NOIDA INSTITUTE OF ENGINEERING AND TECHNOLOGY
GREATER NOIDA-201306
(An Autonomous Institute)
School of Computer Sciences & Engineering in Emerging Technologies

Department of CSE (Data Science)

Session (2021 – 2022)

LAB FILE

ON

Artificial Intelligence

(ACSAI0451)

(4th Semester)

Submitted To:

Dr. Vikas Sagar

Submitted By:

Name: Amritanshu Sharma
Roll No: 2001331540025

Affiliated to Dr. A.P.J Abdul Kalam Technical University, Uttar Pradesh, Lucknow.

Introduction to Artificial Intelligence Lab (ACSAI0451)

INDEX

S. No.	PRACTICAL CONDUCTED	DATE	PAGE NO	SIGNATURE
1	Write a Program to perform Simple Calculator			
2	Write a Program to implement an AI chatbot			
3	Write a Program to perform Water Jug Problem			
4	Write a Program to Implement Alpha Beta Pruning graphically			
5	Use Heuristic Search Techniques to Implement Best first search and A* algorithm			
6	Write Program to implement Hill-Climbing Algorithm Using Heuristic Search Techniques			
7	Write a program to perform the N Queen problem			
8	Write a program to perform the TIC TAC TOE Problem			
9	Write a Program to perform Breadth first search			
10	Write a Program to perform Depth first search			
11	Write a python program to POS (Parts of Speech) tagging for the give sentence using NLTK			
12	Write a program to implement Naïve Bayes Algorithm			

4.Write a Program to implement alpha-beta Pruning.

Code

```
# Python3 program to demonstrate
# working of Alpha-Beta Pruning

# Initial values of Alpha and Beta
MAX, MIN = 1000, -1000

# Returns optimal value for current player
#(Initially called for root and maximizer)
def minimax(depth, nodeIndex, maximizingPlayer,
            values, alpha, beta):

    # Terminating condition. i.e
    # leaf node is reached
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:

        best = MIN

        # Recur for left and right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                          False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

    else:
        best = MAX

        # Recur for left and
        # right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                            True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)
```

```
            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best




values = [3, 5, 6, 9, 1, 2, 0, -1]
print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

Output

```
The optimal value is : 5
```

5. Use Heuristic Search Techniques to Implement Best first search (Best-Solution but not always optimal) and A* algorithm (Always gives optimal solution).

Code – Best Frist Search

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]

# Function For Implementing Best First Search
# Gives output path having lowest cost


def best_first_search(source, target, n):
    visited = [False] * n
    #visited = True
    pq = PriorityQueue()
    pq.put((0, source))
    while pq.empty() == False:
        u = pq.get()[1]
        # Displaying the path having lowest cost
        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()

# Function for adding edges to graph


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


# The nodes shown in above example(by alphabets) are
# implemented using integers addedge(x,y,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)
```

output

```
0 1 0 3 2 8 9
```

Code- A* search algorithm

```
def aStarAlgo(start_node, stop_node):
    open_set= set(start_node)
    closed_set=set()
    g={} # stores the distance from starting node
    parents={} # parents contains an adjacency map of all nodes

    # distance of starting node from itself is zero
    g[start_node] = 0
    #start_node is root node i.e it has no parent nodes
    # so start_nodoe is set to its own parent node
    parents[start_node] = start_node

    while len(open_set) > 0:
        n = None

        #node with lowest f() is found
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
```

```
    if n == stop_node or Graph_nodes[n] == None:
        pass
    else:
        for (m, weight) in get_neighbors(n):
            #nodes 'm' not in first and last set are added to first
            #n is set its parent
            if m not in open_set and m not in closed_set:
                open_set.add(m)
                parents[m] = n
                g[m] = g[n] + weight



            #for each node m,compare its distance from start i.e g(m) to the
            #from start through n node
            else:
                if g[m] > g[n] + weight:
                    #update g(m)
                    g[m] = g[n] + weight
                    #change parent of m to n
                    parents[m] = n

                    #if m in closed set,remove and add to open
                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)

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

```
        # if the current node is the stop_node
        # then we begin reconstructin the path from it to the start_node
        if n == stop_node:
            path = []

            while parents[n] != n:
                path.append(n)
                n = parents[n]

            path.append(start_node)

            path.reverse()

            print('Path found: {}'.format(path))
            return path


        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_set.remove(n)
        closed_set.add(n)

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

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
```

```
    else:
        return None
#for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0,

    }

    return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
}
aStarAlgo('A', 'G')
```

output

```
         Path found: ['A', 'E', 'D', 'G']
Out[14]: ['A', 'E', 'D', 'G']
```

6. Use Heuristic Search Techniques to Implement Hill-Climbing Algorithm.

Code

```
import random

#code an instantiation of the Travelling salesman problem

#For Hill climbing to work, it has to start with a random solution to our Travelling salesman problem



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

# Create a function calculating the length of a route



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

#Create a function generating all neighbours of a solution
```

```
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

#Create a function finding the best neighbour

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

# the Hill climbing algorithm

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

```python
    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()
```

output

```
([0, 1, 2, 3], 1400)
```

7. Write a program to perform the N Queen problem.

```
#n queen problem
# Function to check if two queens threaten each other or not
def isSafe(mat, r, c):
     # return false if two queens share the same column
     for i in range(r):
         if mat[i][c] == 'Q':
             return False
     # return false if two queens share the same `\` diagonal
     (i, j) = (r, c)
     while i >= 0 and j >= 0:
         if mat[i][j] == 'Q':
             return False
         i = i - 1
         j = j - 1
    # return false if two queens share the same `/` diagonal
    (i, j) = (r, c)
    while i >= 0 and j < len(mat):
         if mat[i][j] == 'Q':
             return False
         i = i - 1
         j = j + 1
    return True
def printSolution(mat):
    for r in mat:
        print(str(r).replace(',', '').replace('\'', ''))
     print()
def nQueen(mat, r):
     # if `N` queens are placed successfully, print the solution
     if r == len(mat):
         printSolution(mat)
         return
```

```
        # place queen at every square in the current row `r`
        # and recur for each valid movement
        for i in range(len(mat)):
            # if no two queens threaten each other
            if isSafe(mat, r, i):
                # place queen on the current square
                mat[r][i] = 'Q'
                # recur for the next row
                nQueen(mat, r + 1)
                # backtrack and remove the queen from the current square
                mat[r][i] = '–'
if __name__ == '__main__':
    # `N × N` chessboard
    N = 4
    # `mat[][]` keeps track of the position of queens in
    # the current configuration
    mat = [['–' for x in range(N)] for y in range(N)]
    nQueen(mat, 0)
```

OUTPUT:

```
[– Q – –]
[– – – Q]
[Q – – –]
[– – Q –]

[– – Q –]
[Q – – –]
[– – – Q]
[– Q – –]
```

8. Implement Tic-Tac-Toe.

code

```
# Tic-Tac-Toe Program using
# random number in Python

# importing all necessary libraries
import numpy as np
import random
from time import sleep

# Creates an empty board
def create_board():
    return(np.array([[0, 0, 0],
                    [0, 0, 0],
                    [0, 0, 0]]))

# Check for empty places on board
def possibilities(board):
    l = []

    for i in range(len(board)):
        for j in range(len(board)):

            if board[i][j] == 0:
                l.append((i, j))
    return(l)

# Select a random place for the player
def random_place(board, player):
```

```
    selection = possibilities(board)
    current_loc = random.choice(selection)
    board[current_loc] = player
    return(board)

# Checks whether the player has three
# of their marks in a horizontal row
def row_win(board, player):
    for x in range(len(board)):
        win = True

        for y in range(len(board)):
            if board[x, y] != player:
                win = False
                continue

        if win == True:
            return(win)
    return(win)

# Checks whether the player has three
# of their marks in a vertical row
def col_win(board, player):
    for x in range(len(board)):
        win = True

        for y in range(len(board)):
            if board[y][x] != player:
                win = False
                continue
```

```
        if win == True:
            return(win)
    return(win)

# Checks whether the player has three
# of their marks in a diagonal row
def diag_win(board, player):
    win = True
    y = 0
    for x in range(len(board)):
        if board[x, x] != player:
            win = False
    if win:
        return win
    win = True
    if win:
        for x in range(len(board)):
            y = len(board) - 1 - x
            if board[x, y] != player:
                win = False
    return win

# Evaluates whether there is
# a winner or a tie
def evaluate(board):
    winner = 0

    for player in [1, 2]:
        if (row_win(board, player) or
            col_win(board,player) or
            diag_win(board,player)):
```

```
            winner = player

    if np.all(board != 0) and winner == 0:
        winner = -1
    return winner

# Main function to start the game
def play_game():
    board, winner, counter = create_board(), 0, 1
    print(board)
    sleep(2)

    while winner == 0:
        for player in [1, 2]:
            board = random_place(board, player)
            print("Board after " + str(counter) + " move")
            print(board)
            sleep(2)
            counter += 1
            winner = evaluate(board)
            if winner != 0:
                break
    return(winner)

# Driver Code
print("Winner is: " + str(play_game()))
```

output

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
Board after 1 move
[[0 0 0]
 [0 1 0]
 [0 0 0]]
Board after 2 move
[[0 0 0]
 [0 1 0]
 [2 0 0]]
Board after 3 move
[[0 0 0]
 [0 1 1]
 [2 0 0]]
Board after 4 move
[[0 0 0]
 [2 1 1]
 [2 0 0]]
Board after 5 move
[[0 0 1]
 [2 1 1]
 [2 0 0]]
Board after 6 move
[[2 0 1]
 [2 1 1]
 [2 0 0]]
Winner is: 2
```

9. Write a python Program to Implement BFS.

Code

```
# Python3 Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s.
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (max(self.graph) + 1)

        # Create a queue for BFS
        queue = []
```

```
        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

            # Dequeue a vertex from
            # queue and print it
            s = queue.pop(0)
            print (s, end = " ")

            # Get all adjacent vertices of the
            # dequeued vertex s. If a adjacent
            # has not been visited, then mark it
            # visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True

# Driver code

# Create a graph given in
# the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
```

```
g.addEdge(3, 1)
g.addEdge(3, 2)

print ("Following is Breadth First Traversal"
       " (starting from vertex 0)")
g.BFS(0)
```

output

```
Following is Breadth First Traversal (starting from vertex 0)
0 1 2 3
```

10. Write a python Program to Implement BFS.

Code

```
# Python3 program to print DFS traversal
# from a given given graph
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation



class Graph:

	# Constructor
	def __init__(self):

		# default dictionary to store graph
		self.graph = defaultdict(list)

	# function to add an edge to graph
	def addEdge(self, u, v):
		self.graph[u].append(v)

	# A function used by DFS
	def DFSUtil(self, v, visited):

		# Mark the current node as visited
		# and print it
		visited.add(v)
		print(v, end=' ')
```

```
        # Recur for all the vertices
        # adjacent to this vertex
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)

    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self, v):

        # Create a set to store visited vertices
        visited = set()

        # Call the recursive helper function
        # to print DFS traversal
        self.DFSUtil(v, visited)

# Driver code


# Create a graph given
# in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print("Following is DFS from (starting from vertex 2)")
```

```
g.DFS(2)
```

output

```
Following is DFS from (starting from vertex 2)
2 0 1 3
```

11. Write a python program to POS (Parts of Speech) tagging for the give sentence using NLTK

Code

```
import nltk

nltk.download('stopwords')

nltk.download('punkt')

nltk.download('averaged_perceptron_tagger')

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word_tokenize, sent_tokenize

stop_words = set(stopwords.words('english'))


# Dummy text

txt = "Sukanya, Rajib and Naba are my good friends. " \

    "Sukanya is getting married next year. " \

    "Marriage is a big step in one’s life." \

    "It is both exciting and frightening. " \

    "But friendship is a sacred bond between people." \

    "It is a special kind of love between us. " \

    "Many of you must have tried searching for a friend "\

    "but never found the right one."


# sent_tokenize is one of instances of

# PunktSentenceTokenizer from the nltk.tokenize.punkt module


tokenized = sent_tokenize(txt)

for i in tokenized:


    # Word tokenizers is used to find the words

    # and punctuation in a string

    wordsList = nltk.word_tokenize(i)
```

```
# removing stop words from wordList
wordsList = [w for w in wordsList if not w in stop_words]


# Using a Tagger. Which is part-of-speech
# tagger or POS-tagger.
tagged = nltk.pos_tag(wordsList)


print(tagged)
```

Output

```
[('Sukanya', 'NNP'), (',', ','), ('Rajib', 'NNP'), ('Naba', 'NNP'), ('good', 'JJ'), ('friends', 'NNS'), ('.', '.')]
[('Sukanya', 'NNP'), ('getting', 'VBG'), ('married', 'VBN'), ('next', 'JJ'), ('year', 'NN'), ('.', '.')]
[('Marriage', 'NN'), ('big', 'JJ'), ('step', 'NN'), ('one', 'CD'), ('’', 'NN'), ('life.It', 'NN'), ('exciting', 'VBG'), ('frigh
tening', 'NN'), ('.', '.')]
[('But', 'CC'), ('friendship', 'NN'), ('sacred', 'VBD'), ('bond', 'NN'), ('people.It', 'NN'), ('special', 'JJ'), ('kind', 'N
N'), ('love', 'VB'), ('us', 'PRP'), ('.', '.')]
[('Many', 'JJ'), ('must', 'MD'), ('tried', 'VB'), ('searching', 'VBG'), ('friend', 'NN'), ('never', 'RB'), ('found', 'VBD'),
('right', 'JJ'), ('one', 'CD'), ('.', '.')]
```

12. Write a program to implement Naïve Bayes Algorithm

<u>Code</u>

```
# Importing library
import math
import random
import csv



# the categorical class names are changed to numberic data
# eg: yes and no encoded to 1 and 0
def encode_class(mydata):
	classes = []
	for i in range(len(mydata)):
		if mydata[i][-1] not in classes:
			classes.append(mydata[i][-1])
	for i in range(len(classes)):
		for j in range(len(mydata)):
			if mydata[j][-1] == classes[i]:
				mydata[j][-1] = i
	return mydata



# Splitting the data
def splitting(mydata, ratio):
	train_num = int(len(mydata) * ratio)
	train = []
	# initially testset will have all the dataset
	test = list(mydata)
	while len(train) < train_num:
```

```
            # index generated randomly from range 0
            # to length of testset
            index = random.randrange(len(test))
            # from testset, pop data rows and put it in train
            train.append(test.pop(index))
        return train, test



# Group the data rows under each class yes or
# no in dictionary eg: dict[yes] and dict[no]
def groupUnderClass(mydata):
        dict = {}
        for i in range(len(mydata)):
            if (mydata[i][-1] not in dict):
                dict[mydata[i][-1]] = []
            dict[mydata[i][-1]].append(mydata[i])
        return dict



# Calculating Mean
def mean(numbers):
        return sum(numbers) / float(len(numbers))

# Calculating Standard Deviation
def std_dev(numbers):
        avg = mean(numbers)
        variance = sum([pow(x - avg, 2) for x in numbers]) / float(len(numbers) - 1)
        return math.sqrt(variance)

def MeanAndStdDev(mydata):
        info = [(mean(attribute), std_dev(attribute)) for attribute in zip(*mydata)]
```

```
        # eg: list = [ [a, b, c], [m, n, o], [x, y, z]]
        # here mean of 1st attribute =(a + m+x), mean of 2nd attribute = (b + n+y)/3
        # delete summaries of last class
        del info[-1]
        return info

# find Mean and Standard Deviation under each class
def MeanAndStdDevForClass(mydata):
        info = {}
        dict = groupUnderClass(mydata)
        for classValue, instances in dict.items():
                info[classValue] = MeanAndStdDev(instances)
        return info



# Calculate Gaussian Probability Density Function
def calculateGaussianProbability(x, mean, stdev):
        expo = math.exp(-(math.pow(x - mean, 2) / (2 * math.pow(stdev, 2))))
        return (1 / (math.sqrt(2 * math.pi) * stdev)) * expo



# Calculate Class Probabilities
def calculateClassProbabilities(info, test):
        probabilities = {}
        for classValue, classSummaries in info.items():
                probabilities[classValue] = 1
                for i in range(len(classSummaries)):
                        mean, std_dev = classSummaries[i]
                        x = test[i]
                        probabilities[classValue] *= calculateGaussianProbability(x, mean, std_dev)
        return probabilities
```

```
# Make prediction - highest probability is the prediction
def predict(info, test):
	probabilities = calculateClassProbabilities(info, test)
	bestLabel, bestProb = None, -1
	for classValue, probability in probabilities.items():
		if bestLabel is None or probability > bestProb:
			bestProb = probability
			bestLabel = classValue
	return bestLabel


# returns predictions for a set of examples
def getPredictions(info, test):
	predictions = []
	for i in range(len(test)):
		result = predict(info, test[i])
		predictions.append(result)
	return predictions

# Accuracy score
def accuracy_rate(test, predictions):
	correct = 0
	for i in range(len(test)):
		if test[i][-1] == predictions[i]:
			correct += 1
	return (correct / float(len(test))) * 100.0


# driver code
```

```
# add the data path in your system
filename = r'E:\user\MACHINE LEARNING\machine learning algos\Naive bayes\filedata.csv'


# load the file and store it in mydata list
mydata = csv.reader(open(filename, "rt"))
mydata = list(mydata)
mydata = encode_class(mydata)
for i in range(len(mydata)):
        mydata[i] = [float(x) for x in mydata[i]]


# split ratio = 0.7
# 70% of data is training data and 30% is test data used for testing
ratio = 0.7
train_data, test_data = splitting(mydata, ratio)
print('Total number of examples are: ', len(mydata))
print('Out of these, training examples are: ', len(train_data))
print("Test examples are: ", len(test_data))

# prepare model
info = MeanAndStdDevForClass(train_data)

# test model
predictions = getPredictions(info, test_data)
accuracy = accuracy_rate(test_data, predictions)
print("Accuracy of your model is: ", accuracy)
```

output

```
Total number of examples are: 200
Out of these, training examples are: 140
Test examples are: 60
Accuracy of your model is: 71.2376788
```