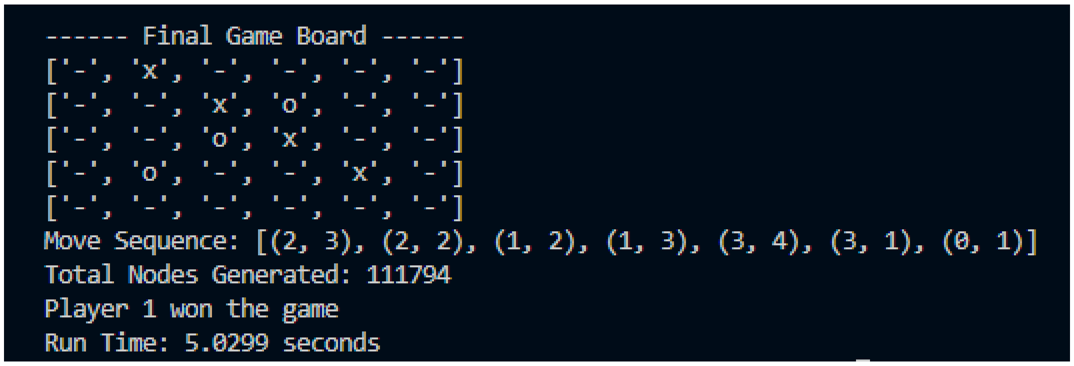
Artificial Intelligence I - Homework 4

Participants:
Olivia Shouse
Ryan Christopher
Luke Schaefer

Language: Python

- Output
    - These values are 0 indexed. (0,0) to (4,5)
    - Every move made during game between player 1 and 2
        - Move 1: Player 1 placed “x” at (2, 3)
        - Move 2: Player 2 placed “o” at (2, 2)
        - Move 3: Player 1 placed “x” at (1, 2)
        - Move 4: Player 2 placed “o” at (1, 3)
        - Move 5: Player 1 placed “x” at (3, 4)
        - Move 6: Player 2 placed “o” at (3, 1)
        - Move 7: Player 1 placed “x” at (0, 1)
    - For each move
        - number of nodes generated by minimax
            - Total Nodes Generated: 111,794 Nodes
        - CPU Execution time
            - 5.0299 Seconds



```
------ Final Game Board ------
['-', 'x', '-', '-', '-', '-']
['-', '-', 'x', 'o', '-', '-']
['-', '-', 'o', 'x', '-', '-']
['-', 'o', '-', '-', 'x', '-']
['-', '-', '-', '-', '-', '-']
Move Sequence: [(2, 3), (2, 2), (1, 2), (1, 3), (3, 4), (3, 1), (0, 1)]
Total Nodes Generated: 111794
Player 1 won the game
Run Time: 5.0299 seconds
```

Classes

- Game Class:
    - Attributes
        - Gameboard: Current Gameboard
        - Player1: First player in the game
        - Player2: Second player in the game
        - MoveSequence: Sequence of moves that can be played in the game
    - Functions
        - Getters and Setters
        - addMoveSequence()

        - playGame()
- Board Class:
    - Attributes
        - Winner: Contains the winner if there is one or 0 if there is no winner yet
        - Board: 2d array of the gameboard
        - Terminal: If there are moves left in the gameboard then it is not terminal
        - OpenMoves: Number of moves available in the gameboard
        - RowCount: Number of rows in the gameboard
        - ColCount: Number of columns in the gameboard
        - TwoSideOpen3ForX: Number of 3 in a row x's with both sides open
    - Functions
        - setNewBoard()
        - printBoard()
        - clearSideOpens()
            - Clears all the calculated values for the number of 'x's and 'o's in a row
        - resetMove()
        - placeMoves()
            - Places the specified symbol at the specified location, updates the side opens values and checks if there is a winner
        - determineMove()
            - Checks rows, columns, and diagonals for if there are multiple 'x's or 'o's in a row
        - checkbounds()
            - Checks the bounds of the given row and column
        - addSideOpen()
            - Updates the side open values based on the given parameters
        - checkRow()
            - Checks all the rows
        - checkCol()
            - Checks all the columns
        - newDiag1Check()
            - Checks the top left half of the board from bottom left to top right for diagonals
        - newDiag2Check()
            - Checks the bottom right half of the board from bottom left to top right for diagonals
        - newDiag3Check()
            - Checks the bottom left half of the board from top left to bottom right for diagonals
        - newDiag4Check()
            - Checks the top right half of the board from top left to bottom right for diagonals
- Node Class:

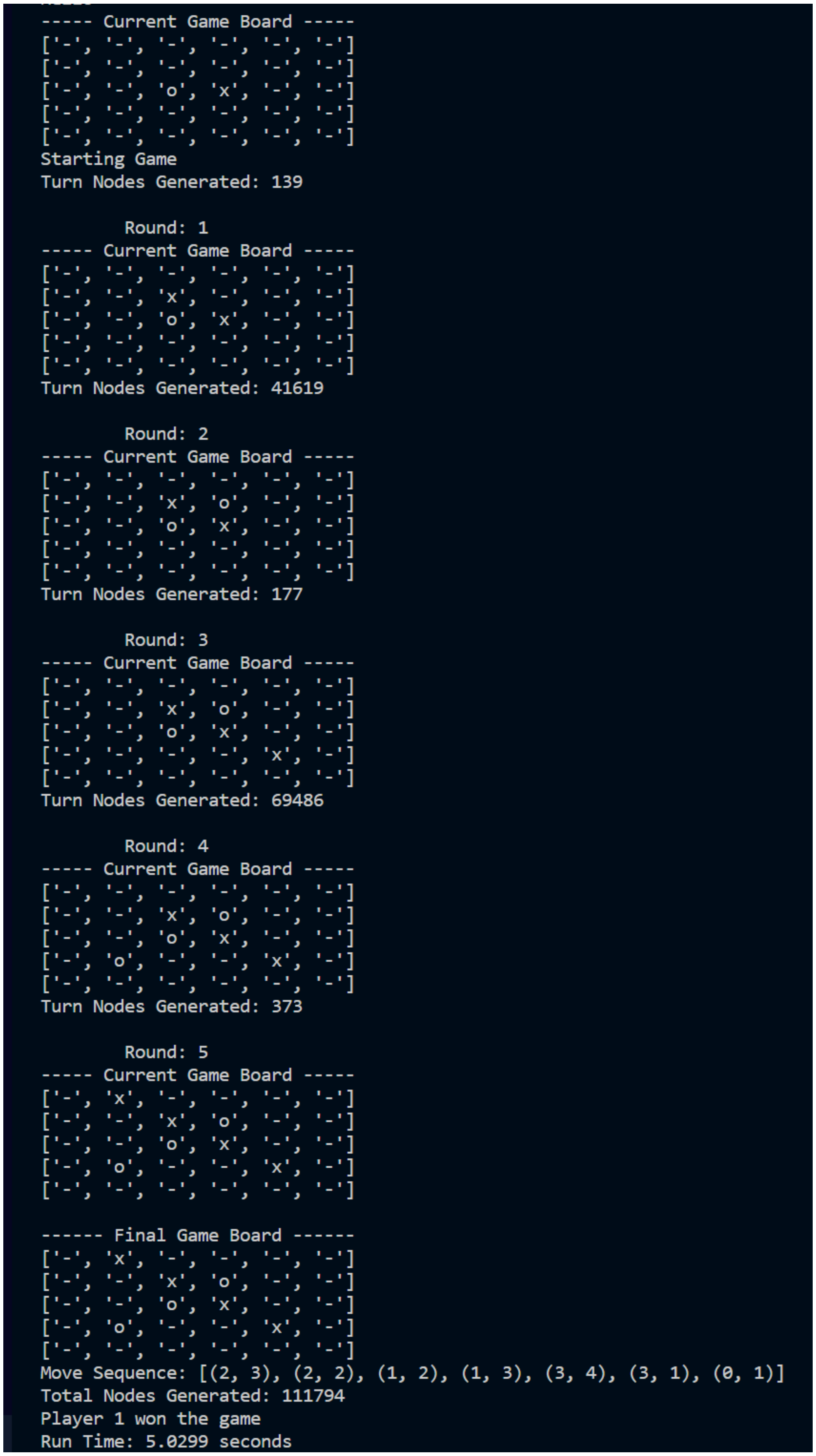
    - Attributes
        - State: Current Gameboard
        - Move: Move taken to get to current gameboard
        - CurrentPlayer: Player whos turn it is
        - MovesLeft: How many moves are left in look ahead
        - Children: Children nodes of current node
    - Functions
        - checkPieceNextToColCheck()
            - Used to check if there is a piece placed next to current location
        - checkPieceNextToRowCheck()
            - Used to check if there is a piece placed next to current location
        - hasPieceNextTo()
            - Checks if there is a piece next to the current location
        - expand()
            - Used to find all the possible children of the given node and append them to the children array
        -
- Player Class:
    - Attributes
        - MovesAhead: Number of moves ahead that the player can look
        - PlayerName: Name of player
        - PlayerSymbol: Symbol of player
        - firstMove: First move that the player makes in the game

Tasks

1. Minimax Algorithm
    a. We implemented the minimax algorithm inside the minimaxSearch function. In this function, we created a root node that contains the number of moves ahead that it is supposed to look, which player is making the current move, and the current board. For abstraction, we created helper functions which include findBestMove() and minimax().
    b. findBestMove() is used to begin the minimax process from the root node. It expands the rootnode and then calls the minimax function on all of its children. After minimax finishes exploring all of its children, it checks for the best value and returns the correlated move and number of nodes generated.
    c. minimax() uses recursion to look ahead the specified number of moves. By doing this, it allows each child to be searched in a tree like fashion until the desired level is reached and the best move is found. The minimax() function first checks if there is a winner and returns the correlated values. Next, it checks if the desired level in the search tree has been reached and returns the utility value of each child if it has. Lastly, it checks if it is on a minimum or maximum level, expands the current node, and runs minimax() recursively on each of the current nodes children.

   d. The output of our minimax algorithm is the best move that can be made given the specified heuristics and the number of nodes generated to obtain that move.
2. Implement Player 1
   a. To implement player 1, we created a Player class which contains the number of moves ahead, the players name, the players symbol, and the first move that the player makes in the game. Using this information, we are able to call minimax with the specified number of moves to look ahead to always find it's next best move. In this situation, the number of moves to look ahead for player 1 was 2 and the first move that player 1 made was [2,3] since we used zero based indexing.
3. Implement Player 2
   a. To implement player 2, we used the same Player class that we used to implement player 1. However, this time, we changed the number of moves to lookahead to 4 and the first move that player makes to [2,2] since we had zero based indexing.
4. Play a game between Player 1 and Player 2
   a. To complete this, we created a Game() class which contained player 1, player 2, the gameboard, and the sequence of moves that were made over the course of the game. We then created a function to play the game which initialized the players with the correct data and the looped between the two until the gameboard reached a terminal state. The possible terminal states are player 1 wins, player 2 wins, or the gameboard fills up and both players tie.
   b. We also included helper classes and functions to aid with the way we kept track of and output our results.

c.



```
----- Current Game Board -----
['-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-']
['-', '-', 'o', 'x', '-', '-']
['-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-']
Starting Game
Turn Nodes Generated: 139

        Round: 1
----- Current Game Board -----
['-', '-', '-', '-', '-', '-']
['-', '-', 'x', '-', '-', '-']
['-', '-', 'o', 'x', '-', '-']
['-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-']
Turn Nodes Generated: 41619

        Round: 2
----- Current Game Board -----
['-', '-', '-', '-', '-', '-']
['-', '-', 'x', 'o', '-', '-']
['-', '-', 'o', 'x', '-', '-']
['-', '-', '-', '-', '-', '-']
['-', '-', '-', '-', '-', '-']
Turn Nodes Generated: 177

        Round: 3
----- Current Game Board -----
['-', '-', '-', '-', '-', '-']
['-', '-', 'x', 'o', '-', '-']
['-', '-', 'o', 'x', '-', '-']
['-', '-', '-', '-', 'x', '-']
['-', '-', '-', '-', '-', '-']
Turn Nodes Generated: 69486

        Round: 4
----- Current Game Board -----
['-', '-', '-', '-', '-', '-']
['-', '-', 'x', 'o', '-', '-']
['-', '-', 'o', 'x', '-', '-']
['-', 'o', '-', '-', 'x', '-']
['-', '-', '-', '-', '-', '-']
Turn Nodes Generated: 373

        Round: 5
----- Current Game Board -----
['-', 'x', '-', '-', '-', '-']
['-', '-', 'x', 'o', '-', '-']
['-', '-', 'o', 'x', '-', '-']
['-', 'o', '-', '-', 'x', '-']
['-', '-', '-', '-', '-', '-']

------ Final Game Board ------
['-', 'x', '-', '-', '-', '-']
['-', '-', 'x', 'o', '-', '-']
['-', '-', 'o', 'x', '-', '-']
['-', 'o', '-', '-', 'x', '-']
['-', '-', '-', '-', '-', '-']
Move Sequence: [(2, 3), (2, 2), (1, 2), (1, 3), (3, 4), (3, 1), (0, 1)]
Total Nodes Generated: 111794
Player 1 won the game
Run Time: 5.0299 seconds
```

d. This output continas the board after each round, the number of nodes generated each round, the move sequence of the game, the total number of nodes generated, the player who won the game, and the total CPU execution time.

Testing

To aid with our development process, we created tests for a handful of the large-workload functions. These functions were essential to the success and accuracy of the program so we ensured to test them separately first before using them within the overall program.

Conclusion

We came to the conclusion that this heuristic was designed to prioritize each players own points over anything else. Therefore, it would always choose to maximize its own points instead of blocking its opponent from winning. An example of this occurred during round 2. Instead of blocking player 1 from getting 3 in a row, player 2 opted to get 2 in a row. Therefore, by the time round 4 came around, player 2 had no way of stopping player 1 from winning. This ultimately made it so that the player who went first was the one to win.