Jafet Calderon Concepcion
CSCI-4740
Dr. Lietchen
Othello Project

Version 1:

Code:

```
import random
from Othello import *

class Player(BasePlayer):
    def __init__(self, timelimit):
        BasePlayer.__init__(self, timelimit)

    def findMove(self, state):
        actions = state.actions()

        if state._turn % 2 == 0:
            best = -10000
            for a in actions:
                result = state.result(a)
                v = self.heuristic(result)
                if v > best:
                    best = v
                    bestMove = a
        else:
            best = 10000
            for a in actions:
                result = state.result(a)
                v = self.heuristic(result)
                if v < best:
                    best = v
                    bestMove = a
        self.setMove(bestMove)
        print('\tBest value', best, state.moveToStr(bestMove))


    def heuristic(self, state):
        return state.score()
```

Description:

For my first version I decided to copy the Greedy.py file to see how well it would hold up against its own code. With that being said, I had to change up some of the code in order for the compiler to run without giving an error. At first, without changing any of the code, it gave me the error: player1 = playerModule.Player(timeLimit) module"MyAgent" has no attribute 'Player'. It gave me this error because in the code it had no line where the Othello code could read a player attribute as it didn't have a defined class and I was able to fix this by adding the following lines:

```
def __init__(self, timelimit):
        BasePlayer.__init__(self, timelimit)
```

After adding these lines I put it up against the following agents: Greedy, MinMax, Random and ran it 10 times with each one with 5 runs with my agent being black and 5 runs with my agent being white. Me results are as follows:

	Greedy	MinMax	Random
MyAgent	Black: 3 - 2 White: 3 - 2	Black: 2-3 White: 1-4	Black: 2-3 White: 3 - 2

Findings:

For the most part I did win a couple games but overall it ended with a draw. Since the code only looks for a move based on position it did not give a very good winning result. Some improvements that could be made is the fact that I can add a more heuristic approach by adding code that looks ahead in order to plot the best move

Version 2:

Code:

```
from Othello import *

class Player(BasePlayer):
    def __init__(self, timeLimit):
        BasePlayer().__init__(self, timeLimit)
        self.maxDepth = 4

    def findMove(self, state):
        bestMove = None
        bestValue = float('-inf')

        for move in state.actions():
            nextState = state.result(move)
            value = self.alphabeta(nextState, self.maxDepth - 1,
                                   float('-inf'), float('inf'), False)
```

```
            if value > bestValue:
                bestValue = value
                bestMove = move

        print(f"AI chooses move: {state.moveToStr(bestMove)} (value:
{bestValue})")
        self.setMove(bestMove)

    def alphabeta(self, state, depth, alpha, beta, maximizingPlayer):
        if depth == 0 or state.isTerminal():
            return self.evaluate(state)

        if maximizingPlayer:
            value = float('-inf')
            for move in state.actions():
                nextState = state.result(move)
                value = max(value, self.alphabeta(nextState, depth-1,
alpha, beta, False))
                alpha = max(alpha, value)
                if alpha >= beta:
                    break
            return value
        else:
            value = float('inf')
            for move in state.actions():
                nextState = state.result(move)
                value = min(value, self.alphabeta(nextState, depth-1,
alpha, beta, True))
                beta = min(beta, value)
                if alpha >= beta:
                    break
            return value

    def evaluate(self, state):
        black, white = state.countPieces()
        if self.color == 'black':
            return black - white
        else:
            return white - black
```

Description:

For this code I followed an alpha-beta pruning method where it would essentially look forward in order to move more strategically rather than greedy like the previous code. The 'Player' class inherits from 'BasePlayer' and sets a maximum search depth of 4 moves ahead. In 'findMove', the agent examines all legal moves and evaluates them by recursively calling the 'alphabeta' function, which explores possible future states while pruning branches that cannot improve the outcome. The 'alphabeta' method chooses between maximizing and minimizing players, updating 'alpha' and 'beta' values to cut off unnecessary searches and increase its efficiency. The 'evaluate' function computes a simple heuristic, giving the difference in piece counts depending on the AI's color. Unlike the first greedy code, this code considers multiple moves into the future and anticipates the opponent's responses. Overall, this makes the alphaBeta much more strategic, as it balances immediate gains with potential future consequences rather than acting purely on short-term heuristics. My code versus greedy, minmax, and random.

	Greedy	MinMax	Random
MyAgent	Black: 4-1 White: 3 - 2	Black: 3-2 White: 2-3	Black: 4-1 White: 2-3

Findings:

As stated previously this code takes a more strategic course as it looks ahead to ensure that the next move it takes is the best decision. I won by a larger margin but I still lost most games against the MinMax agent. Some improvements I could add code snippets to improve its time efficiency, prioritizing corners and having more mobility in its moves.

Version 3:

Code:

```
import random
from Othello import *

class Player(BasePlayer):
    def __init__(self, timeLimit):
        BasePlayer.__init__(self, timeLimit)
        self.maxDepth = 4

    def findMove(self, state):
        bestMove = None
        bestValue = float('-inf')

        # Loop through all possible moves
```

```
        for move in state.actions():
            nextState = state.result(move)
            value = self.alphabeta(nextState, self.maxDepth - 1,
                                   float('-inf'), float('inf'), False)
            if value > bestValue:
                bestValue = value
                bestMove = move

        print(f"AI chooses move: {state.moveToStr(bestMove)} (value:
{bestValue})")
        self.setMove(bestMove)

    def alphabeta(self, state, depth, alpha, beta, maximizingPlayer):

        # Stop if depth limit reached or game is over
        if depth == 0 or state.gameOver():
            return self.heuristic(state)

        if maximizingPlayer:
            value = float('-inf')
            for move in state.actions():
                nextState = state.result(move)
                value = max(value, self.alphabeta(nextState, depth-1,
alpha, beta, False))
                alpha = max(alpha, value)
                if alpha >= beta:
                    break  # Beta cut-off
            return value
        else:
            value = float('inf')
            for move in state.actions():
                nextState = state.result(move)
                value = min(value, self.alphabeta(nextState, depth-1,
alpha, beta, True))
                beta = min(beta, value)
                if alpha >= beta:
                    break  # Alpha cut-off
            return value

    def heuristic(self, state):
```

```
        return state.score()
```

Description:

This code is a merging between the greedy code and an alpha-beta pruning heuristic to evaluate potential moves more strategically than a purely greedy approach. The ‘Player’ class inherits from ‘BasePlayer’ and sets a maximum search depth of 4 moves ahead. In 'findMove', the agent loops through all legal actions, simulates the resulting state for each move, and evaluates it using the ‘alphabeta’ method. The ‘alphabeta’ function recursively explores future states, alternating between maximizing and minimizing players, while using ‘alpha’ and ‘beta’ to prune branches that cannot improve the outcome. The ‘heuristic’ function simply returns the board score, providing a quick evaluation of each state. Unlike the original greedy code, which only looks one move ahead and picks the immediate best score, this implementation considers multiple future moves and anticipates opponent responses, making it more strategic. Compared to previous alpha-beta implementations, this merged code combines the simplicity and direct evaluation of a greedy approach with depth-limited pruning, resulting in a faster yet still forward-looking decision-making process.

	Greedy	MinMax	Random
MyAgent	Black: 4-1 White: 5-0	Black: 4-1 White: 3-2	Black: 4-1 White: 3-2

Findings:

Out of all the versions this is by far the better working one as it provides a clear and more strategic approach to the game. As the games were run, I found more success and a greater margin of winning.