1.
    a. Please see attached Python file
    b. Please see attached text files. The computer as ‘X’ is displayed in output1.txt, and the computer as ‘O’ is displayed in output2.txt.
    c. Josiah created the initial logic code, including the implementation of the minimax algorithm, alpha-beta pruning, and score evaluation. In addition, Josiah rewrote much of the starter code in an attempt to improve efficiency and understandability. Jason handled debugging of the algorithm, and getting the computer to use the algorithms to make logical moves. Jason also handled some of the basic game setup, including some IO and initialization functions, as well as the main game loop and main function.
2.
    a. The next ply is generated by the move function. The computer calls move, which initially calls the maximize function with the computer's marker. This function, in conjunction with its minimize twin, evaluates all possible board positions and returns the first move in the optimal move sequence.
    b. The evaluation functions are maximize and minimize, which call each other recursively. Maximize fills in the computer's marker for each open square (minimize does the same thing for a simulated human marker) until an endgame state is reached, at which point the endgame value is evaluated. A scoring function evaluates the boards; if the state is a "win" for the computer, 1 is returned from the scoring function, whereas -1 is returned for a "loss", and 0 is returned for a "cat game". Because maximize and minimize are based on the minimax algorithm, with alpha-beta pruning, nodes that evaluate to a loss are discarded early, and nodes that evaluate to a win are retained.
3. This code generates children specific to Tic-Tac-Toe boards - there are nine squares, and each move fills in an empty square with the maximizing or minimizing marker. For Othello, there are 64 possible squares, but four are initially filled. Valid moves must be empty squares that also capture at least one piece, so strictly evaluating all empty squares is not an option. Instead, each empty square will have to be evaluated for validity, and likely put into a list of valid squares, before an iterator attempts all these positions. This would likely be done in a helper function called from maximize() and minimize(), since nothing in Tic-Tac-Toe currently accomplishes this. Evaluation will also be different. With Tic-Tac-Toe, there are a very small number of possible positions. The simple heuristic of win, loss, or tie works well in this situation. However, because Othello has many more possible win combinations, a more efficient heuristic must be devised. One possible way of doing this would be to evaluate intermediate positions, and assign them high values when the computer has both a large number of flipped tiles and few moves that flip them to the human player. This would probably be accomplished via a helper function that evaluated positions on the fly, or by integrating checks into the current check_win() function. Because the minimax algorithm and alpha-beta pruning are scalable, the algorithmic side of the code would require little to no alteration. Other changes would have to be made in the main(), print() and other setup functions, but these would be either supplied by the starter code, or easy to implement based on the ideas nurtured in Tic-Tac-Toe.