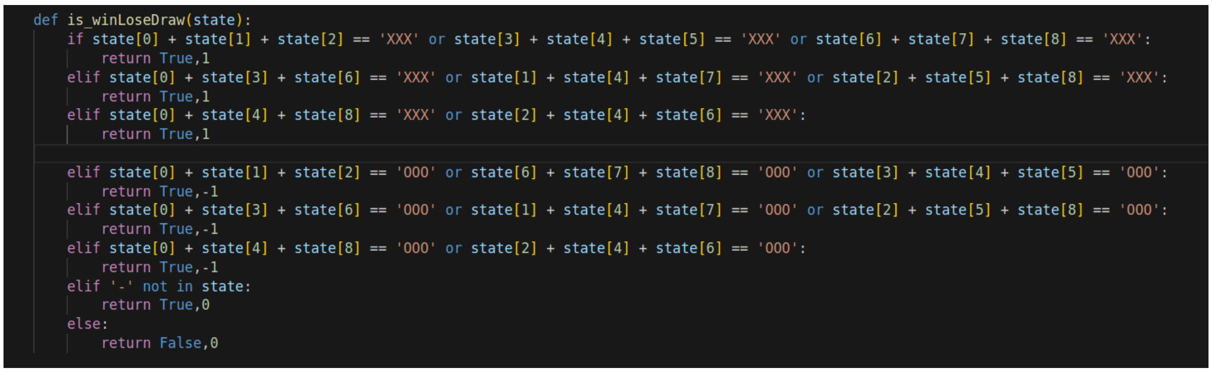
ALGORITHM USED IN TIC TAC TOE

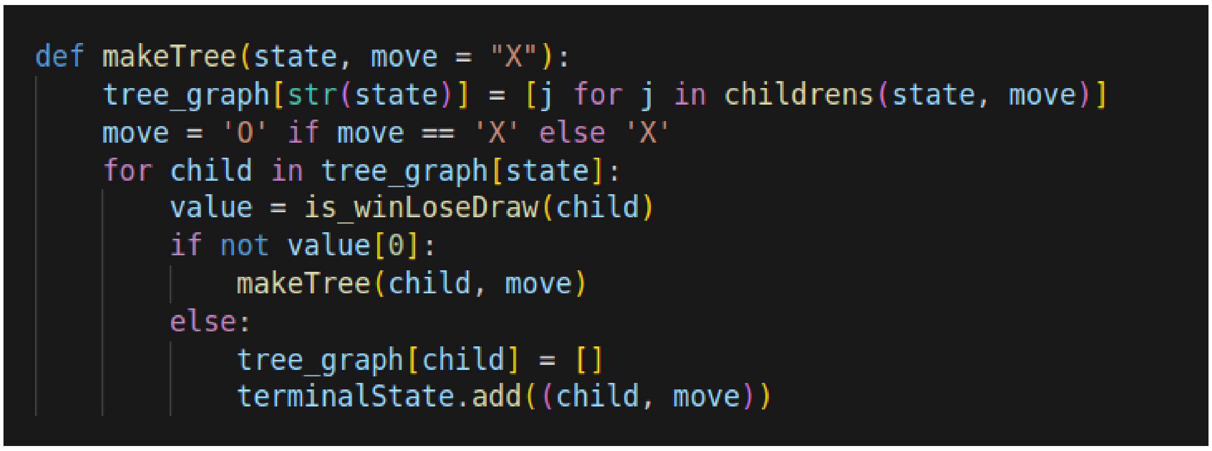
Reinforcement Learning Implementing TicTacToe

1. In this code snippet we are checking if the state has been checking if the state is win , loss or draw. If any of this terminal state is satisfied then it will return the value and true.
   Win value = 1
   Loss value = -1
   Draw value = 0



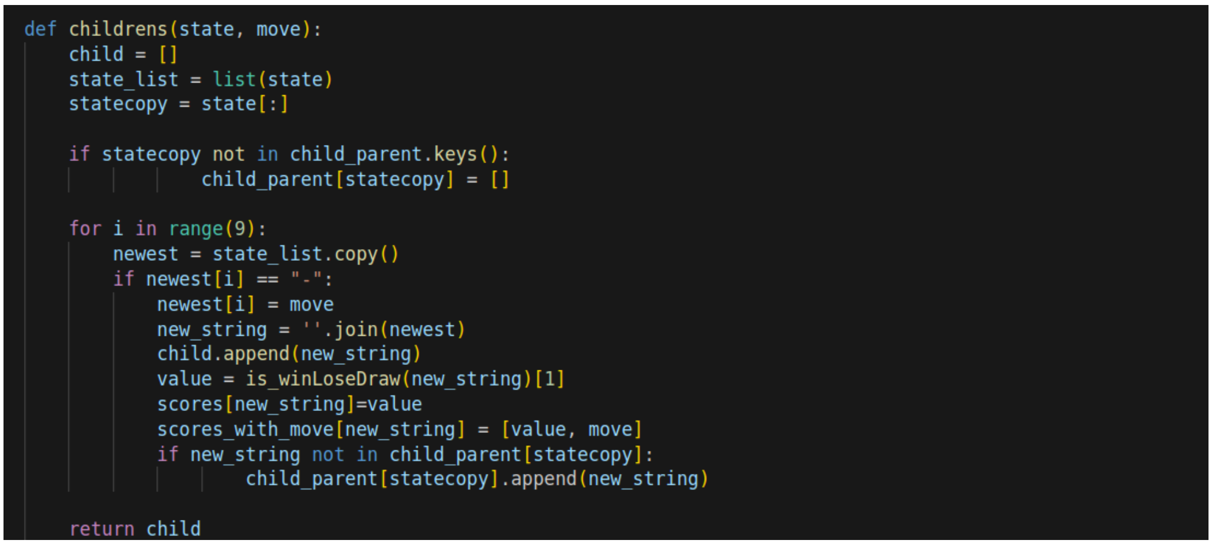
```python
def is_winLoseDraw(state):
    if state[0] + state[1] + state[2] == 'XXX' or state[3] + state[4] + state[5] == 'XXX' or state[6] + state[7] + state[8] == 'XXX':
        return True,1
    elif state[0] + state[3] + state[6] == 'XXX' or state[1] + state[4] + state[7] == 'XXX' or state[2] + state[5] + state[8] == 'XXX':
        return True,1
    elif state[0] + state[4] + state[8] == 'XXX' or state[2] + state[4] + state[6] == 'XXX':
        return True,1

    elif state[0] + state[1] + state[2] == 'OOO' or state[6] + state[7] + state[8] == 'OOO' or state[3] + state[4] + state[5] == 'OOO':
        return True,-1
    elif state[0] + state[3] + state[6] == 'OOO' or state[1] + state[4] + state[7] == 'OOO' or state[2] + state[5] + state[8] == 'OOO':
        return True,-1
    elif state[0] + state[4] + state[8] == 'OOO' or state[2] + state[4] + state[6] == 'OOO':
        return True,-1
    elif '-' not in state:
        return True,0
    else:
        return False,0
```

2. In this method we are generating the possible tree from the initial state of game to all possible and by generating all possible children. This is also changing the move accordingly. If the state is terminal, then it will stop exploring and add it to the terminal dictionary with value.



```python
def makeTree(state, move = "X"):
    tree_graph[str(state)] = [j for j in childrens(state, move)]
    move = 'O' if move == 'X' else 'X'
    for child in tree_graph[state]:
        value = is_winLoseDraw(child)
        if not value[0]:
            makeTree(child, move)
        else:
            tree_graph[child] = []
            terminalState.add((child, move))
```

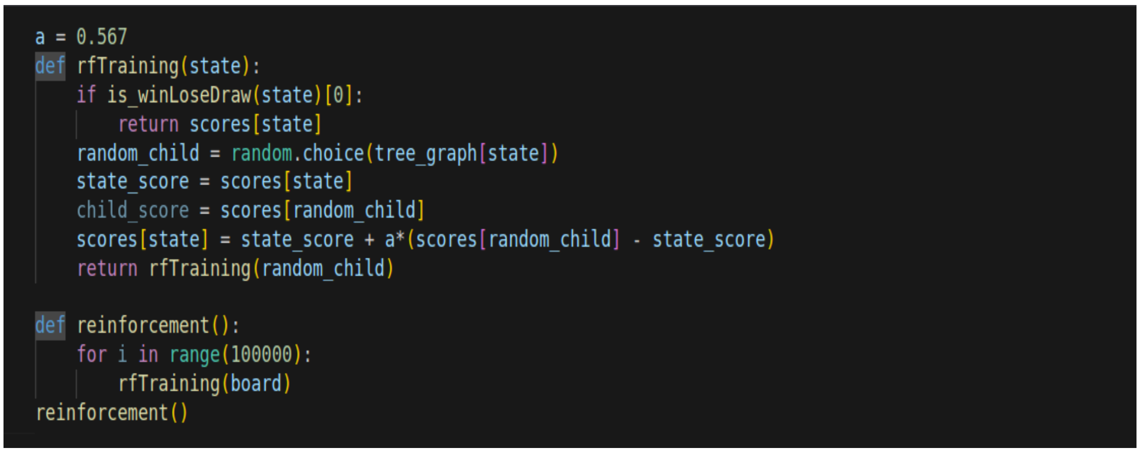
3. This code will generate the children of a given state with move. Here it storing the Scores value with state and also generates a child parent dict with child as key and parents as value.



```python
def childrens(state, move):
    child = []
    state_list = list(state)
    statecopy = state[:]

    if statecopy not in child_parent.keys():
            child_parent[statecopy] = []

    for i in range(9):
        newest = state_list.copy()
        if newest[i] == "-":
            newest[i] = move
            new_string = ''.join(newest)
            child.append(new_string)
            value = is_winLoseDraw(new_string)[1]
            scores[new_string]=value
            scores_with_move[new_string] = [value, move]
            if new_string not in child_parent[statecopy]:
                child_parent[statecopy].append(new_string)

    return child
```

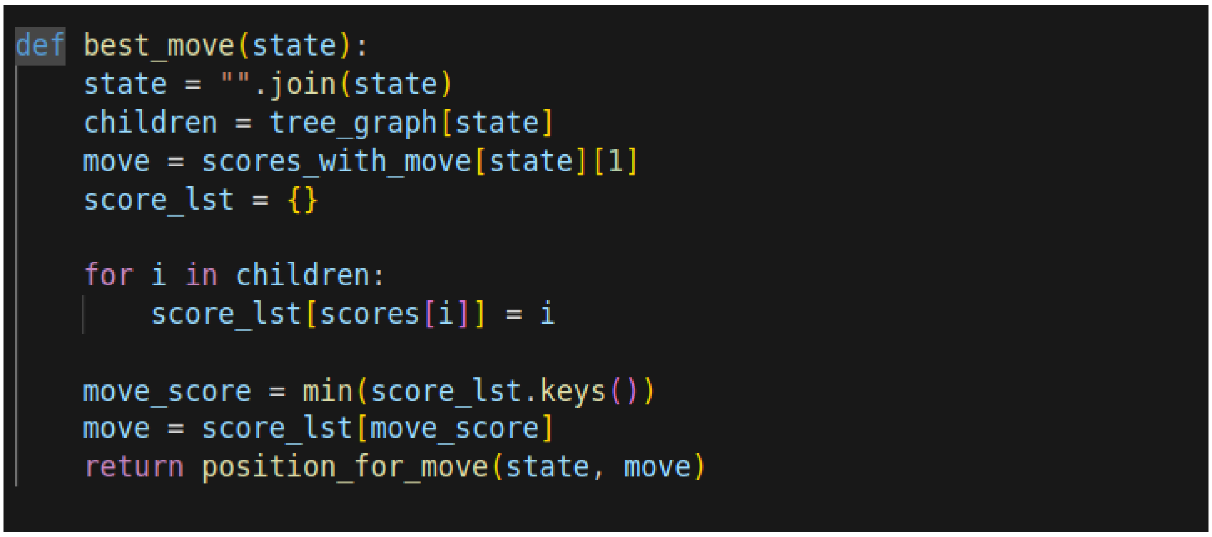
4. Training part: in this method if state is terminal then it will return the scores otherwise it will randomly select the child and iteratively update the state value according to the given algorithms recursively. Here is alpha, that is the learning rate.

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[V(S_{t+1}) - V(S_t)\Big]$$



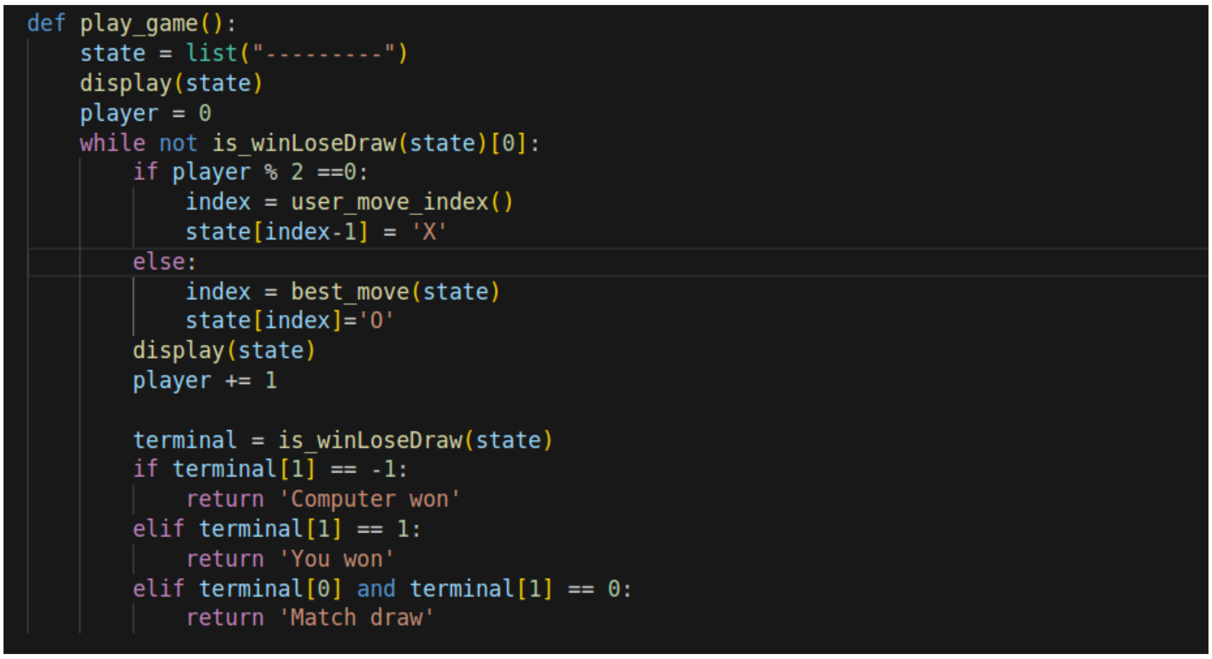
```python
a = 0.567
def rfTraining(state):
    if is_winLoseDraw(state)[0]:
        return scores[state]
    random_child = random.choice(tree_graph[state])
    state_score = scores[state]
    child_score = scores[random_child]
    scores[state] = state_score + a*(scores[random_child] - state_score)
    return rfTraining(random_child)

def reinforcement():
    for i in range(100000):
        rfTraining(board)
reinforcement()
```

5. In this case the computer will choose the value of children of state and find the min score among the children and return the state.



```python
def best_move(state):
    state = "".join(state)
    children = tree_graph[state]
    move = scores_with_move[state][1]
    score_lst = {}

    for i in children:
        score_lst[scores[i]] = i

    move_score = min(score_lst.keys())
    move = score_lst[move_score]
    return position_for_move(state, move)
```

6. And finally you can play the game by running the play_game() and declaring the win , loss or draw.
7.



```python
def play_game():
    state = list("---------")
    display(state)
    player = 0
    while not is_winLoseDraw(state)[0]:
        if player % 2 ==0:
            index = user_move_index()
            state[index-1] = 'X'
        else:
            index = best_move(state)
            state[index]='O'
        display(state)
        player += 1

        terminal = is_winLoseDraw(state)
        if terminal[1] == -1:
            return 'Computer won'
        elif terminal[1] == 1:
            return 'You won'
        elif terminal[0] and terminal[1] == 0:
            return 'Match draw'
```

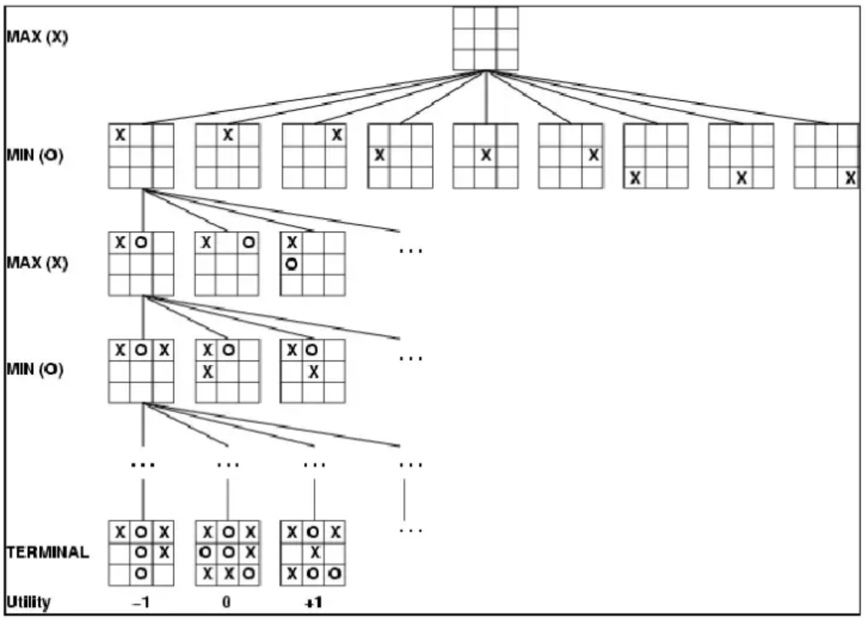
Min max algorithm

Min max algorithms are used here to compute the optimal move in the game.
This algorithm that I made is using the DFS, a search algorithm for the making of game trees. I had stored the tree in the dictionary as parent as key and its possible children as its value in list. And there is a score dictionary which stores each state associated with their value and its turn.

ADVANTAGE
The Minimax algorithm helps to find the best move, by working backwards from the end of the game. At each step it assumes that the player is trying to maximise the chances of its winning, while on the next turn the player is trying to minimise the chances of A's winning.

DISADVANTAGE
A disadvantage of the minimax algorithm is that each board state has to be visited twice: one time to find its children and a second time to evaluate the value.

GAME TREE



Reference:

In min max code: taken help from bharat