Practical - 1

AIM: Write a program to create a Chat Bot

```
import nltk
from nltk.chat.util import Chat,reflections
reflections={
    "i am" : "you are",
    "i was" : "you were",
    "i" : "you",
    "i'm" : "you are",
    "i'd" : "you would",
    "i've" : "you have",
    "i'll" : "you will",
    "my" : "your",
    "You are" : "I am",
    "You were" : "I was",
    "You've" : "I have",
    "You'll" : "I will",
    "Your" : "my",
    "Yours" : "mine",
    "You" : "me",
    "me" : "you"
}
pairs=[
       [
        r"my name is (.*)",
        ["Hello %1, How are you today?",]

       ],
       [
        r"hii|hey|hello",
        ["Holla","Hey There",]
       ],
       [
        r"What is ypur name ?",
        ["I am a bot createrd by Kunal, You can call me anything you like|",]
       ],
       [
        r"how are you ?",
        ["I'm doing good and how about you ?",]
       ],
       [
        r"Sorry (.*)",
        ["It's alright","It's Okay","Never mind",]
```

```
        ],
        [
         r"I am fine",
         ["Great to hear that, HOw can i help you?",]
        ],
        [
         r"i'm (.*) doing good",
         ["Nice to hear that","How can i help you?",]
        ],
        [
         r"(.*) age?",
         ["I'm a computer program dude and seriously you are asking me this?",]
        ],
        [
         r"who (.*) sportsperson ?",
         ["MEssy","Ronaldo"]
        ],
        [
         r"who (.*) (moviestar|actor) ?",
         ["Brad Pitt"]
        ],
        [
         r"Fav Sports?",
         ["Cricket"]
        ],
        [
         r"quit|bye",
         ["Bye take care, it was nice talimg to you, so long..."]
        ],
]
def chat():
  print("I am a chatbot created by Kunal for your service")
  chat=Chat(pairs,reflections)
  chat.converse()
#initiates the conversation
if__name__=="__main__":
  chat()
```

Output:

```
I am a chatbot created by Kunal for your service
>
>i am noah
None
>my name is noah
Hello noah, How are you today?
>i am fine
Great to hear that, HOw can i help you?
> bye
Bye take care, it was nice talimg to you, so long...
```

Practical - 2

AIM: Write a program to create a Tic-Tac-Toe game

```
import numpy as np
def create_board():
    return (np.array([[0, 0, 0],
                      [0, 0, 0],
                      [0, 0, 0]]))
def coordinates(board, player):
    i, j, cn = (-1, -1, 0)
    while (i > 3 or i < 0 or j < 0 or j > 3) or (board[i][j] != 0):
        if cn > 0:
            print("Wrong Input! Try Again")
        print("Player {}'s turn".format(player))
        i = int(input("x-coordinates: "))
        j = int(input("y-coordinates: "))
        i = i - 1
        j = j - 1
        cn = cn + 1
    board[i][j] = player
    return board
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
    return win
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
    return win
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
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
def play_game():
    board, winner, counter = create_board(), 0, 1
    print(board)
    while winner == 0:
        for player in [1, 2]:
            board = coordinates(board, player)
            print("Board after " + str(counter) + " move")
            print(board)
            counter += 1
            winner = evaluate(board)
            if winner != 0:
                break
    return winner
print("Winner is: " + str(play_game()))
```

Output:

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]

Player 1's turn
x-coordinates: 6
y-coordinates:  1
Wrong Input! Try Again

Player 1's turn
x-coordinates: 2
y-coordinates: 2
Board after 1 move
```

```
[[0 0 0]
 [0 1 0]
 [0 0 0]]

Player 2's turn
x-coordinates: 3
y-coordinates: 2
Board after 2 move
[[0 0 0]
 [0 1 0]
 [0 2 0]]

Player 1's turn
x-coordinates: 1
y-coordinates: 1
Board after 3 move
[[1 0 0]
 [0 1 0]
 [0 2 0]]

Player 2's turn
x-coordinates: 2
y-coordinates: 1
Board after 4 move
[[1 0 0]
 [2 1 0]
 [0 2 0]]

Player 1's turn
x-coordinates: 3
y-coordinates: 3
Board after 5 move
[[1 0 0]
 [2 1 0]
 [0 2 1]]

Winner is: 1
```

Practical - 6

AIM: Write a python program to implement Simple Calculator program

```
# Program make a simple calculator that can add, subtract, multiply and divide u
sing #functions


# define functions
def add(x, y):
   """This function adds two numbers"""
   return x + y
def subtract(x, y):
   """This function subtracts two numbers"""
   return x - y
def multiply(x, y):
   """This function multiplies two numbers"""
   return x * y
def divide(x, y):
   """This function divides two numbers"""
   return x / y
# take input from the user
print("Select operation.")
print("1.Add")
print("2.Subtract")
print("3.Multiply")
print("4.Divide")
choice = input("Enter choice(1/2/3/4):")
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
if choice == '1':
```

```
    print(num1,"+",num2,"=", add(num1,num2))
elif choice == '2':
    print(num1,"-",num2,"=", subtract(num1,num2))
elif choice == '3':
    print(num1,"*",num2,"=", multiply(num1,num2))
elif choice == '4':
    print(num1,"/",num2,"=", divide(num1,num2))
else:
    print("Invalid input")
```

Output:

```
Select operation.
1.Add
2.Subtract
3.Multiply
4.Divide
Enter choice(1/2/3/4):3
Enter first number: 23
Enter second number: 56
23 * 56 = 128
```

Practical-5

AIM: Write program to implement/solve a Water Jug Problem

```
#Water_Jug_Problem
def pour(jug1, jug2):
    max1, max2, fill = 5, 7, 4  #Change maximum capacity and final capacity
    print(" %d\t %d" % (jug1, jug2))
    if jug2 is fill:
        return
    elif jug2 is max2:
        pour(0, jug1)
    elif jug1 != 0 and jug2 is 0:
        pour(0, jug1)
    elif jug1 is fill:
        pour(jug1, 0)
    elif jug1 < max1:
        pour(max1, jug2)
    elif jug1 < (max2-jug2):
        pour(0, (jug1+jug2))
    else:
        pour(jug1-(max2-jug2), (max2-jug2)+jug2)

print("JUG1\tJUG2")
pour(5, 7)
```

Output:

```
JUG1  JUG2
 5      7
 0      5
 5      5
 3      7
 0      3
 5      3
 1      7
 0      1
 5      1
 0      6
 5      6
 4      7
 0      4
```

Practical-3

AIM: Write a program for implementation of Breadth First Search

```
graph ={
    '5': ['3','7'],
    '3': ['2','4'],
    '7': ["8"],
    '2': [],
    '4': ['8'],
    '8': []
}

visited =[]
queue=[]

def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)

  while queue:
    m= queue.pop(0)
    print(m,end= " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

print(" Follwing is the Breadth_first search: ")
bfs(visited, graph,"5")
```

Output:

```
Follwing is the Breadth_first search:
5 3 7 2 4 8
```

Practical-4

AIM: Write a program for implementation of Depth First Search

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Output:

```
Following is the Depth-First Search
5
3
2
4
8
7
```

Practical - 9

AIM: Write program to implement a 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

# Driver Code
if__name__== "__main__":

    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
```

Parctical-10

AIM: Use Heuristic Search Techniques to Implement Best first search.

```
graph={
    "A":[["B",3],["C",2]],
    "B":[["D",2],["E",3]],
    "C":[],
    "D":[],
    "E":[]

}

l=[]
def BTS(graph ,l,node):
    q=[]
    print(node,end=" ")
    l.append(node)
    for i in graph[node]:
        q.append(tuple(i))
    q=sorted(q,key=lambda x:x[1])
    while q:
        s=q.pop(0)
        print(s[0],end=" ")
        for i in graph[s[0]]:
            if i[0] not in l:
                l.append(i[0])
                q.append(i)
                q=sorted(q,key=lambda x:x[1])
BTS(graph,l,"A")
```

Output:

```
A C B D E
```

Parctical-7

AIM: Write a program to solve N-Queens 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 – –]

Parctical 8

AIM: Solve 8-puzzle problem using best first search

```
class Solution:
   def solve(self, board):
      dict = {}
      flatten = []
      for i in range(len(board)):
         flatten += board[i]
      flatten = tuple(flatten)

      dict[flatten] = 0

      if flatten == (0, 1, 2, 3, 4, 5, 6, 7, 8):
         return 0

      return self.get_paths(dict)

   def get_paths(self, dict):
      cnt = 0
      while True:
         current_nodes = [x for x in dict if dict[x] == cnt]
         if len(current_nodes) == 0:
            return -1

         for node in current_nodes:
            next_moves = self.find_next(node)
            for move in next_moves:
```

```
            if move not in dict:
                dict[move] = cnt + 1
            if move == (0, 1, 2, 3, 4, 5, 6, 7, 8):
                return cnt + 1
        cnt += 1

def find_next(self, node):
    moves = {
        0: [1, 3],
        1: [0, 2, 4],
        2: [1, 5],
        3: [0, 4, 6],
        4: [1, 3, 5, 7],
        5: [2, 4, 8],
        6: [3, 7],
        7: [4, 6, 8],
        8: [5, 7],
    }

    results = []
    pos_0  =  node.index(0)
    for move in moves[pos_0]:
        new_node = list(node)
        new_node[move], new_node[pos_0] = new_node[pos_0], new_node[move]
        results.append(tuple(new_node))

    return results
```

```
ob = Solution()
matrix = [
   [3, 1, 2],
   [4, 7, 5],
   [6, 8, 0]
]
print(ob.solve(matrix))
```

OUTPUT:

4

Parctical-10

AIM: Use Heuristic Search Techniques to Implement Hill-Climbing Algorithm.

```
import math
from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp
def create_data_model():
    """Stores the data for the problem."""
    data = {}
    # Locations in block units
    data['locations'] = [
        (288, 149), (288, 129), (270, 133), (256, 141), (256, 157), (246, 157),
        (236, 169), (228, 169), (228, 161), (220, 169), (212, 169), (204, 169),
        (196, 169), (188, 169), (196, 161), (188, 145), (172, 145), (164, 145),
        (156, 145), (148, 145), (140, 145), (148, 169), (164, 169), (172, 169),
        (156, 169), (140, 169), (132, 169), (124, 169), (116, 161), (104, 153),
        (104, 161), (104, 169), (90, 165), (80, 157), (64, 157), (64, 165),
        (56, 169), (56, 161), (56, 153), (56, 145), (56, 137), (56, 129),
        (56, 121), (40, 121), (40, 129), (40, 137), (40, 145), (40, 153),
        (40, 161), (40, 169), (32, 169), (32, 161), (32, 153), (32, 145),
        (32, 137), (32, 129), (32, 121), (32, 113), (40, 113), (56, 113),
        (56, 105), (48, 99), (40, 99), (32, 97), (32, 89), (24, 89),
        (16, 97), (16, 109), (8, 109), (8, 97), (8, 89), (8, 81),
        (8, 73), (8, 65), (8, 57), (16, 57), (8, 49), (8, 41),
        (24, 45), (32, 41), (32, 49), (32, 57), (32, 65), (32, 73),
        (32, 81), (40, 83), (40, 73), (40, 63), (40, 51), (44, 43),
        (44, 35), (44, 27), (32, 25), (24, 25), (16, 25), (16, 17),
        (24, 17), (32, 17), (44, 11), (56, 9), (56, 17), (56, 25),
        (56, 33), (56, 41), (64, 41), (72, 41), (72, 49), (56, 49),
        (48, 51), (56, 57), (56, 65), (48, 63), (48, 73), (56, 73),
        (56, 81), (48, 83), (56, 89), (56, 97), (104, 97), (104, 105),
        (104, 113), (104, 121), (104, 129), (104, 137), (104, 145), (116, 145),
        (124, 145), (132, 145), (132, 137), (140, 137), (148, 137), (156, 137),
        (164, 137), (172, 125), (172, 117), (172, 109), (172, 101), (172, 93),
        (172, 85), (180, 85), (180, 77), (180, 69), (180, 61), (180, 53),
        (172, 53), (172, 61), (172, 69), (172, 77), (164, 81), (148, 85),
        (124, 85), (124, 93), (124, 109), (124, 125), (124, 117), (124, 101),
        (104, 89), (104, 81), (104, 73), (104, 65), (104, 49), (104, 41),
        (104, 33), (104, 25), (104, 17), (92, 9), (80, 9), (72, 9),
        (64, 21), (72, 25), (80, 25), (80, 25), (80, 41), (88, 49),
        (104, 57), (124, 69), (124, 77), (132, 81), (140, 65), (132, 61),
        (124, 61), (124, 53), (124, 45), (124, 37), (124, 29), (132, 21),
```

```
        (124, 21), (120, 9), (128, 9), (136, 9), (148, 9), (162, 9),
        (156, 25), (172, 21), (180, 21), (180, 29), (172, 29), (172, 37),
        (172, 45), (180, 45), (180, 37), (188, 41), (196, 49), (204, 57),
        (212, 65), (220, 73), (228, 69), (228, 77), (236, 77), (236, 69),
        (236, 61), (228, 61), (228, 53), (236, 53), (236, 45), (228, 45),
        (228, 37), (236, 37), (236, 29), (228, 29), (228, 21), (236, 21),
        (252, 21), (260, 29), (260, 37), (260, 45), (260, 53), (260, 61),
        (260, 69), (260, 77), (276, 77), (276, 69), (276, 61), (276, 53),
        (284, 53), (284, 61), (284, 69), (284, 77), (284, 85), (284, 93),
        (284, 101), (288, 109), (280, 109), (276, 101), (276, 93), (276, 85),
        (268, 97), (260, 109), (252, 101), (260, 93), (260, 85), (236, 85),
        (228, 85), (228, 93), (236, 93), (236, 101), (228, 101), (228, 109),
        (228, 117), (228, 125), (220, 125), (212, 117), (204, 109), (196, 101),
        (188, 93), (180, 93), (180, 101), (180, 109), (180, 117), (180, 125),
        (196, 145), (204, 145), (212, 145), (220, 145), (228, 145), (236, 145),
        (246, 141), (252, 125), (260, 129), (280, 133)
    ]
 # yapf: disable
    data['num_vehicles'] = 1
    data['depot'] = 0
    return data
def compute_euclidean_distance_matrix(locations):
    """Creates callback to return distance between points."""
    distances = {}
    for from_counter, from_node in enumerate(locations):
        distances[from_counter] = {}
        for to_counter, to_node in enumerate(locations):
            if from_counter == to_counter:
                distances[from_counter][to_counter] = 0
            else:
                # Euclidean distance
                distances[from_counter][to_counter] = (int(
                    math.hypot((from_node[0] - to_node[0]),
                               (from_node[1] - to_node[1]))))
    return distances
def print_solution(manager, routing, solution):
    """Prints solution on console."""
    print('Objective: {}'.format(solution.ObjectiveValue()))
    index = routing.Start(0)
    plan_output = 'Route:\n'
    route_distance = 0
    while not routing.IsEnd(index):
        plan_output += ' {} ->'.format(manager.IndexToNode(index))
        previous_index = index
        index = solution.Value(routing.NextVar(index))
```

```
        route_distance += routing.GetArcCostForVehicle(previous_index, index, 0)
    plan_output +=  '  {}\n'.format(manager.IndexToNode(index))
    print(plan_output)
    plan_output += 'Objective: {}m\n'.format(route_distance)

def main():
    """Entry point of the program."""
    # Instantiate the data problem.
    data = create_data_model()

    # Create the routing index manager.
    manager = pywrapcp.RoutingIndexManager(len(data['locations']),
    data['num_vehicles'], data['depot'])

    # Create Routing Model.
    routing = pywrapcp.RoutingModel(manager)

    distance_matrix = compute_euclidean_distance_matrix(data['locations'])
    def distance_callback(from_index, to_index):
        """Returns the distance between the two nodes."""
        # Convert from routing variable Index to distance matrix NodeIndex.
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
       return distance_matrix[from_node][to_node]
       transit_callback_index =
routing.RegisterTransitCallback(distance_callback)

    # Define cost of each arc.
    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

    # Setting first solution heuristic.
    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)

    # Solve the problem.
    solution = routing.SolveWithParameters(search_parameters)

    # Print solution on console.
    if solution:
        print_solution(manager, routing, solution)
if__name__== '__main__':
    main()
```

OUTPUT:

Objective: 2790

Route:

0 -> 1 -> 279 -> 2 -> 278 -> 277 -> 248 -> 247 -> 243 -> 242 -> 241 -> 240 -> 239 -> 238 -> 245 -> 244 -> 246 -> 249 -> 250 -> 229 -> 228 -> 231 -> 230 -> 237 -> 236 -> 235 -> 234 -> 233 -> 232 -> 227 -> 226 -> 225 -> 224 -> 223 -> 222 -> 218 -> 221 -> 220 -> 219 -> 202 -> 203 -> 204 -> 205 -> 207 -> 206 -> 211 -> 212 -> 215 -> 216 -> 217 -> 214 -> 213 -> 210 -> 209 -> 208 -> 251 -> 254 -> 255 -> 257 -> 256 -> 253 -> 252 -> 139 -> 140 -> 141 -> 142 -> 143 -> 199 -> 201 -> 200 -> 195 -> 194 -> 193 -> 191 -> 190 -> 189 -> 188 -> 187 -> 163 -> 164 -> 165 -> 166 -> 167 -> 168 -> 169 -> 171 -> 170 -> 172 -> 105 -> 106 -> 104 -> 103 -> 107 -> 109 -> 110 -> 113 -> 114 -> 116 -> 117 -> 61 -> 62 -> 63 -> 65 -> 64 -> 84 -> 85 -> 115 -> 112 -> 86 -> 83 -> 82 -> 87 -> 111 -> 108 -> 89 -> 90 -> 91 -> 102 -> 101 -> 100 -> 99 -> 98 -> 97 -> 96 -> 95 -> 94 -> 93 -> 92 -> 79 -> 88 -> 81 -> 80 -> 78 -> 77 -> 76 -> 74 -> 75 -> 73 -> 72 -> 71 -> 70 -> 69 -> 66 -> 68 -> 67 -> 57 -> 56 -> 55 -> 54 -> 53 -> 52 -> 51 -> 50 -> 49 -> 48 -> 47 -> 46 -> 45 -> 44 -> 43 -> 58 -> 60 -> 59 -> 42 -> 41 -> 40 -> 39 -> 38 -> 37 -> 36 -> 35 -> 34 -> 33 -> 32 -> 31 -> 30 -> 29 -> 124 -> 123 -> 122 -> 121 -> 120 -> 119 -> 118 -> 156 -> 157 -> 158 -> 173 -> 162 -> 161 -> 160 -> 174 -> 159 -> 150 -> 151 -> 155 -> 152 -> 154 -> 153 -> 128 -> 129 -> 130 -> 131 -> 18 -> 19 -> 20 -> 127 -> 126 -> 125 -> 28 -> 27 -> 26 -> 25 -> 21 -> 24 -> 22 -> 23 -> 13 -> 12 -> 14 -> 11 -> 10 -> 9 -> 7 -> 8 -> 6 -> 5 -> 275 -> 274 -> 273 -> 272 -> 271 -> 270 -> 15 -> 16 -> 17 -> 132 -> 149 -> 177 -> 176 -> 175 -> 178 -> 179 -> 180 -> 181 -> 182 -> 183 -> 184 -> 186 -> 185 -> 192 -> 196 -> 197 -> 198 -> 144 -> 145 -> 146 -> 147 -> 148 -> 138 -> 137 -> 136 -> 135 -> 134 -> 133 -> 269 -> 268 -> 267 -> 266 -> 265 -> 264 -> 263 -> 262 -> 261 -> 260 -> 258 -> 259 -> 276 -> 3 -> 4 -> 0