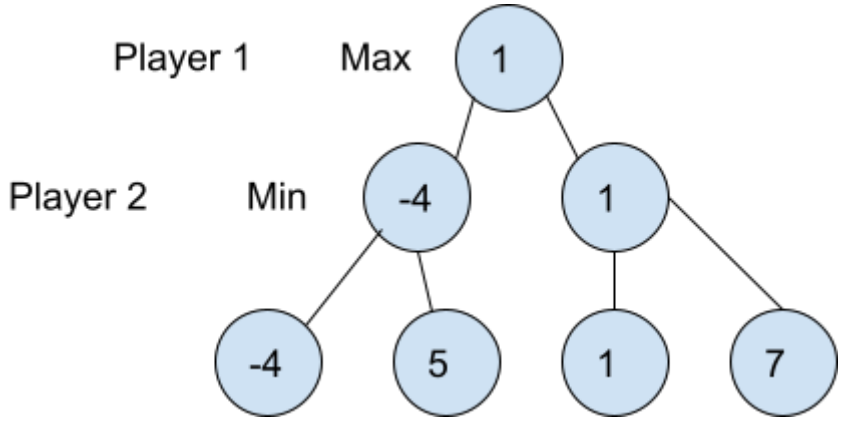
Introduction:

In this project, our objective is to implement an adversarial search algorithm, specifically a minimax algorithm, which is used to play tic-tac-toe. The plan is to create the AI so it can never lose, only tie or win. This will be tested by playing against the AI and recording its wins, ties, and losses. To create this minimax algorithm we intend to use the skeleton algorithm given to us, the help of our TAs, and any external sources we find that are useful. We are using Python to code the algorithm.

Methods:

Each player is trying to maximize their score and minimize their opponent's score and they get alternating turns in the game, which creates a recursion through the max value finder function and the min value finder. When it is the other players turn they have to choose the best option from the worst options their opponents picked for them. This explanation can be seen in the tree drawn below:



The final nodes have the determined value and the Player 2 chooses the smallest outcome because it wants to minimize the Player 1's outcome, which maximizes its outcome. Then Player 1 chooses the maximum of the two outcomes because it wants to maximize its score, which minimizes Player 2's score.

```
def max_helper(asp, state, player):
    if asp.is_terminal_state(state):
        e = asp.evaluate_terminal(state)
        return e[player]
    v = float("-inf")
    for a in asp.get_available_actions(state):
        next_state = asp.transition(state, a)
        v= max(v, min_helper(asp, next_state, player))
    return v
```

-Def Max Overall:

This function takes in the class AdversarialSearchProblem the current state of the board and the player that is on their turn. The output is the value of the max value of one of the children states of the current state passed through the function.

-Step By Step Analysis:

First, the function checks if the game is over and returns the player's score and the scores of the other player. If the game is not over, the function then creates a variable v which will be set to infinity at first so any value found later will be bigger than v now. Then for each action that is available to the player at this board state, it gets the next state resulting from that action. Finally, the v is made the max of the current value of v or the minimum value of the next state because of the alternating players. This value is then returned.

```
def min_helper(asp, state, player):
    if asp.is_terminal_state(state):
        e = asp.evaluate_terminal(state)
        return e[player]
    v = float("inf")
    for a in asp.get_available_actions(state):
        next_state = asp.transition(state, a)
        v= min(v, max_helper(asp, next_state, player))
    return v
```

-Def Min Overall:

This function takes in the class AdversarialSearchProblem the current state of the board and the player that is on their turn. The output is the value of the min value of one of the children states of the current state passed through the function.

-Step By Step Analysis:

First, the function checks if the game is over and returns the player's score and the scores of the other player. If the game is not over, the function then creates a variable v which will be set to infinity at first so any value found later will be smaller than v now. Then for each action that is available to the player at this board state, it gets the next state resulting from that action. Finally, v becomes the minimum value between the current v and the maximum value of the next state because of the alternating players.

```
def minimax(asp: AdversarialSearchProblem[GameState, Action]) -> Action:

    state = asp.get_start_state()
    player = state.player_to_move()
    bestMove = None
    bestVal = float("-inf")
    for a in asp.get_available_actions(state):
        next_state = asp.transition(state, a)
        val = min_helper(asp, next_state, player)
```

```
        if val > bestVal:
            bestMove = a
            bestVal = val
```

-Def Minimax Overall:

This function takes in the class AdverasrialSearchProblem and returns the best action for the current player.

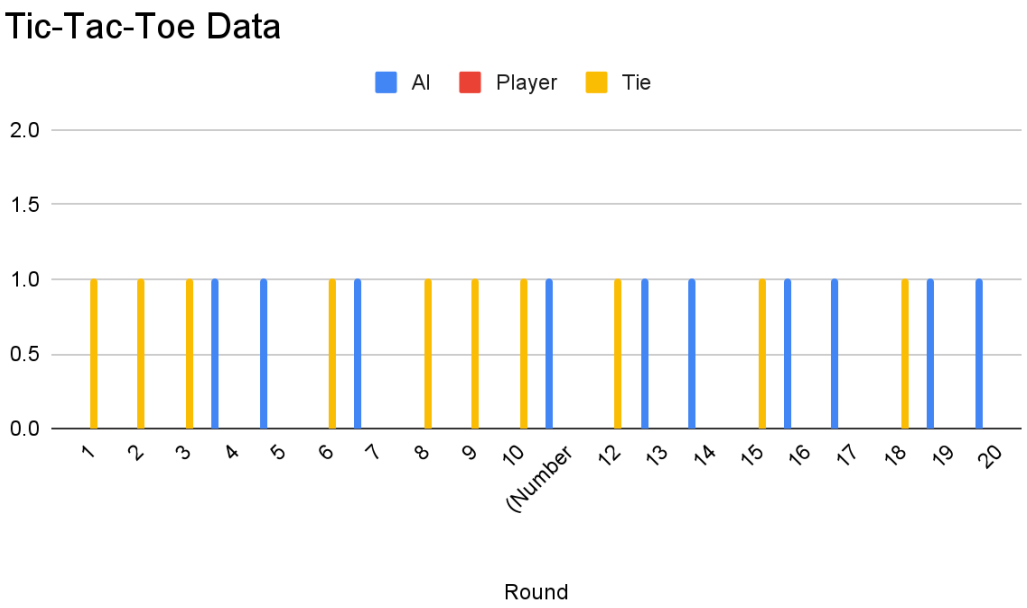
-Step By Step Analysis:

First, the start state of the board is placed in the variable state, which would be the blank board. The player whose move the game is on is assigned the variable player. Then two variables, one that will hold the best action that the player should take and one that holds the value of that best action. The best action is the one that maximizes the current player's score and minimizes the opposition's score. The best value is first set to infinity so that any score found later is bigger than that. The function then “kick starts” the minimax process by starting to find the max value because no matter whose turn it is they are trying to max their score. This is done by iterating through all the actions that could come from this state. Then a new variable is made that holds the state that comes from that action. The class AdverasrialSearchProblem, the new state we just found and the player passes into the min function to get the minimum value for the next round. This then starts the recursion through all the functions to find the best path for the AI. If this value is better than the best value we defined before then make that the new best value and assign the bestMove the action that the iteration was on.

To implement this we mainly used help from Lucy and Jackson and their pseudocode that they talk through during office hours. They also helped us debug our code. We also used this video for understanding how minimax works and how you could implement code.

Results:

The result of our algorithm is that it worked. We decided to test the algorithm by playing 20 times against the AI. In doing so, we checked for errors, complications, and confusion in the coding.



-Bar Graph:

The bar graph shows the data from the experiment. The results show us that there was a 50-50 between the AI winning and Ties. This helps us infer that the algorithm is working properly and there are no complications.

Round	AI	Player	Tie
1	0	0	1
2	0	0	1
3	0	0	1
4	1	0	0
5	1	0	0
6	0	0	1
7	1	0	0
8	0	0	1
9	0	0	1
10	0	0	1

(Number Picker Introduced): 11	1	0	0
12	0	0	1
13	1	0	0
14	1	0	0
15	0	0	1
16	1	0	0
17	1	0	0
18	0	0	1
19	1	0	0
20	1	0	0
Total:	10	0	10

Discussions:

Based on what we did to create the algorithm, everything we chose and did was appropriate for the algorithm in particular. As stated earlier, we know this due to running an experiment on the algorithm. Something we would do differently is understand the work and algorithm more and use less of the help offered to us. However saying this, the help given to us was necessary and did help accelerate our coding process. The ethical implications imported are that the players would always lose or get a tie against the AI.