Lab # 8

Adversarial Search – Alpha Beta Pruning

7.1 Objective:

To Learn basics of adversarial search and to implement alpha beta pruning

7.2 Scope:

The student should know the following:

- To implement tic tac toe game, human vs computer
- To use minmax for embedded thinking
- To use alpha beta pruning for pruning of useless branches of the tree

7.3 Useful Concepts:

Alpha–beta (α–β) algorithm was discovered independently by a few researches in mid 1900s. Alpha–beta is actually an improved minimax using a heuristic. It stops evaluating a move when it makes sure that it's worse than previously examined move. Such moves need not to be evaluated further.

When added to a simple minimax algorithm, it gives the same output, but cuts off certain branches that can't possibly affect the final decision - dramatically improving the performance.

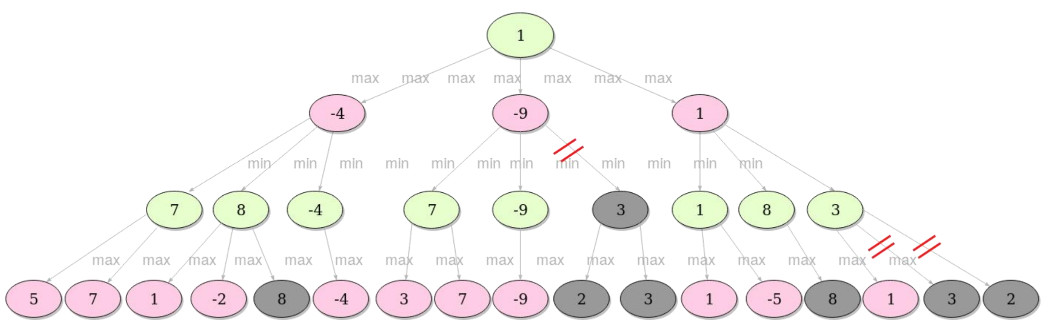
The main concept is to maintain two values through whole search:

Alpha: Best already explored option for player Max

Beta: Best already explored option for player Min

Initially, alpha is negative infinity and beta is positive infinity, i.e. in our code we'll be using the worst possible scores for both players.

Let's see how the previous tree will look if we apply alpha-beta method:



When the search comes to the first grey area (8), it'll check the current best (with minimum value) already explored option along the path for the minimizer, which is at that moment 7. Since 8 is bigger than 7, we are allowed to cut off all the further children of the node we're at (in this case there aren't any), since if we play that move, the opponent will play a move with value 8, which is worse for us than any possible move the opponent could have made if we had made another move.

A better example may be when it comes to a next grey. Note the nodes with value -9. At that point, the best (with maximum value) explored option along the path for the maximizer is -4. Since -9 is less than -4, we are able to cut off all the other children of the node we're at.

This method allows us to ignore many branches that lead to values that won't be of any help for our decision, nor they would affect it in any way.

```
/* Find the child state with the lowest utility value */

function MINIMIZE(state, α, β)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, ∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MAXIMIZE(child, α, β)

        if utility < minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

        if minUtility ≤ α:
            break

        if minUtility < β:
            β = minUtility

    return ⟨minChild, minUtility⟩
```

```
/* Find the child state with the highest utility value */

function MAXIMIZE(state, α, β)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, −∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MINIMIZE(child, α, β)

        if utility > maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

        if maxUtility ≥ β:
            break

        if maxUtility > α:
            α = maxUtility

    return ⟨maxChild, maxUtility⟩
```

```
/* Find the child state with the highest utility value */

function DECISION(state)
    returns STATE :

    ⟨child, _⟩ = MAXIMIZE(state, −∞, ∞)

    return child
```

Figure 1: Alpha beta pruning algorithm

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
```

```python
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()
```

```python
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

computer wins

o		x
x	x	
o	o	o

```
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

Exercise -1) implement the alpha beta pruning algorithm given in the Fig. 8.1.

Exercise -2) Complete implementation of minimize () function given in section 8.3

Exercise -3) Complete implementation of maximize () function given in section 8.3

Exercise -4) Implement of minmax () function given in section 8.3

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

d. Computer wins
e. Draw
f. Human wins

7.6 Home Work

1) Create jupyter Notebook file for the alpha beta purning algorithm given in a lab and provide description of the algorithm using markdown cells

2) Implement alpha beta pruning algorithm for chess game (optional). Example in given in following https://github.com/devinalvaro/yachess