Goal: Our code for this assignment will be able to play 4 by 4 by 4 tic-tac-toe using AI techniques, specifically alpha-beta pruning, minimax, and heuristic evaluation.

Method and implementation:

Pseudocode:

Main algorithm:

1. Initialize current score and an ideal score to store highest score during the following loop
2. Find all open position on board
3. Iterate through open positions
4. Copy the current board and play an open move on the copied board
5. Assess the score of the position using minimax
6. Store the highest score and return the move associated with that score

Minimax:

1. Determine if the game is over. This is a terminal node if there is a winner or a tie.
2. Return the heuristic of the board if at depth 0.
3. Initialize value variable
4. If maximizing agent
    a. Set value to minimum integer
    b. Iterate through all current open positions
    c. Play the given move
    d. Call minimax of new board with depth - 1. Also set the player to the minimizing agent and set value equal to the maximum between current value and the value returned from minimax.
    e. Update alpha by finding the max between current alpha and value
    f. If alpha if greater than or equal to beta, break to prune the tree.
5. If minimizing agent
    a. Set value to maximum integer
    b. Iterate through all current open positions
    c. Play the given move
    d. Call minimax of new board with depth - 1. Also set the player to the maximizing agent and set value equal to the minimum between current value and the value returned from minimax.
    e. Update alpha by finding the minimum between current alpha and value
    f. If alpha if greater than or equal to beta, break to prune the tree.
6. Return value

Heuristic:

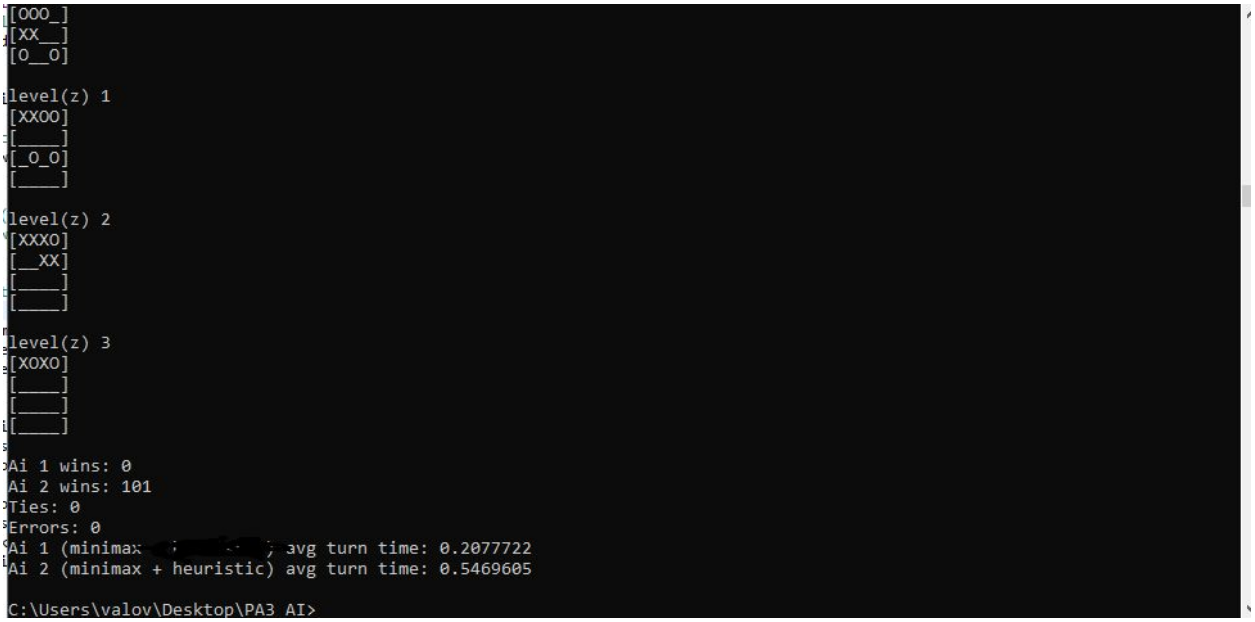
1. Initialize total score variable
2. Iterate through winning lines
    a. Check positions are filled by maximizing player.
        i. If no positions in the line are filled, break.
        ii. If 1 position if filled, increment total score by 1.
        iii. If 2 positions are filled, increment total score by 10.
        iv. If 3 positions are filled, increment total score by 100.
    b. Check positions are filled by minimizing player.
        i. If no positions in the line are filled, break.
        ii. If 1 position if filled, decrease total score by 1.
        iii. If 2 positions are filled, decrease total score by 10.
        iv. If 3 positions are filled, decrease total score by 100.
3. Return total score

Methods implemented:

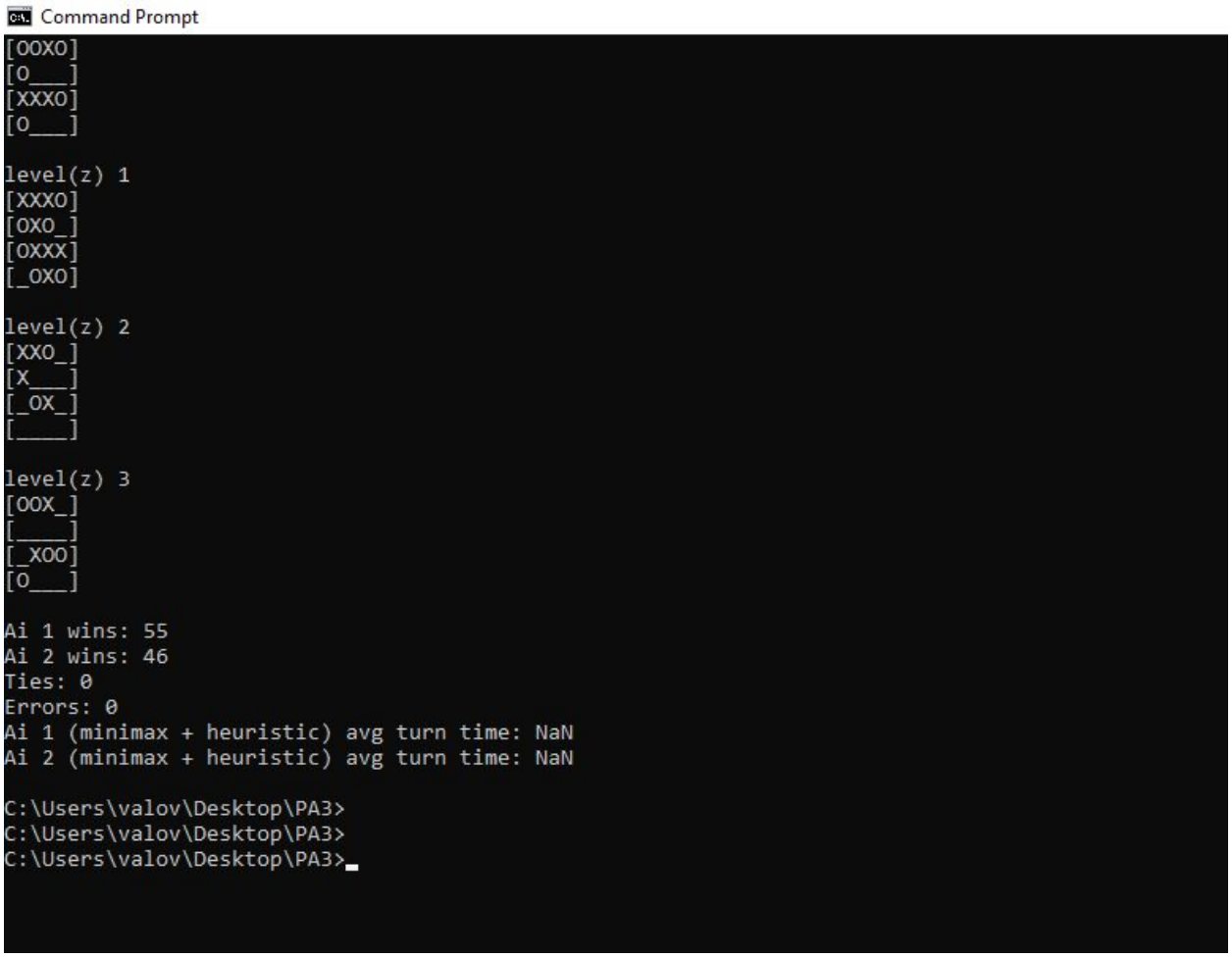
- getOpenPositions(board object): We created this method to return all current open positions on the board in a list.
- minimax(board object, int depth, int alpha, int beta, boolean maximizing): See above for pseudocode.
- Heuristic(board object): See above for pseudocode. Goal was to get the score of the current board position.
- deepCopyATicTacToeBoard(board object): Deep copied the board. The code was identical to the code given in the runTicTacToe class.
- isEnded(board object): Checked if the game ended and returned who the winner was or -1 if it was a tie. Code was identical to the code given in the runTicTacToe class.

Experiments and Result:

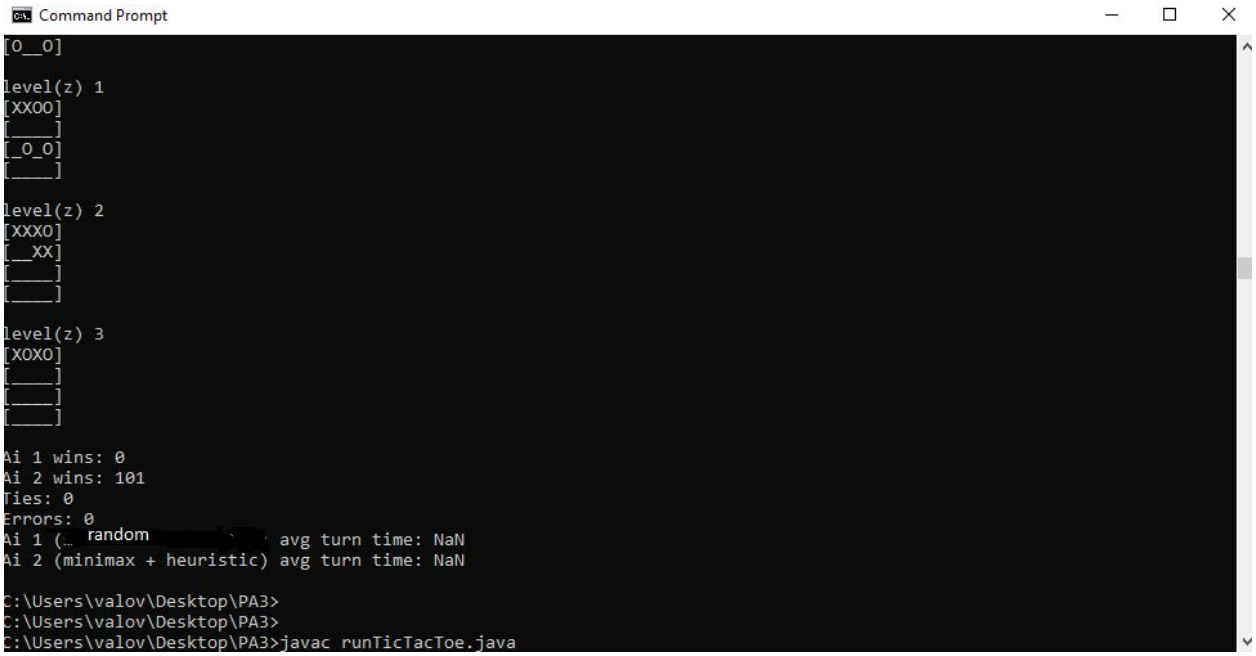
Minimax + Heuristic vs Random: 100% win rate
Minimax + Heuristic vs Minimax 100% win rate
Minimax + Heuristic vs Minimax + Heuristic 55% win rate



Minimax AI wins	0
Minimax and Heuristic AI wins	101



Minimax and Heuristic AI 1 wins	55
Minimax and Heuristic AI 2 wins	46



Random AI wins	0
Minimax and Heuristic AI wins	101

Discussions:

As seen in the results, our AI performed better when using both minimax (with alpha beta pruning) and assigning heuristic compared to a random move and an AI without the heuristic implementation. This showed that our AI was making moves better than a random move generator and that heuristic evaluation of a position was an essential step in the minimax procedure. Interestingly, when testing our AI against itself, the game split was not evenly split 50/50. We believe this is due to the fact that the initial move is randomly give to either AI 1 or AI 2. If we increased the number of tests, the game split would have trended closer to 50/50.

Moves were made by iterating through a list of all open positions and playing that position. After playing a position the board would be evaluated using minimax and heuristics. Child nodes using during the minimax procedure were all the remaining open positions after playing the initial position. Possible moves were pruned out using alpha beta pruning in order to decrease evaluation time and increase efficiency. For the heuristic evaluation, we first iterated through all winning lines and analyzed the state of the position in each winning line. If a winning line had 1, 2, or 3 positions marked by the maximizing player after playing the move, then the value of that position would increase by 1, 10, or 100 points respectively. On the other hand, if a winning line had 1, 2, or 3 positions marked by the minimizing player after the move, that

position value would decrease by 1, 10, or 100 points respectively. In this way, our AI favored positions where it could fill more positions of a winning line that did not already have an opposing piece on it. For example, a line with 3 spots filled by the maximizing player would gain 100 points. A line with 3 spots filled by the maximizing player but 1 spot filled by the minimizing player would gain 100 points but then lose 1 point. In this way, the first position would be favored over the second position.

Conclusion:

Overall our AI was fairly consistent with what it could beat. There are a few ways that this program can be improved. Firstly, we utilize a depth of 2 currently which consistently beats everything we have played against. However, assuming another AI program goes to a depth of 3, our AI will most likely lose every single game due to the deeper search. However, there is an issue with computational time and timing out the turn. Therefore, it would be possible to institute a deeper search if the number of available turns was bigger than 14 for example. Otherwise, the other method of implementing this AI would be multithreading utilizing computer cores so that we can search deeper and faster than any other AI. Assuming a connection when the ideal move constantly gets updated to the terminal node and both minimax and heuristics are implemented, this AI would be virtually unbeatable.