    m = 0

    #calculate board score
score = (10 * p) + (801.724 * c) + (382.026 * l) + (78.922 * m) + (74.396 * f) + (10 * d)
return score
```

Testing and methodology:

Since I created an AI that worked to play against a human I ended up running two instances of the game to get the AI to play against itself with different heuristic functions. On one computer I had the computer go first and on another computer I had the human player (me) go first and I would enter the computer's move as my first move and continue this back and forth entering until the end of the game.

For the first set of tests I separated out heuristics from each other and ran them against other heuristics.

	Piece Parity	Corners	Mobility	Stability
Piece Parity	N/A	27-37 Corners win	14 - 50 Mobility win	26-38 Stability win
Corners	53-11 Corners win	N/A	39-25 Corners win	39-25 Corners win
Mobility	59-5 Mobility win	29-35 Corners win	N/A	0-42 Stability win
Stability	58-0 Stability win	13-51 Corners win	23-41 Mobility win	N/A

The corners heuristic if used by itself proves to be the most useful (this is why it is weighted heavier than the other heuristics in my code sample above). Overall, it Won 254 pieces and lost 130. The next best was mobility which won 204 pieces but lost 158. Stability came in third winning 199 pieces and losing 157. Piece parity came in a distant fourth winning a lowly 83 pieces and losing 295. The corner heuristic does best because it guides the game in a direction that increases the AI's chance of capturing corners. The more corners a player has the more control they have over the middle part of the board (includes flanking). Because of this, the corners heuristic can somewhat nullify the benefits provided by the other heuristics when playing against them.

Playing the combined heuristic against that of the static board heuristic was almost laughable. I did two runs of this test to make sure each had the opportunity to go first (this does effect the game by a wide margin). The game where the static heuristic went first resulted in a 26-38 win for the combined heuristic. The game where the combined heuristic went first resulted in a 60-4 win again for the combined heuristic. The reason that the static board heuristic does so much worse is that it isn't able to adjust for the current game state. Although it trends play towards corners, it doesn't take into account what is currently happening in the game and this puts it at a serious disadvantage.

Playing my combined heuristic against a human shows that although it plays very well compared to basic heuristics it still has a long way to go. Although I cannot beat it, I'm a novice at Reversi, and after asking a friend of mine who plays the game regularly to give it a shot she reported she was able to beat it by 15 points. So, did I fulfill my goal? Well, yes and no, it does play at a respectable level, and performs much better than random or even a heuristic that only accounts for one strategy. It does not, however, play very well against experienced players. I've had a wonderful time developing and researching this game (less fun debugging it…) and I hope you find this project as interesting as I have!