My implementation of Othello's AI agent builds on top of the code provided by the instructor. No changes or adjustments have been made to the othello and game files containing all associated classes. While most of my contributions are reflected in the agent file, as instructed, slight modifications were made inside main as well.
The HumanPlayer class was largely left untouched, with only a condition to return None if the number of moves available are 0 in order to avoid an error which was discovered during testing. The RandomAgent class is very similar to the HumanPlayer. The code checks for the length of the list of returned moves and returns the first element if it is equal to 1, None if empty, and generates a random integer index within the list length otherwise. The move of the random index is returned.
Moving on to the MinimaxAgent, an extra parameter was added into the class to keep track of the player turn: 1 for O, and 2 for X. The main file was in turn adapted to pass these values to the respective player class. The depth limit is initialized as an instance parameter as well. The choose_move method initializes the Minimax algorithm. The initial depth is 0, and the first set of possible moves are generated within it. For each one of the moves, the method checks if the player was the first to move or not, in which case it computes the minimum value of the state resulting by applying the move, and the maximum value if the player was the second to move. The min_value method takes a state and the current depth of the algorithm and is used recursively. If the state is game over or if the current depth is higher than the depth limit, the method returns the computed score. The initial v is equal to infinity. For each possible move, v is updated to be the minimum between its current value and the max_value of the resulting state. The max_value method is almost identical to min_value, but v is initialized to be negative infinity, and is updated as the maximum between its current value and the min_value of the resulting state. After the recursion ends, we are left with the expected values for each of the possible first moves. If the agent is the first to move, then we are trying to maximize the values, so we are returning the move with the highest value, and opposite if the agent moves second.
The implementation of the AlphaBeta agent is almost identical. This time around, we are also sending the alpha and beta parameters to the min and max_value recursive methods, which are again initialized with negative and positive infinity. After the recursive call inside the for loop iterating through each successor move, we also check if v is less or equal to alpha inside min_value, returning v if true, and then updating beta to be the minimum of its current value and v. For max_value, we check if v is greater or equal to beta and returning v if true, and updating alpha to be the maximum of its current value and v. All other steps are identical to the Minimax agent.
The first step of the testing procedure was creating a separate main file called repeat, which plays multiple games and records the winning percentage of the players, which was ran through a shell script named run2. The results are as follows:

minimax random 1:
    15/20

minimax random 2:
    19/20

minimax random 3:
6/10

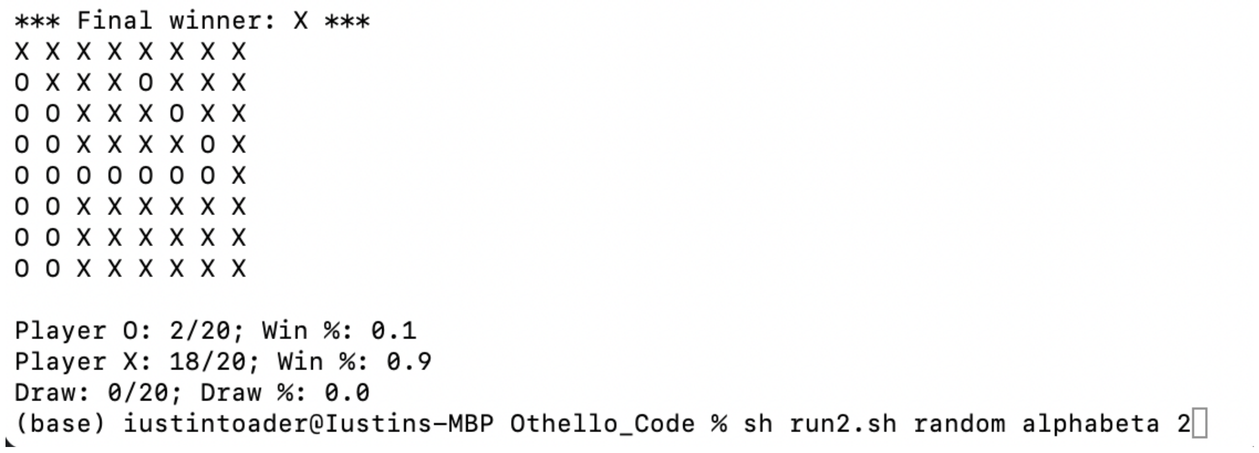
alphabeta random 1:
7/10

alphabeta random 2
9/10

alphabeta random 3:
9/10

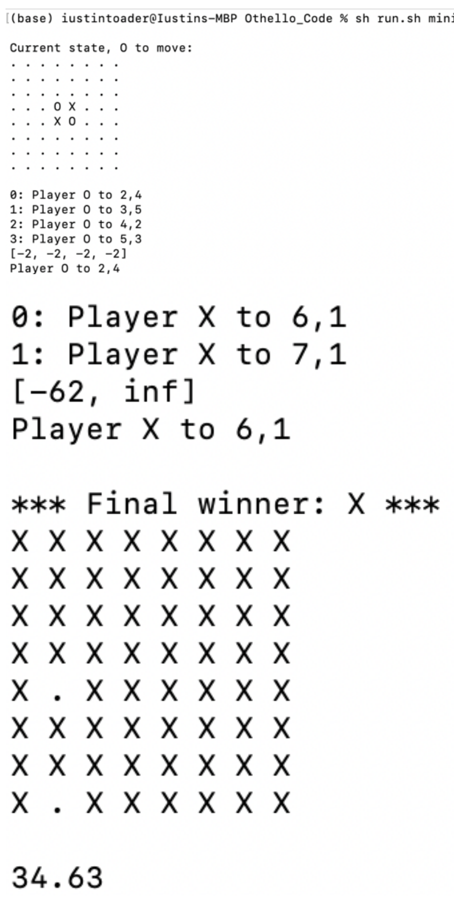
alphabeta random 4:
10/10

alphabeta random 5:
4/5

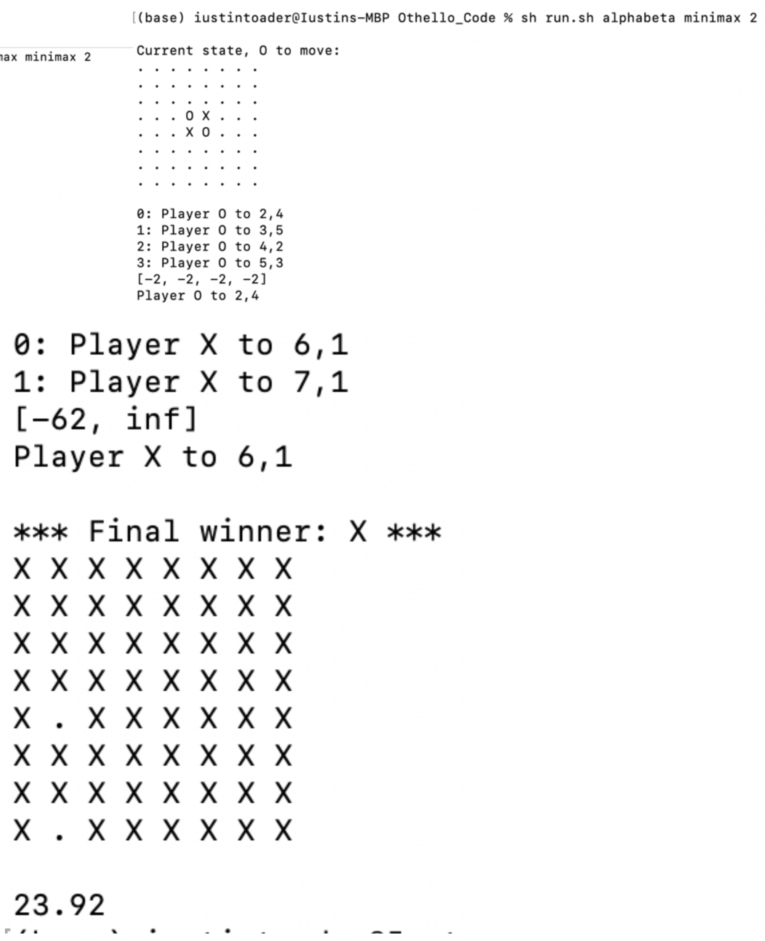

```
*** Final winner: X ***
X X X X X X X X
O X X X O X X X
O O X X X O X X
O O X X X X O X
O O O O O O O X
O O X X X X X X
O O X X X X X X
O O X X X X X X

Player O: 2/20; Win %: 0.1
Player X: 18/20; Win %: 0.9
Draw: 0/20; Draw %: 0.0
(base) iustintoader@Iustins-MBP Othello_Code % sh run2.sh random alphabeta 2
```

As we can see, our AI agents systematically outperform random agents and provide a high winning percentage. Although we expect the winning percentage to increase proportionally with the search depth, anything over 3 presents computational issues for the Minimax agent, and for the Alphabeta agent reaching a depth of 4. As such, the number of games played had to be decreased for higher depths. What's more, both algorithms are optimal against optimal opponents, but our opponent in this instance makes completely random moves, which makes it hard to evaluate performance and creates unexpected winning percentages for certain depths. Regardless, the results confirm the correctness of the implementations.
The same tests were also run with the alphabeta or minimax agent moving second and confirmed functionality. Only one such example is provided.

To also ensure that the Alphabeta implementation does indeed work as expected, a minimax agent was made to play against another minimax agent to ensure the same moves are made across calls, and then an alphabeta agent playing against a minimax agent. Since the second minimax agent makes the same moves against an optimal agent, and the Alphabeta method should work identically to the Minimax method, only faster, the outputs should be the same. This was confirmed, and the first and last lines of output for both calls are shown here:



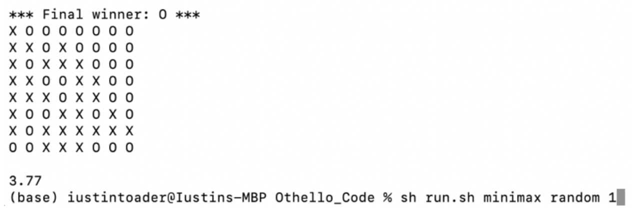
```
(base) iustintoader@Iustins-MBP Othello_Code % sh run.sh minimax minimax 2

Current state, O to move:
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . O X . . .
. . . X O . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .

0: Player O to 2,4
1: Player O to 3,5
2: Player O to 4,2
3: Player O to 5,3
[-2, -2, -2, -2]
Player O to 2,4

0: Player X to 6,1
1: Player X to 7,1
[-62, inf]
Player X to 6,1

*** Final winner: X ***
X X X X X X X X
X X X X X X X X
X X X X X X X X
X X X X X X X X
X . X X X X X X
X X X X X X X X
X X X X X X X X
X . X X X X X X

34.63
```



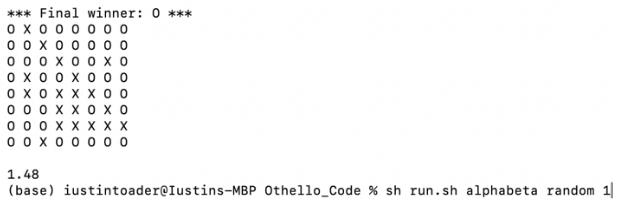
```
(base) iustintoader@Iustins-MBP Othello_Code % sh run.sh alphabeta minimax 2

Current state, O to move:
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . O X . . .
. . . X O . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .

0: Player O to 2,4
1: Player O to 3,5
2: Player O to 4,2
3: Player O to 5,3
[-2, -2, -2, -2]
Player O to 2,4

0: Player X to 6,1
1: Player X to 7,1
[-62, inf]
Player X to 6,1

*** Final winner: X ***
X X X X X X X X
X X X X X X X X
X X X X X X X X
X X X X X X X X
X . X X X X X X
X X X X X X X X
X X X X X X X X
X . X X X X X X

23.92
```

Another noteworthy discovery was the fact that, when an alphabeta or minimax agent played against another, the second player always won, up until depth 5. For minimax, the computation time took too long, but two alphabeta agents managed to finish the game in a reasonable time at depth 5. This appears to be slightly counterintuitive, as one would expect the first player that moves to have the upper hand in a situation where both players make optimal decisions. This result is most likely attributed to the fact that the agent can only check a few steps ahead, and not the entire game. As such, one would expect that, with high enough depths, the game would be skewed in the favor of the first player to move. Indeed, this is the case for a search depth of 5, but it is unclear if it persists for depths higher than that, since testing it would take too long.

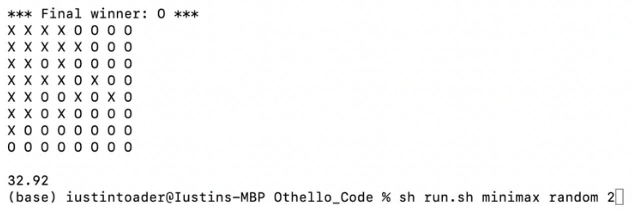
The next are results of both agents and their performance across different depths: (the output comes first and the call second)



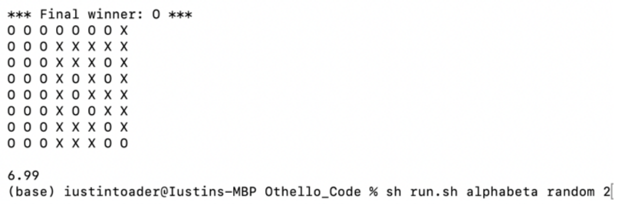
```
*** Final winner: O ***
X O O O O O O O
X X O X O O O O
X O X X X O O O
X X O O X X O O
X X X O X X O O
X O O X X O X O
X O X X X X X X
O O X X X O O O

3.77
(base) iustintoader@Iustins-MBP Othello_Code % sh run.sh minimax random 1
```



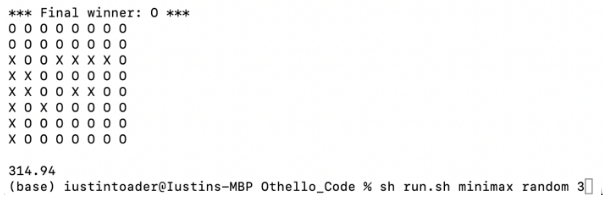
```
*** Final winner: O ***
O X O O O O O O
O O X O O O O O
O O O X O O X O
O X O O X O O O
O X O X X X O O
O O O X X O X O
O O O X X X X X
O O X O O O O O

1.48
(base) iustintoader@Iustins-MBP Othello_Code % sh run.sh alphabeta random 1
```



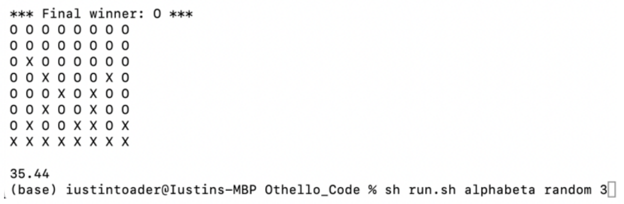
```
*** Final winner: O ***
X X X X O O O O
X X X X X O O O
X X O X O O O O
X X X X O X O O
X X O O X O X O
X X O X O O O O
X O O O O O O O
O O O O O O O O

32.92
(base) iustintoader@Iustins-MBP Othello_Code % sh run.sh minimax random 2
```



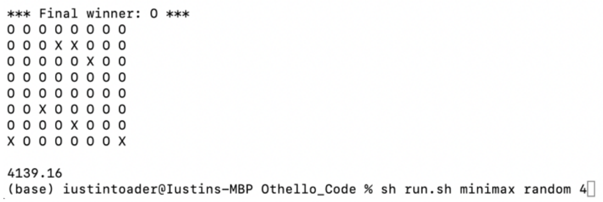
```
*** Final winner: O ***
O O O O O O O X
O O O X X X X X
O O O X X X O X
O O O X O X O X
O O O X O X X X
O O O X O O X X
O O O X X X O X
O O O X X X O O

6.99
(base) iustintoader@Iustins-MBP Othello_Code % sh run.sh alphabeta random 2
```



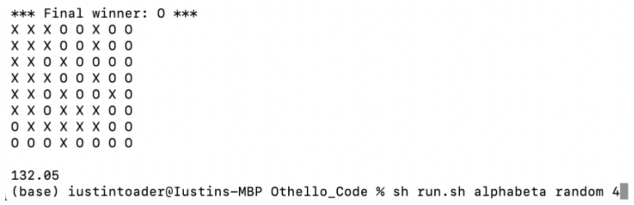
```
*** Final winner: O ***
O O O O O O O O
O O O O O O O O
X O O X X X X O
X X O O O O O O
X X O O X X O O
X O X O O O O O
X O O O O O O O
X O O O O O O O

314.94
(base) iustintoader@Iustins-MBP Othello_Code % sh run.sh minimax random 3
```



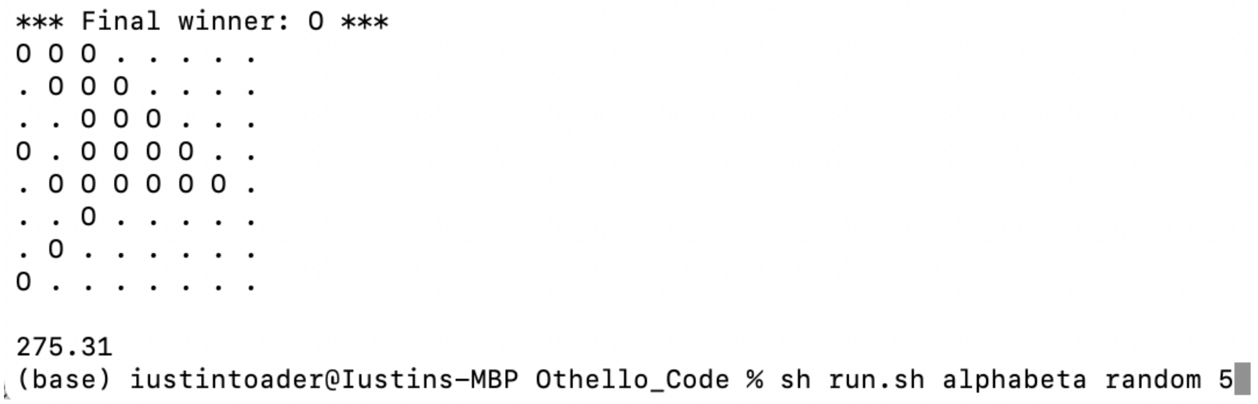
```
*** Final winner: O ***
O O O O O O O O
O O O O O O O O
O X O O O O O O
O O X O O O X O
O O O X O X O O
O O X O O X O O
O X O O X X O X
X X X X X X X X

35.44
(base) iustintoader@Iustins-MBP Othello_Code % sh run.sh alphabeta random 3
```



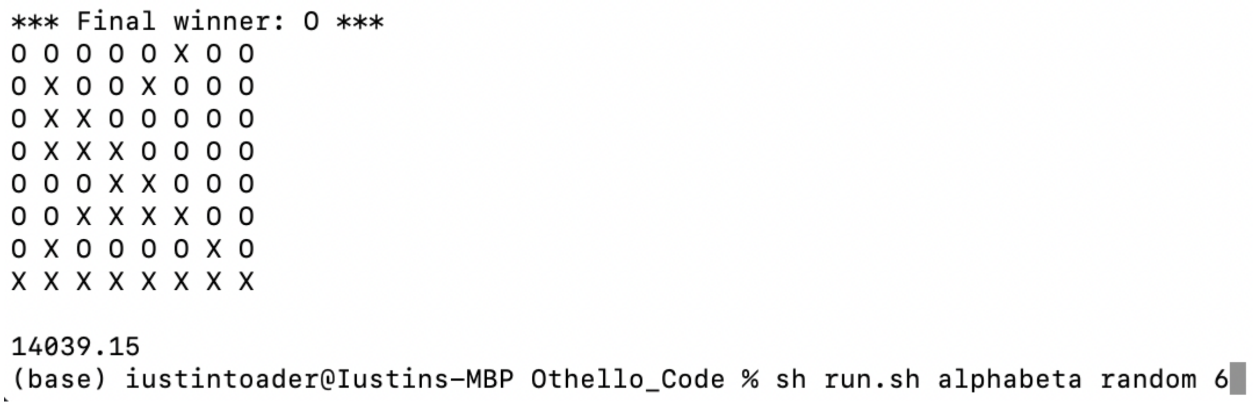
```
*** Final winner: O ***
O O O O O O O O
O O O X X O O O
O O O O O X O O
O O O O O O O O
O O O O O O O O
O O X O O O O O
O O O O X O O O
X O O O O O O X

4139.16
(base) iustintoader@Iustins-MBP Othello_Code % sh run.sh minimax random 4
```



```
*** Final winner: O ***
X X X O O X O O
X X X O O X O O
X X O X O O O O
X X X O O X O O
X X O X O O X O
X X O X X X O O
O X X X X X O O
O O O X O O O O

132.05
(base) iustintoader@Iustins-MBP Othello_Code % sh run.sh alphabeta random 4
```



```
*** Final winner: O ***
O O O . . . . .
. O O O . . . .
. . O O O . . .
O . O O O O . .
. O O O O O O .
. . O . . . . .
. O . . . . . .
O . . . . . . .

275.31
(base) iustintoader@Iustins-MBP Othello_Code % sh run.sh alphabeta random 5
```



```
*** Final winner: O ***
O O O O O X O O
O X O O X O O O
O X X O O O O O
O X X X O O O O
O O O X X O O O
O O X X X X O O
O X O O O O X O
X X X X X X X X

14039.15
(base) iustintoader@Iustins-MBP Othello_Code % sh run.sh alphabeta random 6
```

From the provided results, the benefit of alpha-beta pruning becomes apparent. Up until depth 3, the time complexity of alpha-beta pruning is closer to that of the minimax algorithm at the lower depth than that of minimax at the same depth. At depth 4 and up, alpha-beta pruning significantly outperforms minimax at even its lower depth. For example, alphabeta random 4 only took 132.05 seconds to terminate and alphabeta random 5 took 275.31, while minimax random 3 took 314.94 seconds. On the other hand, minimax random 4 took 4139.16 seconds to finish executing. Any depth larger than 4 results in a time complexity too high to test. Alphabeta was also run at depth 6, where it took 14039.5 seconds to finish executing.
Just as expected, our testing results confirm the exponential behavior expected, as minimax has a time complexity of O(b^m). An exact time complexity is harder to compute for alpha-beta pruning. With perfect ordering, the time complexity drops to O(b^*m/2)). However, my implementation does not impose this optimal ordering, so the results are harder to quantify. In any case, the alpha-beta pruning algorithm provides a dramatic improvement in time complexity, and is clearly the preferred option for an intelligent agent.