Lab # 7

Adversarial Search - Minmax

7.1 Objective:

Learn basics of adversarial search and to implement minmax algorithm

7.2 Scope:

The student should know the following:

- To implement tic tac toe game, human vs computer
- To use minmax for embedded thinking

7.3 Useful Concepts:

Tic Tac Toe

Let's start by defining what it means to play a perfect game of tic tac toe:

If you play perfectly, every time you play, you will either win the game, or you will draw the game. Furthermore, if you play against another perfect player, you will always draw the game.How might we describe these situations quantitatively? Let's assign a score to the "end game conditions:" you win, hurray! you get 10 points! If you lose. you lose 10 points (because the other player gets 10 points) You draw, whatever. you get zero points; nobody gets any points. So now we have a situation where we can determine a possible score for any game end state.

Example

To apply this, let's take an example from near the end of a game, where it is my turn. You are X. Your goal here, obviously, is to maximize my end game score.

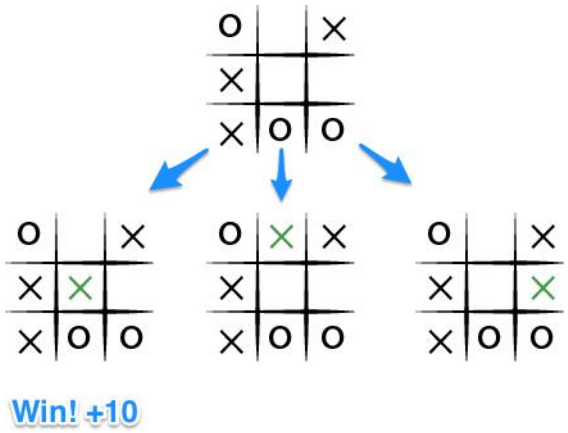

Figure 7.1

If the top of this image represents the state of the game you see when it is your turn, then you have some choices to make, there are three places you can play, one of which clearly results in you wining and earning the 10 points. If you don't make that move, O could very easily win. And you don't want O to win, so my goal here, as the first player, should be to pick the maximum scoring move.

Story of the other side

What do we know about O? Well, we should assume that O is also playing to win this game, but relative to us, the first player, O wants obviously wants to choose the move that results in the worst score for us, it wants to pick a move that would minimize our ultimate score. Let's look at things from O's perspective, starting with the two other game states from above in which we don't immediately win:

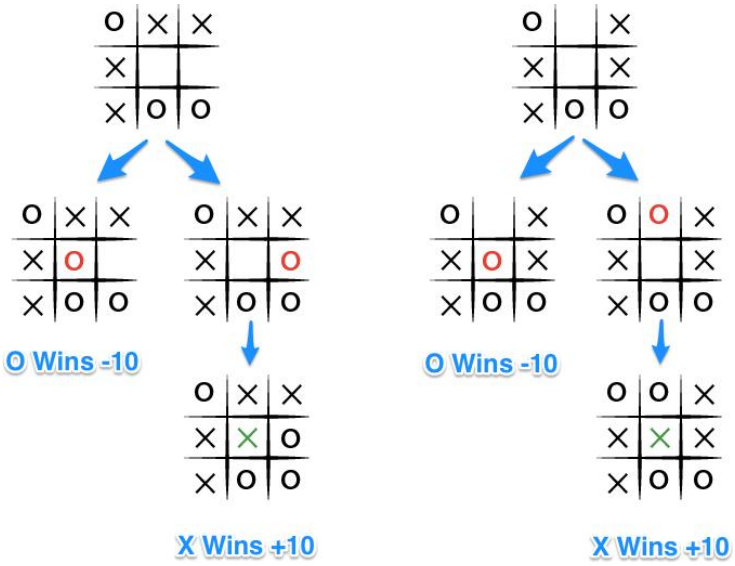

Figure 7.2

The choice is clear, O would pick any of the moves that result in a score of -10.

Minimax

The key to the Minimax algorithm is a back and forth between the two players, where the player whose "turn it is" desires to pick the move with the maximum score. In turn, the scores for each of the available moves are determined by the opposing player deciding which of its available moves has the minimum score. And the score for the opposing players moves are again determined by the turn-taking player trying to maximize its score and so on all the way down the move tree to an end state.

A description for the algorithm, assuming X is the "turn taking player," would look something like:

- If the game is over, return the score from X's perspective.
- Otherwise get a list of new game states for every possible move
- Create a scores list
- For each of these states add the minimax result of that state to the scores list
- If it's X's turn, return the maximum score from the scores list
- If it's O's turn, return the minimum score from the scores list

You'll notice that this algorithm is recursive, it flips back and forth between the players until a final score is found. Let's walk through the algorithm's execution with the full move tree, and show why, algorithmically, the instant winning move will be picked:

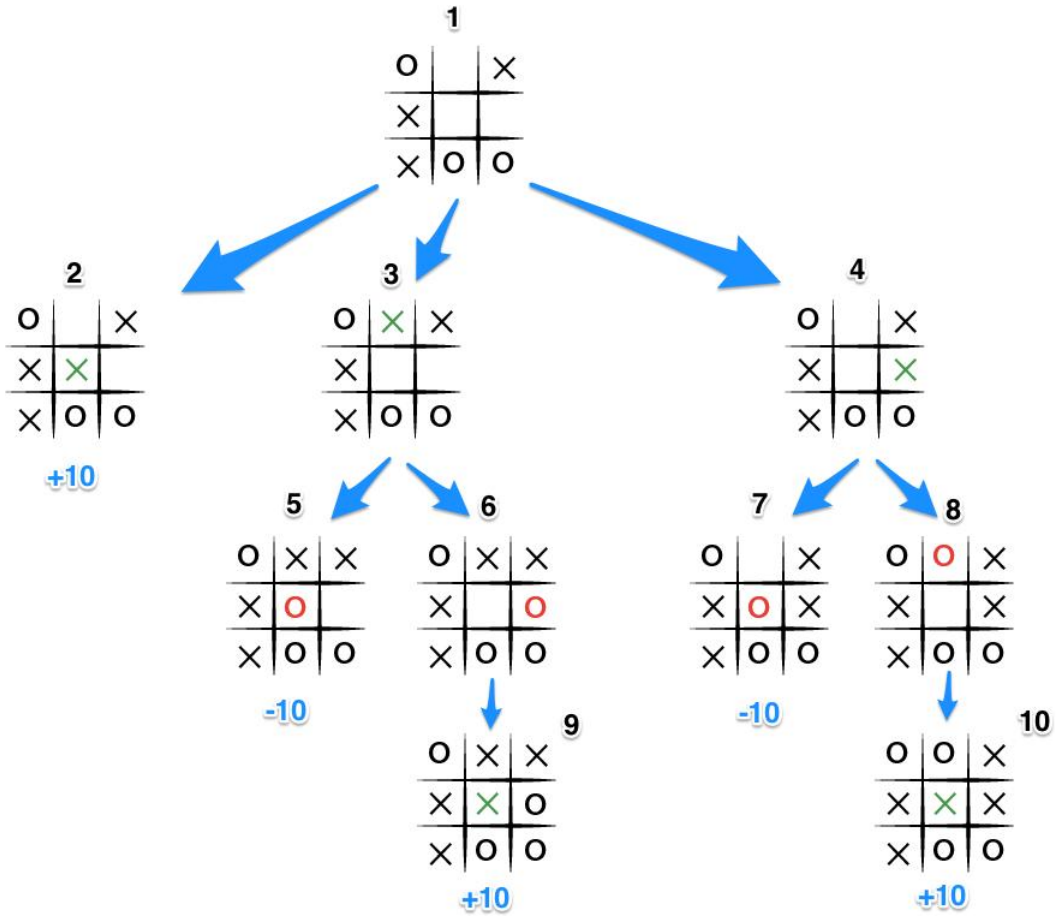

Figure 7.3

- It's X's turn in state 1. X generates the states 2, 3, and 4 and calls minimax on those states.
- State 2 pushes the score of +10 to state 1's score list, because the game is in an end state.
- State 3 and 4 are not in end states, so 3 generates states 5 and 6 and calls minimax on them, while state 4 generates states 7 and 8 and calls minimax on them.
- State 5 pushes a score of -10 onto state 3's score list, while the same happens for state 7 which pushes a score of -10 onto state 4's score list.
- State 6 and 8 generate the only available moves, which are end states, and so both of them add the score of +10 to the move lists of states 3 and 4.
- Because it is O's turn in both state 3 and 4, O will seek to find the minimum score, and given the choice between -10 and +10, both states 3 and 4 will yield -10.
- Finally, the score list for states 2, 3, and 4 are populated with +10, -10 and -10 respectively, and state 1 seeking to maximize the score will chose the winning move with score +10, state 2.

Here is the function for scoring the game:

```
# @player is the turn taking player
def score(game)
    if game.win?(@player)
        return 10
    elsif game.win?(@opponent)
        return -10
    else
        return 0
    end
end
```

Figure 7.4

Simple enough, return +10 if the current player wins the game, -10 if the other player wins and 0 for a draw. You will note that who the player is doesn't matter. X or O is irrelevant, only who's turn it happens to be. And now the actual minimax algorithm; note that in this implementation a choice or move is simply a row / column address on the board, for example [0,2] is the top right square on a 3x3 board.

```
def minimax(game)
    return score(game) if game.over?
    scores = [] # an array of scores
    moves = []  # an array of moves

    # Populate the scores array, recursing as needed
    game.get_available_moves.each do |move|
        possible_game = game.get_new_state(move)
        scores.push minimax(possible_game)
        moves.push move
    end

    # Do the min or the max calculation
    if game.active_turn == @player
        # This is the max calculation
        max_score_index = scores.each_with_index.max[1]
        @choice = moves[max_score_index]
        return scores[max_score_index]
    else
        # This is the min calculation
        min_score_index = scores.each_with_index.min[1]
        @choice = moves[min_score_index]
        return scores[min_score_index]
    end
end
```

Figure 7.5

When this algorithm is run inside a PerfectPlayer class, the ultimate choice of best move is stored in the @choice variable, which is then used to return the new game state in which the current player has moved.

##A Perfect but Fatalist Player Implementing the above algorithm will get you to a point where your tic tac toe game can't be beat. But and interesting nuance that I discovered while testing is that a perfect player must always be perfect. In other words, in a situation where the perfect player eventually will lose or draw, the decisions on the next move are rather fatalistic. The algorithm essentially says: "hey I'm gonna lose anyway, so it really doesn't matter if I lose in the next more or 6 moves from now."

I discovered this by passing an obviously rigged board, or one with a "mistake" in it to the algorithm and asked for the next best move. I would have expected the perfect player to at least put up a fight and block my immediate win. It however, did not:

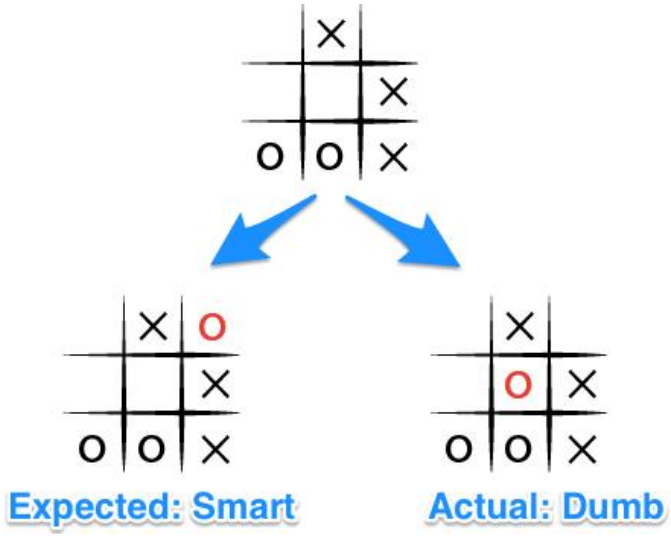

Figure 7.6

Let's see what is happening here by looking through the possible move tree (some of the moves for clarity)

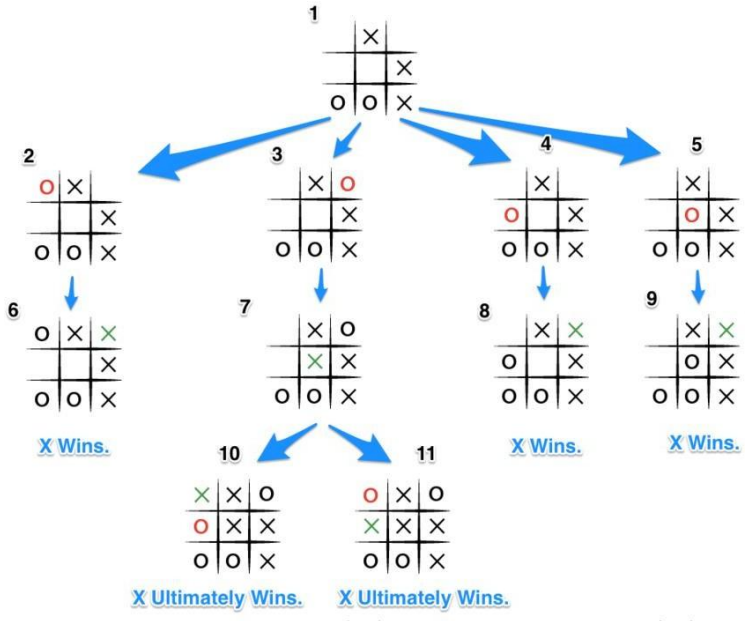

Figure 7.7

Given the board state 1 where both players are playing perfectly, and O is the computer player. O choses the move in state 5 and then immediately loses when X wins in state 9. But if O blocks X's win as in state 3, X will obviously block O's potential win as shown in state 7. This puts two certain wins for X as shown in state 10 and 11, so no matter which move O picks in state 7, X will ultimately win.

As a result of these scenarios, and the fact that we are iterating through each blank space, from left to right, top to bottom, all moves being equal, that is, resulting in a loss for O, the last move will be chosen as shown in state 5, as it is the last of the available moves in state 1. The array of moves being: [top-left, top-right, middle-left, middle-centre].

The key improvement to this algorithm, such that, no matter the board arrangement, the perfect player will play perfectly unto its demise, is to take the "depth" or number of turns till the end of the game into account. Basically, the perfect player should play perfectly, but prolong the game as much as possible.

In order to achieve this we will subtract the depth that is the number of turns, or recursions, from the end game score, the more turns the lower the score, the fewer turns the higher the score. Updating our code from above we have something that looks like this:

Pseudo-code

```
def score(game, depth)
    if game.win?(@player)
        return 10 - depth
    elsif game.win?(@opponent)
        return depth - 10
    else
        return 0
    end
end
```

Figure 7.8

```
def minimax(game, depth)
    return score(game) if game.over?
    depth += 1
    scores = [] # an array of scores
    moves = []  # an array of moves

    # Populate the scores array, recursing as needed
    game.get_available_moves.each do |move|
        possible_game = game.get_new_state(move)
        scores.push minimax(possible_game, depth)
        moves.push move
    end

    # Do the min or the max calculation
    if game.active_turn == @player
        # This is the max calculation
        max_score_index = scores.each_with_index.max[1]
        @choice = moves[max_score_index]
        return scores[max_score_index]
    else
        # This is the min calculation
        min_score_index = scores.each_with_index.min[1]
        @choice = moves[min_score_index]
        return scores[min_score_index]
    end
```

Figure 7.9

Thus each time we invoke minimax, depth is incremented by 1 and when the end game state is ultimately calculated, the score is adjusted by depth. Let's see how this looks in our move tree

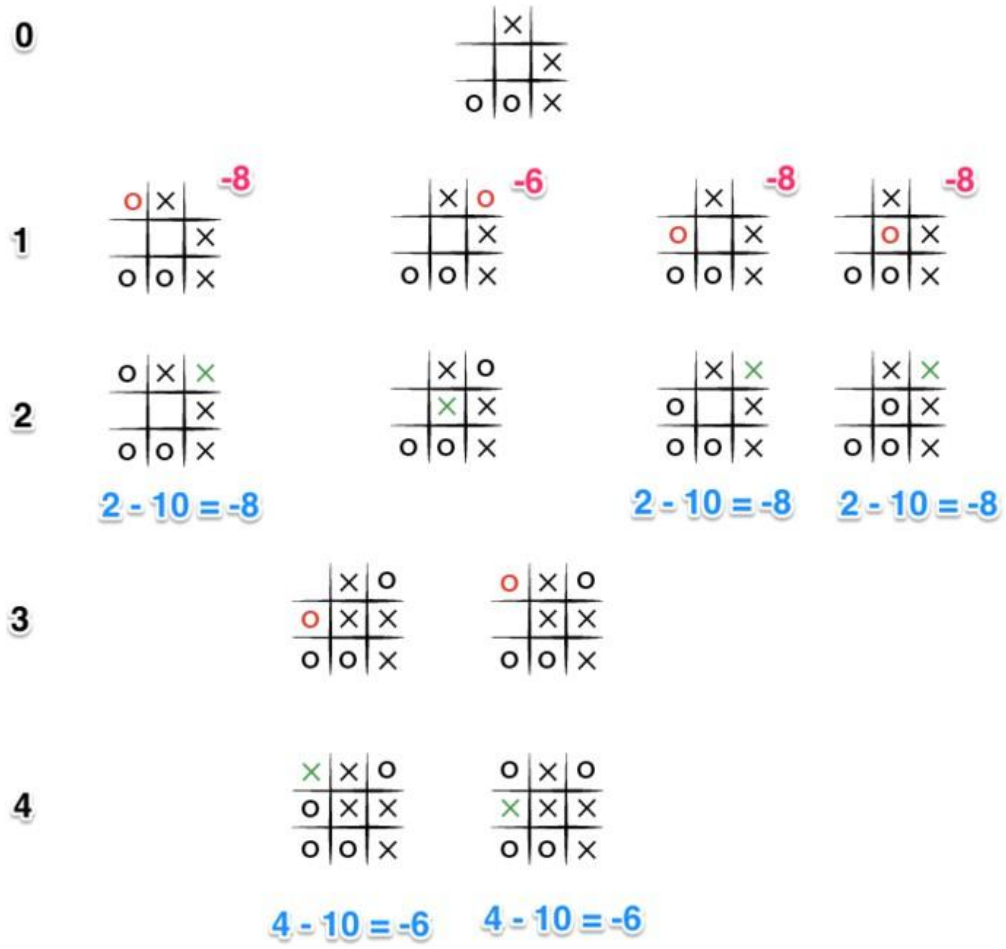

Figure 7.10

This time the depth (Shown in black on the left) causes the score to differ for each end state, and because the level 0 part of minimax will try to maximize the available scores (because O is the turn taking player), the -6 score will be chosen as it is greater than the other states with a score of -8. And so even faced with certain death, our trusty, perfect player now will choose the blocking move, rather than losing.

Implementation example

```
from ipywidgets import widgets, HBox, VBox, Layout
from IPython.display import display
from functools import partial
import numpy as np
#import tictactoe
import tictactoe_minimax_helper as minimax_helper
```

```
class General_functions(object):
    def __init__(self, matrix, actual_turn):
        self.N = 3
        self.button_list = None
        self.text_box = None
        self.matrix = matrix
        self.game_finished = False
        self.actual_turn = actual_turn

    def display_matrix(self):
        N = self.N
        childs = []
        for i in range(N):
            for j in range(N):
                if self.matrix[i,j]==1:
                    self.button_list[i*N + j].description = 'o'
                if self.matrix[i,j]==-1:
                    self.button_list[i*N + j].description = 'x'

    def on_button_clicked(self, index, button):
        N = self.N

        if self.game_finished:
            return

        y = index%N
        x = int(index/N)
        if self.matrix[x,y]!=0:
            self.text_box.value = 'No se puede ahi!'
            return
        button.description = self.actual_turn[0]

        if self.actual_turn == 'o':
            self.matrix[x,y] = 1
            self.game_finished, status = minimax_helper.game_over(self.matrix)
            if self.game_finished:
                if (status!=0):
                    self.text_box.value = 'o wins'
                else:
                    self.text_box.value = 'draw'
            else:
                self.actual_turn = 'x'

                self.text_box.value = 'Juega '+self.actual_turn
        else:
            self.matrix[x,y] = -1
            self.game_finished, status = minimax_helper.game_over(self.matrix)
            if self.game_finished:
                if (status!=0):
                    self.text_box.value = 'x wins'
                else:
                    self.text_box.value = 'draw'
            else:
                self.actual_turn = 'o'
                self.text_box.value = 'Juega '+self.actual_turn
        self.computer_play()
```

```
def draw_board(self):
    self.text_box = widgets.Text(value = 'Juega '+self.actual_turn, layout=Layout(width='129px', height='40p
    self.button_list = []
    for i in range(9):
        button = widgets.Button(description='',
        disabled=False,
        button_style='', # 'success', 'info', 'warning', 'danger' or ''
        tooltip='Click me',
        icon='',
        layout=Layout(width='40px', height='40px'))
        self.button_list.append(button)
        button.on_click(partial(self.on_button_clicked, i))
    tic_tac_toe_board = VBox([HBox([self.button_list[0],self.button_list[1],self.button_list[2]]),
            HBox([self.button_list[3],self.button_list[4],self.button_list[5]]),
            HBox([self.button_list[6],self.button_list[7],self.button_list[8]])])
    display(VBox([self.text_box, tic_tac_toe_board]))
    return

def computer_play(self):

    if self.game_finished:
        return

    if self.actual_turn=='x':
        turn = -1
        next_turn = 'o'
    if self.actual_turn=='o':
        turn = 1
        next_turn = 'x'

        self.matrix = self.get_best_play(turn)
        self.display_matrix()
        self.actual_turn = next_turn
        self.text_box.value = 'Juega '+self.actual_turn
        self.game_finished, status = minimax_helper.game_over(self.matrix)
        if self.game_finished:
            if (status!=0):
                self.text_box.value = 'computer wins'
            else:
                self.text_box.value = 'draw'

    def get_best_play(self, turn):
        # 1000 is an infinite value compared with the highest cost of 10 that we can get

        choice, points, nodes_visited = minimax_helper.minimax(self.matrix, turn)
        print('points:',points)
        print('nodes_visited:',nodes_visited)
        return choice
```

```python
def start_game(computer_starts = True, user_icon='x', start_mode = 'center'):
    matrix = np.zeros((3,3))

    if user_icon=='x':
        computer_icon_representation = 1
    else:
        computer_icon_representation = -1

    GF = General_functions(matrix, user_icon)
    GF.draw_board()

    if computer_starts:
        if start_mode == 'center':
            matrix[1,1] = computer_icon_representation
        elif start_mode == 'minimax':
            GF.computer_play()
        elif start_mode == 'random':
            x = np.random.randint(3)
            y = np.random.randint(3)
            matrix[x,y] = computer_icon_representation

    GF.display_matrix()
```

```python
# start_mode:
# 'minimax': Uses minimax to select the first move
# 'center': Starts on the center
# 'random': Starts on a random position
# user_icon:
#  'x': user is x
#  'o': user is o
# computer_starts: True or False

start_game(computer_starts = True, user_icon = 'x', start_mode = 'random')
```

Output: Computer Wins


```
points: 0
nodes_visited: 2382
points: 7
nodes_visited: 102
points: 9
nodes_visited: 10
```

Output: Draw


```
points: 0
nodes_visited: 1489
points: 0
nodes_visited: 128
points: 0
nodes_visited: 9
points: 0
nodes_visited: 1
```

Helper functions

Function: Get next turn

```python
def get_next_turn(active_turn):
    # Change player
    # 1 -> o
    #-1 -> x
    if active_turn == 1:
        next_turn = -1
    else:
        next_turn = 1
    return next_turn
```

Function: Get Childs

```python
def get_childs(matrix, turn):
    # turn is 1 or -1
    # 1 -> o
    #-1 -> x
    # 0 -> Free space
    # return all posible plays for user 'turn' ('x' or 'o')
    N = 3
    childs = []
    for i in range(N):
        for j in range(N):
            if matrix[i,j]==0:
                child = matrix.copy()
                child[i,j] = turn
                childs.append(child)
    return childs
```

Function: Get Status

```
def game_status(matrix):
    # Returns 1 if 'o' win, -1 if 'x' win, 0 if draw
    points = 1
    if (matrix[0,:].sum() == 3)|(matrix[1,:].sum() == 3)|(matrix[2,:].sum() == 3)|(matrix[:,0].sum() == 3)|(matrix[:,
    1].sum() == 3)|(matrix[:,2].sum() == 3):
        return points
    if (matrix[0,0]==matrix[1,1])&(matrix[2,2]==matrix[1,1])&(matrix[0,0]==1):
        return points
    if (matrix[0,2]==matrix[1,1])&(matrix[2,0]==matrix[1,1])&(matrix[2,0]==1):
        return points
    if (matrix[0,:].sum() == -3)|(matrix[1,:].sum() == -3)|(matrix[2,:].sum() == -3)|(matrix[:,0].sum() == -3)|(
    matrix[:,1].sum() == -3)|(matrix[:,2].sum() == -3):
        return -points
    if (matrix[0,0]==matrix[1,1])&(matrix[2,2]==matrix[1,1])&(matrix[0,0]==-1):
        return -points
    if (matrix[0,2]==matrix[1,1])&(matrix[2,0]==matrix[1,1])&(matrix[2,0]==-1):
        return -points
    return 0
```

Function: Game over

```
def game_over(matrix):
    # status <- Returns 1 if 'o' win, -1 if 'x' win, 0 if draw
    # game_finished: true is game is over
    game_finished = False
    status = game_status(matrix)
    if status!=0:
        #Game finishes, someone won
        #Write code here
    if abs(matrix).sum() == 9:
        #No more moves
        #Write code here
    return game_finished, status
```

Function: Maximize

```
def maximize(matrix, active_turn, player, depth, alpha, beta, nodes_visited):
    game_finished,_ = game_over(matrix)
    if game_finished:
        return None, score(matrix, player, depth), nodes_visited
    depth += 1

    infinite_number = 100000
    maxUtility = -infinite_number
    choice = None

    childs = get_childs(matrix, active_turn)
    for child in childs:
        nodes_visited = nodes_visited + 1
        _, utility, nodes_visited = minimize(child, get_next_turn(active_turn), player, depth, alpha, beta,
        nodes_visited)

        if utility > maxUtility:
            choice = child
            # complete following code
            maxUtility = None

        if maxUtility >= beta:
            break
        if maxUtility > alpha:
            # complete following code
            alpha = None
    return choice, maxUtility, nodes_visited
```

Function: Minimize

```
def minimize(matrix, active_turn, player, depth, alpha, beta, nodes_visited):
    game_finished,_ = game_over(matrix)
    if game_finished:
        return None, score(matrix, player, depth), nodes_visited
    depth += 1
    infinite_number = 100000
    minUtility = infinite_number
    choice = None

    childs = get_childs(matrix, active_turn)
    for child in childs:
        nodes_visited = nodes_visited + 1
        _, utility, nodes_visited = maximize(child, get_next_turn(active_turn), player, depth, alpha, beta,
        nodes_visited)

        if utility < minUtility:
            choice = child
            # complete following code
            minUtility = None

        if minUtility <= alpha:
            break
        if minUtility < beta:
           # complete following code
            beta = None
    return choice, minUtility, nodes_visited
```

Function: Minmax

```
def minimax(matrix, player):

    #initialize infinite number here
    #initialize alpha
    #initialize beta
    # call maxminze function which should return choice, score, nodes visited

    return choice, score, nodes_visited
```

7.5 Exercises for lab

Exercises-1) The pseudocode for score function in given in the Fig. 7.9. You are required to implement it

Exercises -2) Complete implementation of minimize () function given in section 7.3

Exercises -3) Complete implementation of maximize () function given in section 7.3

Exercises -4) Implement of minmax () function given in section 7.3

Exercises -5) Test tic tac toa game for following outcome

a. Computer wins
b. Draw
c. Human wins

7.6 Home Work

1) Create jupyter Notebook file for the minmax algorithm given in a lab and provide description of the algorithm using markdown cells
2) Implement minmax algorithm for chess game (optional). Example in given in following https://github.com/devinalvaro/yachess