Artificial Intelligence

Practical File

Name: Sachin Shukla

Roll No.: 2021UIN2379

1.Tic Tac Toe game

```
import os
import time
import random
```

```python
board = ["", " ", " ", " ", " ", " ", " ", " ", " ", " "]

def print_header():
    print(
    """

 _____  _  ____     _____  ____  ____     _____  ____  _____
/__ __\/ \/   _\   /__ __\/  _ \/   _\   /__ __\/  _ \/  __/
  / \  | ||  / _____ / \  | / \||  / _____ / \  | / \||  \
  | |  | ||  \_\____\| |  | |-|||  \_\____\| |  | \_/||  /_
  \_/  \_/\____/     \_/  \_/ \|\____/     \_/  \____/\____\




""")

def print_board():
    print("   |   |   ")
    print(" " + board[1] + " | " + board[2] + " | " + board[3] + "  ")
    print("   |   |   ")
    print("---|---|---")
    print("   |   |   ")
    print(" " + board[4] + " | " + board[5] + " | " + board[6] + "  ")
    print("   |   |   ")
    print("---|---|---")
    print("   |   |   ")
    print(" " + board[7] + " | " + board[8] + " | " + board[9] + "  ")
    print("   |   |   ")


def is_winner(board, player):
    if (board[1] == player and board[2] == player and board[3] == player) or \
            (board[4] == player and board[5] == player and board[6] == player) or \
            (board[7] == player and board[8] == player and board[9] == player) or \
            (board[1] == player and board[4] == player and board[7] == player) or \
            (board[2] == player and board[5] == player and board[8] == player) or \
            (board[3] == player and board[6] == player and board[9] == player) or \
            (board[1] == player and board[5] == player and board[9] == player) or \
            (board[3] == player and board[5] == player and board[7] == player):
        return True
    else:
        return False


def is_board_full(board):
    if " " in board:
        return False
    else:
        return True
```

```python
def get_computer_move(board, player):

    if board[5] == " ":
        return 5

    while True:
        move = random.randint(1, 9)
        if board[move] == " ":
            return move
            break

    return 5


while True:
    os.system("cls")
    print_header()
    print_board()

    choice = input("Please choose an empty space for X. ")
    choice = int(choice)

    if board[choice] == " ":
        board[choice] = "X"
    else:
        print("Sorry, that space is not empty!")
        time.sleep(1)

    if is_winner(board, "X"):
        os.system("cls")
        print_header()
        print_board()
        print("X wins! Congratulations")
        break

    os.system("cls")
    print_board()

    if is_board_full(board):
        print("Tie!")
        break

    choice = get_computer_move(board, "O")

    if board[choice] == " ":
        board[choice] = "O"
```

```
    else:
        print
        "Sorry, that space is not empty!"
        time.sleep(1)

    if is_winner(board, "O"):
        os.system("cls")
        print_header()
        print_board()
        print("O wins! Congratulations")
        break

    if is_board_full(board):
        print("Tie!")
        break
```

2. Tile Slide Puzzle

```
import copy

from heapq import heappush, heappop

n = 3

row = [ 1, 0, -1, 0 ]
col = [ 0, -1, 0, 1 ]

class priorityQueue:

    def __init__(self):
        self.heap = []

    def push(self, k):
        heappush(self.heap, k)

    def pop(self):
        return heappop(self.heap)

    def empty(self):
        if not self.heap:
            return True
        else:
            return False

class node:

    def __init__(self, parent, mat, empty_tile_pos,
                cost, level):
```

```
        self.parent = parent

        self.mat = mat

        self.empty_tile_pos = empty_tile_pos

        self.cost = cost

        self.level = level

    def __lt__(self, nxt):
        return self.cost < nxt.cost

def calculateCost(mat, final) -> int:

    count = 0
    for i in range(n):
        for j in range(n):
            if ((mat[i][j]) and
                (mat[i][j] != final[i][j])):
                count += 1

    return count

def newNode(mat, empty_tile_pos, new_empty_tile_pos,
            level, parent, final) -> node:

    new_mat = copy.deepcopy(mat)

    x1 = empty_tile_pos[0]
    y1 = empty_tile_pos[1]
    x2 = new_empty_tile_pos[0]
    y2 = new_empty_tile_pos[1]
    new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]

    cost = calculateCost(new_mat, final)

    new_node = node(parent, new_mat, new_empty_tile_pos,
                    cost, level)
    return new_node

def printMatrix(mat):

    for i in range(n):
        for j in range(n):
            print("%d " % (mat[i][j]), end = " ")
```

```
        print()

def isSafe(x, y):

    return x >= 0 and x < n and y >= 0 and y < n

def printPath(root):

    if root == None:
        return

    printPath(root.parent)
    printMatrix(root.mat)
    print()

def solve(initial, empty_tile_pos, final):

    pq = priorityQueue()

    cost = calculateCost(initial, final)
    root = node(None, initial,
                empty_tile_pos, cost, 0)

    pq.push(root)

    while not pq.empty():

        minimum = pq.pop()

        if minimum.cost == 0:

            printPath(minimum)
            return

        for i in range(4):
            new_tile_pos = [
                minimum.empty_tile_pos[0] + row[i],
                minimum.empty_tile_pos[1] + col[i], ]

            if isSafe(new_tile_pos[0], new_tile_pos[1]):

                child = newNode(minimum.mat,
                                minimum.empty_tile_pos,
                                new_tile_pos,
                                minimum.level + 1,
                                minimum, final,)

                pq.push(child)
```

```
initial = [ [ 1, 2, 3 ],
            [ 5, 6, 0 ],
            [ 7, 8, 4 ] ]


final = [ [ 1, 2, 3 ],
          [ 5, 8, 6 ],
          [ 0, 7, 4 ] ]

empty_tile_pos = [ 1, 2 ]

solve(initial, empty_tile_pos, final)
```

3. Water Jug problem

```
from collections import defaultdict

jug1, jug2, aim = 4, 3, 2

visited = defaultdict(lambda: False)

def waterJugSolver(amt1, amt2):

    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True

    if visited[(amt1, amt2)] == False:
        print(amt1, amt2)

        visited[(amt1, amt2)] = True

        return (waterJugSolver(0, amt2) or
                waterJugSolver(amt1, 0) or
                waterJugSolver(jug1, amt2) or
                waterJugSolver(amt1, jug2) or
                waterJugSolver(amt1 + min(amt2, (jug1-amt1)),
                amt2 - min(amt2, (jug1-amt1))) or
                waterJugSolver(amt1 - min(amt1, (jug2-amt2)),
                amt2 + min(amt1, (jug2-amt2))))

    else:
        return False

print("Steps: ")

waterJugSolver(0, 0)
```

4. Generate and test Rat in a maze

```
n = 4

def isValid(n, maze, x, y, res):

    if x >= 0 and y >= 0 and x < n and y < n and maze[x][y] == 1 and res[x][y] == 0:
        return True
    return False

def RatMaze(n, maze, move_x, move_y, x, y, res):

    if x == n-1 and y == n-1:
        return True
    for i in range(4):
        x_new = x + move_x[i]

        y_new = y + move_y[i]

        if isValid(n, maze, x_new, y_new, res):

            res[x_new][y_new] = 1
            if RatMaze(n, maze, move_x, move_y, x_new, y_new, res):
                return True
            res[x_new][y_new] = 0
    return False


def solveMaze(maze):

    res = [[0 for i in range(n)] for i in range(n)]
    res[0][0] = 1

    move_x = [-1, 1, 0, 0]

    move_y = [0, 0, -1, 1]

    if RatMaze(n, maze, move_x, move_y, 0, 0, res):
        for i in range(n):
            for j in range(n):
                print(res[i][j], end=' ')
            print()
    else:
        print('Solution does not exist')


if __name__ == "__main__":
```

```
    maze = [[1, 0, 0, 0],
            [1, 1, 0, 1],
            [0, 1, 0, 0],
            [1, 1, 1, 1]]

    solveMaze(maze)
```

5. Systematic generate and Test

```
n = 4

def isValid(n, maze, x, y, res):

    if x >= 0 and y >= 0 and x < n and y < n and maze[x][y] == 1 and res[x][y] == 0:
        return True
    return False

def RatMaze(n, maze, move_x, move_y, x, y, res):

    if x == n-1 and y == n-1:
        return True
    for i in range(4):
        x_new = x + move_x[i]

        y_new = y + move_y[i]

        if isValid(n, maze, x_new, y_new, res):

            res[x_new][y_new] = 1
            if RatMaze(n, maze, move_x, move_y, x_new, y_new, res):
                return True
            res[x_new][y_new] = 0
    return False


def solveMaze(maze):

    res = [[0 for i in range(n)] for i in range(n)]
    res[0][0] = 1

    move_x = [-1, 1, 0, 0]

    move_y = [0, 0, -1, 1]

    if RatMaze(n, maze, move_x, move_y, 0, 0, res):
        for i in range(n):
            for j in range(n):
                print(res[i][j], end=' ')
```

```
            print()
    else:
        print('Solution does not exist')


if __name__ == "__main__":
    maze = [[1, 0, 0, 0],
            [1, 1, 0, 1],
            [0, 1, 0, 0],
            [1, 1, 1, 1]]

    solveMaze(maze)
```

6. Simple hill climbing

```
import random

def randomSolution(tsp):
    cities = list(range(len(tsp)))
    solution = []

    for i in range(len(tsp)):
        randomCity = cities[random.randint(0, len(cities) - 1)]
        solution.append(randomCity)
        cities.remove(randomCity)

    return solution

def routeLength(tsp, solution):

    routeLength = 0
    for i in range(len(solution)):
        routeLength += tsp[solution[i - 1]][solution[i]]
    return routeLength

def getNeighbours(solution):
    neighbours = []
    for i in range(len(solution)):
        for j in range(i + 1, len(solution)):
            neighbour = solution.copy()
            neighbour[i] = solution[j]
            neighbour[j] = solution[i]
            neighbours.append(neighbour)
    return neighbours

def getBestNeighbour(tsp, neighbours):

    bestRouteLength = routeLength(tsp, neighbours[0])
```

```
    bestNeighbour = neighbours[0]
    for neighbour in neighbours:

        currentRouteLength = routeLength(tsp, neighbour)
        if currentRouteLength < bestRouteLength:

            bestRouteLength = currentRouteLength
            bestNeighbour = neighbour

    return bestNeighbour, bestRouteLength

def hillClimbing(tsp):

    currentSolution = randomSolution(tsp)
    currentRouteLength = routeLength(tsp, currentSolution)
    neighbours = getNeighbours(currentSolution)
    bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp,neighbours)
    while bestNeighbourRouteLength < currentRouteLength:

        currentSolution = bestNeighbour
        currentRouteLength = bestNeighbourRouteLength
        neighbours = getNeighbours(currentSolution)
        bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp,neighbours)
    return currentSolution, currentRouteLength

def main():
 tsp = [
 [0, 400, 500, 300],
 [400, 0, 300, 500],
 [500, 300, 0, 400],
 [300, 500, 400, 0]
 ]
 print(hillClimbing(tsp))

if __name__ == "__main__":
 main()
```

7. Steepest Ascent Hill climbing

```
import random

def randomSolution(tsp):
    cities = list(range(len(tsp)))
```

```
    solution = []
    for i in range(len(tsp)):
        randomCity = cities[random.randint(0, len(cities) - 1)]
        solution.append(randomCity)
        cities.remove(randomCity)
    return solution

def routeLength(tsp, solution):
    routeLength = 0
    for i in range(len(solution)):
        routeLength += tsp[solution[i - 1]][solution[i]]
    return routeLength

def getNeighbours(solution):
    neighbours = []
    for i in range(len(solution)):
        for j in range(i + 1, len(solution)):
            neighbour = solution.copy()
            neighbour[i] = solution[j]
            neighbour[j] = solution[i]
            neighbours.append(neighbour)
    return neighbours

def getBestNeighbour(tsp, neighbours):
    bestRouteLength = routeLength(tsp, neighbours[0])
    bestNeighbour = neighbours[0]
    for neighbour in neighbours:
        currentRouteLength = routeLength(tsp, neighbour)
        if currentRouteLength < bestRouteLength:
            bestRouteLength = currentRouteLength
            bestNeighbour = neighbour
    return bestNeighbour, bestRouteLength

def hillClimbing(tsp):
    currentSolution = randomSolution(tsp)
    currentRouteLength = routeLength(tsp, currentSolution)
    neighbours = getNeighbours(currentSolution)
```

8. best first search

```
from queue import PriorityQueue
v = 14
```

```python
graph = [[] for i in range(v)]

def best_first_search(actual_Src, target, n):
    visited = [False] * n
    pq = PriorityQueue()
    pq.put((0, actual_Src))
    visited[actual_Src] = True

    while pq.empty() == False:
        u = pq.get()[1]
        print(u, end=" ")
        if u == target:
            break

        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
    print()

def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))

addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)

source = 0
target = 9
best_first_search(source, target, v)
```

9 A* search

```python
from collections import deque

class Graph:
```

```python
    def __init__(self, adjac_lis):
        self.adjac_lis = adjac_lis

    def get_neighbors(self, v):
        return self.adjac_lis[v]

    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]

    def a_star_algorithm(self, start, stop):

        open_lst = set([start])
        closed_lst = set([])

        poo = {}
        poo[start] = 0

        par = {}
        par[start] = start

        while len(open_lst) > 0:
            n = None

            for v in open_lst:
                if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            if n == stop:
                reconst_path = []

                while par[n] != n:
                    reconst_path.append(n)
                    n = par[n]

                reconst_path.append(start)

                reconst_path.reverse()
```

```python
                print('Path found: {}'.format(reconst_path))
                return reconst_path

            for (m, weight) in self.get_neighbors(n):

                if m not in open_lst and m not in closed_lst:
                    open_lst.add(m)
                    par[m] = n
                    poo[m] = poo[n] + weight

                else:
                    if poo[m] > poo[n] + weight:
                        poo[m] = poo[n] + weight
                        par[m] = n

                        if m in closed_lst:
                            closed_lst.remove(m)
                            open_lst.add(m)

            open_lst.remove(n)
            closed_lst.add(n)

        print('Path does not exist!')
        return None
```

10. AO* search

```python
class Graph:
    def __init__(self, graph, heuristicNodeList, startNode):
        self.graph = graph
        self.H = heuristicNodeList
        self.start = startNode
        self.parent = {}
        self.status = {}
        self.solutionGraph = {}

    def applyAOStar(self):
        self.aoStar(self.start, False)

    def getNeighbors(self, v):
        return self.graph.get(v, '')

    def getStatus(self, v):
        return self.status.get(v, 0)

    def setStatus(self, v, val):
        self.status[v] = val
```

```python
    def getHeuristicNodeValue(self, n):
        return self.H.get(n, 0)

    def setHeuristicNodeValue(self, n, value):
        self.H[n] = value

    def printSolution(self):
        print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE:", self.start)
        print("------------------------------------------------------------")
        print(self.solutionGraph)
        print("------------------------------------------------------------")

    def computeMinimumCostChildNodes(self, v):
        minimumCost = 0
        costToChildNodeListDict = {}
        costToChildNodeListDict[minimumCost] = []
        flag = True
        for nodeInfoTupleList in self.getNeighbors(v):
            cost = 0
            nodeList = []
            for c, weight in nodeInfoTupleList:
                cost = cost + self.getHeuristicNodeValue(c) + weight
                nodeList.append(c)
            if flag == True:
                minimumCost = cost
                costToChildNodeListDict[minimumCost] = nodeList
                flag = False
            else:
                if minimumCost > cost:
                    minimumCost = cost
                    costToChildNodeListDict[minimumCost] = nodeList
        return minimumCost, costToChildNodeListDict[minimumCost]

    def aoStar(self, v, backTracking):
        print("HEURISTIC VALUES:", self.H)
        print("SOLUTION GRAPH:", self.solutionGraph)
        print("PROCESSING NODE:", v)
        print("------------------------------------------------------")

        if self.getStatus(v) >= 0:
            minimumCost, childNodeList = self.computeMinimumCostChildNodes(v)
            print(minimumCost, childNodeList)
            self.setHeuristicNodeValue(v, minimumCost)
            self.setStatus(v, len(childNodeList))
            solved = True
            for childNode in childNodeList:
                self.parent[childNode] = v
```

```
                if self.getStatus(childNode) != -1:
                    solved = solved & False
            if solved == True:
                self.setStatus(v, -1)
                self.solutionGraph[v] = childNodeList
            if v != self.start:
                self.aoStar(self.parent[v], True)
            if backTracking == False:
                for childNode in childNodeList:
                    self.setStatus(childNode, 0)
                self.aoStar(childNode, False)

print("Graph")
h1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}
graph1 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'C': [[('J', 1)]],
    'D': [[('E', 1), ('F', 1)]],
    'G': [[('I', 1)]]
}
G1 = Graph(graph1, h1, 'A')
G1.applyAOStar()
G1.printSolution()
```

11 NLP

```
import numpy as np
import pandas as pd
import re
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix
import nltk
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer

# Download NLTK data
nltk.download('stopwords')

# Read dataset
dataset = pd.read_csv('Restaurant_Reviews.tsv', delimiter='\t')

# Initialize empty array to append clean text
corpus = []

# Clean and preprocess the reviews
```

```
for i in range(0, 1000):
    review = re.sub('[^a-zA-Z]', ' ', dataset['Review'][i])
    review = review.lower()
    review = review.split()

    ps = PorterStemmer()

    review = [ps.stem(word) for word in review if not word in
set(stopwords.words('english'))]

    review = ' '.join(review)

    corpus.append(review)

# Create a CountVectorizer
cv = CountVectorizer(max_features=1500)

# X contains the corpus (dependent variable)
X = cv.fit_transform(corpus).toarray()

# y contains answers if the review is positive or negative
y = dataset.iloc[:, 1].values

# Create a RandomForestClassifier model
model = RandomForestClassifier(n_estimators=501, criterion='entropy')
model.fit(X_train, y_train)

# Evaluate the model
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

12. Tic tac toe using minimax with alpha beta pruning

```
import numpy as np
import sys
from copy import copy

rows = 3
cols = 3
board = np.zeros((rows, cols))
```

```python
inf = 9999999999
neg_inf = -9999999999

def printBoard():
    for i in range(0, rows):
        for j in range(0, cols):
            if board[i, j] == 0:
                sys.stdout.write(' _ ')
            elif board[i, j] == 1:
                sys.stdout.write(' X ')
            else:
                sys.stdout.write(' O ')
        print('')

heuristicTable = np.zeros((rows + 1, cols + 1))
numberOfWinningPositions = rows + cols + 2

for index in range(0, rows + 1):
    heuristicTable[index, 0] = 10 ** index
    heuristicTable[0, index] = -10 ** index

winningArray = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [0, 3, 6], [1, 4, 7], [2, 5,
8], [0, 4, 8], [2, 4, 6]])

def utilityOfState(state):
    stateCopy = copy(state.ravel())
    heuristic = 0

    for i in range(0, numberOfWinningPositions):
        maxp = 0
        minp = 0

        for j in range(0, rows):
            if stateCopy[winningArray[i, j]] == 2:
                maxp += 1
            elif stateCopy[winningArray[i, j]] == 1:
                minp += 1

        heuristic += heuristicTable[maxp][minp]

    return heuristic

def minimax(state, alpha, beta, maximizing, depth, maxp, minp):
    if depth == 0:
        return utilityOfState(state), state

    rowsLeft, columnsLeft = np.where(state == 0)
```

```
    returnState = copy(state)

    if rowsLeft.shape[0] == 0:
        return utilityOfState(state), returnState

    if maximizing:
        utility = neg_inf

        for i in range(0, rowsLeft.shape[0]):
            nextState = copy(state)
            nextState[rowsLeft[i], columnsLeft[i]] = maxp
            Nutility, Nstate = minimax(nextState, alpha, beta, False, depth - 1, maxp,
minp)

            if Nutility > utility:
                utility = Nutility
                returnState = copy(nextState

            if utility > alpha:
                alpha = utility

            if alpha >= beta:
                break

        return utility, returnState

    else:
        utility = inf

        for i in range(0, rowsLeft.shape[0]):
            nextState = copy(state)
            nextState[rowsLeft[i], columnsLeft[i]] = minp
            Nutility, Nstate = minimax(nextState, alpha, beta, True, depth - 1, maxp,
minp)

            if Nutility < utility:
                utility = Nutility
                returnState = copy(nextState)

            if utility < beta:
                beta = utility

            if alpha >= beta:
                break

        return utility, returnState

def checkGameOver(state):
```

```python
    stateCopy = copy(state)
    value = utilityOfState(stateCopy)

    if value >= 1000:
        return 1

    return -1

def main():
    num = int(input('enter player num (1st or 2nd) '))
    value = 0
    global board

    for turn in range(0, rows * cols):
        if (turn + num) % 2 == 1:
            r = int(input("Enter the row: "))
            c = int(input("Enter the column: "))

            board[r - 1, c - 1] = 1
            printBoard()
            value = checkGameOver(board)

            if value == 1:
                print('U win. Game Over')
                sys.exit()

        else:
            state = copy(board)
            value, nextState = minimax(state, neg_inf, inf, True, 2, 2, 1)
            board = copy(nextState)
            printBoard()
            value = checkGameOver(board)

            if value == 1:
                print('PC wins. Game Over')
                sys.exit()

    print('It\'s a draw')

main()
```