I represented the state for a tic tac toe game as a 2d char array where the row and column directly translated to a position on a 3 by 3 tic tac toe board and a boolean player. I originally started with a Node subclass that each contained the board state, children, parent, and a boolean player but found that all unnecessary after implementing the minimax functions as minimax was coded recursively and implemented as two separate functions where one, the helper function, returned the utility value of a specific state. The other function returned a string representing the best action.

I made my evaluate function take in a 2d char array representing a board state and return 1 if player ‘X’ won, -1 if player ‘O’ won, 0 if it was a tie, and -2 if the game wasn’t over. It doesn’t return a value for non terminal board states other than -2 as the minimax functions reach the terminal states pretty quick so there was no need for a cutoff and a heuristic estimate for non terminal states.

The printBoard and movesLeft functions are self explanatory, the printBoard function takes a 2d char array input and outputs the board to System.err while movesLeft returns true if there are any moves available and false otherwise.

My two functions for minimax are miniMaxValue and miniMax. The function miniMaxValue takes in a 2d char array, a boolean player, and two ints alpha and beta that are initialized to -infinity and infinity respectively as alpha represents the maximum lower bound which is the best move for player ‘X’ and beta represents the minimum upper bound which represents the best move for player ‘O’ and returns the best utility value of the inputted state. I tested implementing alpha beta pruning in the simpler tic tac toe program before trying to implement it in part 2. The miniMax function uses miniMaxValue inside of it and the miniMax function takes in a state, and, for every possible move from that state, retrieves a utility value of the resulting state from the action and keeps track of the move with the best utility value, ultimately returning that action in the very end. As my tic tac toe program allows the computer to go first, the miniMax function takes in a boolean isPlayerX representing whether or not player X is the user or computer and player ‘X’ is always the maximizing player while player ‘O’ is always the minimizing player so the miniMax function calculates the best move for the computer whether the computer is playing as ‘X’ or ‘O’ through the use of a if else statement.

Due to the way the board is represented, I needed to make two move functions, one that took in a single integer from 1-9 and another that took in a row and column input and returned a single integer from 1-9 representing position. When playing, it is better for the user to just enter a single number representing board position and so if the board is to be updated that would need to be converted to a row and column. Likewise, when the computer makes a move, a single integer from 1-9 representing the move it made needs to be printed to System.out which is why the other move function takes in a row and column input.

When the program is run, it first asks the user to decide between playing as ‘X’ or ‘O’ and if the player chooses ‘X’, prompts the user to choose a position from 1-9 with the numbers 1-3 representing the first row of the board and so on. If the player chooses ‘O’, it does the same thing except the computer always chooses the first position on the board and prints to Standard.out a single number representing the position it played after every move it makes (which is 1 the first time). The reason it chooses 1 is because the miniMax function for an empty board state doesn't give a better utility value than placing a piece in the first position and since it always checks the state that results from moving in the first spot available and records that move, that move is returned by the miniMax function and not a different move as the first move is never overwritten as stated before, the state that results from moving in any other spot doesn't return a better utility value. If I made a heuristic function that evaluates non terminal board states then it could possibly make the computer choose a different starting play but this seemed to be useless as, assuming both players play optimally, all games played ended in a tie and it would be logical to see that there is no way for either player to guarantee a win assuming both players play optimally.

In terms of playing speed, the tic tac toe program played nearly instantaneously so clearly the recursive minimax algorithm didn't go deep enough to encounter problems so there was no need to come up with a heuristic function and a depth cutoff, unlike for adv tic tac toe.

For adv tic tac toe, I represented a board state as a 2d char array again except the "row" represented the board number (ranging from 0-8) while the "column" represented a board (ranging from 1-9). In hindsight this actually made things more complicated as the disagreement (no 1:1 correspondence) between the position played and the board number made it so that I had to increment or decrement stored values. It would have been far simpler to make both numbers range from 0-8 or both range from 1-9.

I originally coded a function randomStartGame where the computer plays random valid moves to try to come up with some sort of strategy to play the game before coding a heuristic function. I made an evaluate function that takes in a 2d char array and returns values for a win, loss, tie, or an incomplete game. Then, a separate heuristic function was made that calls evaluate within it and makes the function return 10000 if the board is a terminal board and 'X' won, -10000 if 'O' won, or returns a score value if the board is a non terminal board. Originally, I had the score calculated by going through each board individually, and squaring the difference of the numbers of 'X's and 'O's and making it negative if there were more 'O's than 'X's then summing the scores of the boards and returning that sum. This stemmed from my original idea that it was better to concentrate your pieces rather than spread them out but this turned out to not be true after coding a different heuristic function.

The heuristic function then became a simpler one where it added 50 to the score if there were two 'X's in a row with an empty spot in the third position and -50 if it was two 'O's. This seemed to be a good heuristic as the utility score it gave wasn't too high and it would never be better than the value of a winning board state. Also, a board having two in a row with an empty spot in the third position is a better state than one without that two in a row.

Originally, the miniMaxValue function went to depth 7 before terminating and it would take the computer around 2 seconds to make the first move and later in the game it would sometimes take a bit less time as there weren't as many non terminal states when there are more pieces on the board. After implementing the alpha beta pruning where it would break the state search whenever alpha was greater than or equal to beta, I could set the depth to 10 and it would take the computer the same amount of time (around 2 seconds) to make the first move. In comparison, the miniMaxValue function without alpha beta pruning would not make the first move in a reasonable amount of time (under a few minutes) if I set the cutoff depth to 10 so alpha beta pruning resulted in a large improvement in runtime and it only took a few extra lines of code.

Although for adv tic tac toe, the project description did not explicitly state that the user can have a choice of going first or second, I did try to implement it but it did not work the same way it did for the regular tic tac toe program. I could not figure out what the problem was even though intuitively, since both my tic tac toe and adv tic tac toe programs followed general structure, implementing a player choice for going first or second (playing as 'X' or 'O') should work the same way but it didn't. Furthermore, my heuristic function is very simple right now and it could be improved a lot more but I could not figure out a surefire strategy to win a game. Originally, when I played my current iteration of adv tic tac toe, I tried to employ a strategy where I focused more on what board I would send the bot to play rather than seeing what the best move in the current board was but that eventually turned into the bot having more winning boards than me (a winning board being a board where there are two of your pieces in a row without the third spot being filled) which meant that if I set the bot to any of those boards, I would just lose instantly. I had more success with random, non losing moves (a losing move being a move where I send the bot to a winning board).

To make a better heuristic, it might have been worthwhile to have the bot play itself and print the steps it took to get to a game end state and build a strategy from what the bot did based on my heuristic and maybe be able to improve my heuristic from those observations. Currently, the heuristic function I implemented has the bot try to get to states that result in it having more winning board where a winning board is any board that it would win if it got sent to play on that board first. Ultimately, after playing my game numerous times, the bot did end up winning more times than it lost and from what I've seen, it generally makes logical moves although there were some instances where it sent me to one of my winning boards, resulting in the bot losing, although there were alternative plays it could have made.