B.TECH. (2020-24)
Artificial Intelligence

LAB FILE

on

ARTIFICIAL INTELLIGENCE
[CSE401]

Submitted To
Mr. Soumya Ranjan Nayak

Submitted By
DEVANSHU KUMAR SINGH
A023119820024
4CSE11 (AI)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY
AMITY UNIVERSITY UTTAR PRADESH
NOIDA (U.P)

Experiment 1

Aim

Write a program to implement A* algorithm in python

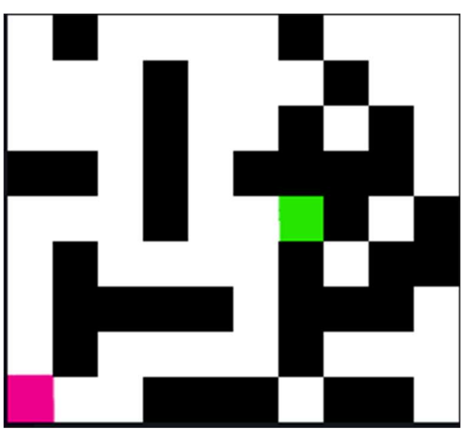
Program

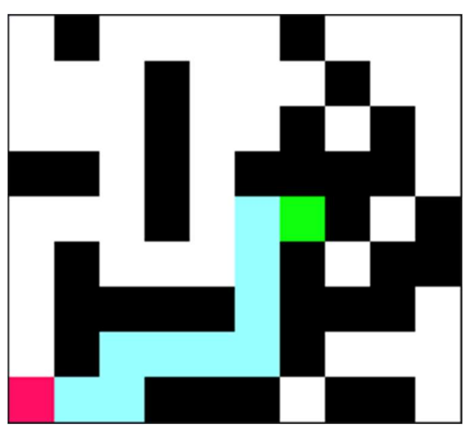
```
import numpy as np
import cv2

mazeName = "maze1"
fileType = "png"
maze = cv2.imread(f"./mazes/{mazeName}.{fileType}")[:, :, 0]

bounds = maze.shape

goal = [4,6]
start = [bounds[0]-1, 0]
stlessStart = [bounds[0]-1, 0]

path = []
visited = [] # will contain tuples of traversed coordinates

x = 0
y = 1

def distance(coords, goal):
    '''
        Returns distance between a node and the goal node
    '''
    euDist = ((coords[x]-goal[x])**2 + (coords[y]-goal[y])**2)**0.5
    return euDist

def moveGen(coords, bounds):
    '''
        Returns a list of possible blocks to move to
    '''
    gen = []
    moves = [
       [1,0],
       [-1,0],
       [0,1],
       [0,-1]
    ]
    for i in range(4):
        X = moves[i][x]
        Y = moves[i][y]
        new = [coords[x] + X, coords[y] + Y]
        if 0 <= new[x] < bounds[x] and 0 <= new[y] < bounds[y] and
maze[new[x], new[y]] == 255:
```

```
            gen.append(new)
    return gen

# getting the path
while True:

    gen = moveGen(start, bounds)
    dist = np.inf
    next = []
    # getting the next block
    for i in gen:
        if i not in visited:
            if distance(i, goal) < dist:
                dist = distance(i, goal)
                next = i
    if next == goal:
        break
    else:
        visited.append(start)
        start = next
        path.append(start)

# completing the maze image
maze = cv2.imread(f"./mazes/{mazeName}.{fileType}")
# defining the goal block color
maze[:, :, 2][goal[x], goal[y]] = 15
maze[:, :, 0][goal[x], goal[y]] = 15
# defining the start block color
maze[:, :, 0][stlessStart[x], stlessStart[y]] = 100
maze[:, :, 1][stlessStart[x], stlessStart[y]] = 15

for i in path:
    maze[:, :, 2][i[x], i[y]] = 150

cv2.imwrite(f"./mazes/{mazeName}Answer.png", maze)
```

Output

Problem Maze (Left), Solution Maze (Right)





Devanshu Kumar Singh
A023119820024

Experiment 2

Aim

Write a program to implement Single Player Game

Program

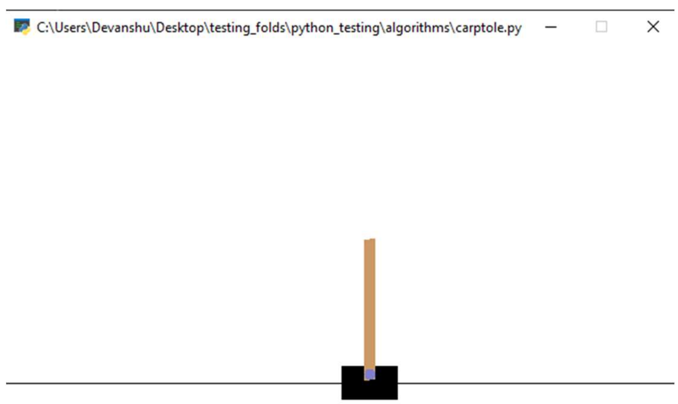
```
import gym
import numpy as np

env = gym.make("CartPole-v0")

def Qtable(state_vars, actions, bin_size=30):
    '''
    Routine that returns a Q-Table and bin values of each state for the
environment
    '''
    bins = [
        np.linspace(-4.8, 4.8, bin_size),
        np.linspace(-3, 3, bin_size),
        np.linspace(-0.418, 0.418, bin_size),
        np.linspace(-3, 3, bin_size),
    ]

    q_table = np.random.uniform(low=-1, high=1,
size=([bin_size]*state_vars + [actions]))

    return q_table, bins

def Discrete(state, bins):
    '''
    Routine that discretizes a state according to the Q-Table
    '''
    index = []
    for i in range(len(state)):
        index.append(np.digitize(state[i], bins[i]) - 1)
    return tuple(index)


# creating a qtable
qTable, bins = Qtable(
    len(env.observation_space.low), # or .high, doesn't really matter
    env.action_space.n
)


def Q(qTable, bins, episodes=5000, gamma=0.95, eta=0.1, timestep=1000,
epsilon=0.15):
    rewards = 0
    steps = 0
    runs = [0]
```

```
    data = {'max': [0], 'avg': [0]}
    solved = False

    for episode in range(1, episodes+1):
        currentState = Discrete(env.reset(), bins)
        score = 0
        done = False

        while not done:
            steps += 1

            if episode%episodes == 0:
                env.render()

            # checking to see whether to explore or exploit
            if np.random.uniform(0,1) < epsilon:
                action = env.action_space.sample()
            else:
                action = np.argmax(qTable[currentState])

            # getting new state
            obs, reward, done, _ = env.step(action)
            newState = Discrete(obs, bins)

            # increasing the reward
            score += reward

            # updating the Qtable
            if not done:
                maxFutureQ = np.max(qTable[newState])
                currentQ = qTable[currentState + (action,)]
                newQ = (1-eta)*currentQ + eta*(reward +
gamma*maxFutureQ)
                qTable[currentState + (action,)] = newQ

            currentState = newState

        else:
            rewards += score
            runs.append(score)
            if score > 195 and steps >= 100 and solved == False: #
considered as a solved:
                solved = True
                print('Solved in episode : {}'.format(episode))

        # Timestep value update
        if episode % timestep == 0:
            print('Episode : {} | Reward -> {} | Max reward :
{}'.format(episode,rewards/timestep, max(runs)))
            data['max'].append(max(runs))
            data['avg'].append(rewards/timestep)
            if rewards/timestep >= 195:
                print('Solved in episode : {}'.format(episode))
            rewards, runs= 0, [0]
```

```
q_table, bins = Qtable(len(env.observation_space.low),
env.action_space.n, bin_size=25)

Q(q_table, bins, eta=0.15, gamma=0.995, episodes=5*10**3, timestep=1000)
```

Output

```
Episode : 1000 | Reward -> 76.382 | Max reward : 200.0
Episode : 2000 | Reward -> 146.145 | Max reward : 200.0
Episode : 3000 | Reward -> 158.031 | Max reward : 200.0
Episode : 4000 | Reward -> 165.398 | Max reward : 200.0
Episode : 5000 | Reward -> 165.811 | Max reward : 200.0
Episode : 6000 | Reward -> 167.955 | Max reward : 200.0
Episode : 7000 | Reward -> 169.561 | Max reward : 200.0
Episode : 8000 | Reward -> 174.245 | Max reward : 200.0
Episode : 9000 | Reward -> 173.848 | Max reward : 200.0
Episode : 10000 | Reward -> 175.079 | Max reward : 200.0
```



Experiment 3

Aim

Write a program to implement Tic-Tac-Toe game problem

Program

```
import os
from tabnanny import check
# Python3 program to find the next optimal move for a player
player, opponent = 'x', 'o'

# This function returns true if there are moves
# remaining on the board. It returns false if
# there are no moves left to play.
def isMovesLeft(board) :

	for i in range(3) :
		for j in range(3) :
			if (board[i][j] == '_') :
				return True
	return False

def evaluate(b) :

	# Checking for Rows for X or O victory.
	for row in range(3) :
		if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
			if (b[row][0] == player) :
				return 10
			elif (b[row][0] == opponent) :
				return -10

	# Checking for Columns for X or O victory.
	for col in range(3) :

		if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :

			if (b[0][col] == player) :
				return 10
			elif (b[0][col] == opponent) :
				return -10

	# Checking for Diagonals for X or O victory.
	if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :

		if (b[0][0] == player) :
			return 10
		elif (b[0][0] == opponent) :
			return -10
```

Devanshu Kumar Singh
A023119820024

```
	if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :

		if (b[0][2] == player) :
			return 10
		elif (b[0][2] == opponent) :
			return -10

	# Else if none of them have won then return 0
	return 0

# This is the minimax function. It considers all
# the possible ways the game can go and returns
# the value of the board
def minimax(board, depth, isMax) :
	score = evaluate(board)

	# If Maximizer has won the game return his/her
	# evaluated score
	if (score == 10) :
		return score

	# If Minimizer has won the game return his/her
	# evaluated score
	if (score == -10) :
		return score

	# If there are no more moves and no winner then
	# it is a tie
	if (isMovesLeft(board) == False) :
		return 0

	# If this maximizer's move
	if (isMax) :
		best = -1000

		# Traverse all cells
		for i in range(3) :
			for j in range(3) :

				# Check if cell is empty
				if (board[i][j]=='_') :

					# Make the move
					board[i][j] = opponent

					# Call minimax recursively and choose
					# the maximum value
					best = max( best, minimax(board,
											depth +
1,
											not
isMax) )

					# Undo the move
```

```
                              board[i][j] = '_'
            return best

      # If this minimizer's move
      else :
            best = 1000

            # Traverse all cells
            for i in range(3) :
                  for j in range(3) :

                        # Check if cell is empty
                        if (board[i][j] == '_') :

                              # Make the move
                              board[i][j] = player

                              # Call minimax recursively and choose
                              # the minimum value
                              best = min(best, minimax(board, depth + 1,
not isMax))

                              # Undo the move
                              board[i][j] = '_'
            return best

# This will return the best possible move for the player
def findBestMove(board) :
      bestVal = -1000
      bestMove = (-1, -1)

      # Traverse all cells, evaluate minimax function for
      # all empty cells. And return the cell with optimal
      # value.
      for i in range(3) :
            for j in range(3) :

                  # Check if cell is empty
                  if (board[i][j] == '_') :

                        # Make the move
                        board[i][j] = opponent

                        # compute evaluation function for this
                        # move.
                        moveVal = minimax(board, 0, False)

                        # Undo the move
                        board[i][j] = '_'

                        # If the value of the current move is
                        # more than the best value, then update
                        # best/
                        if (moveVal > bestVal) :
```

```
                              bestMove = (i, j)
                              bestVal = moveVal

      print("The value of the best Move is :", bestVal)
      print()
      return bestMove

def checkWin(board):
      for r in board:
            if r == ['o','o','o']:
                  return 1
            if r == ['x','x','x']:
                  return 0
      for i in range(len(board)):
            c = []
            for j in range(len(board)):
                  c.append(board[j][i])

            if c == ['o','o','o']:
                  return 1
            if c == ['x','x','x']:
                  return 0

      if [board[0][0], board[1][1], board[2][2]] == ['o','o','o']:
            return 1
      if [board[0][0], board[1][1], board[2][2]] == ['x','x','x']:
            return 0

      if [board[0][2], board[1][1], board[2][0]] == ['o','o','o']:
            return 1
      if [board[0][2], board[1][1], board[2][0]] == ['x','x','x']:
            return 0

def printBoard(board):
      for i in board:
            for j in i:
                  print(j, end=" ")
            print("\n")
# Driver code
board = [
      [ '_', '_', '_' ],
      [ '_', '_', '_' ],
      [ '_', '_', '_' ]
]

bestMove = findBestMove(board)

print("The Optimal Move is :")
print("ROW:", bestMove[0], " COL:", bestMove[1])

win = False
draw = False

while not (win or draw):
```

```
    printBoard(board)
    print()
    bestMove = findBestMove(board)
    board[bestMove[0]][bestMove[1]] = 'o'
    if checkWin(board) == 1:
          print('cpu wins!')
          win = True
          break

    r = int(input("enter your move (r) : "))
    c = int(input("enter your move (c) : "))
    board[r-1][c-1] = 'x'

    if checkWin(board) == 0:
          print('player wins!')
          win = True
          break

    if checkWin(board) != 1 or checkWin(board) != 0:
          print("drawwww!!!")
```

Output

```
_ _ _

_ _ _

_ _ _


The value of the best Move is : 0

enter your move (r) : 1
enter your move (c) : 2
drawwww!!!
_ x _

_ o _

_ _ _


The value of the best Move is : 0
```

```
enter your move (r) : 1
enter your move (c) : 3
drawwww!!!
_ x x

_ o _

_ o _


The value of the best Move is : 0

enter your move (r) : 1
enter your move (c) : 1
player wins!
```

Experiment 4

Aim

Implement Brute force solution to the Knapsack problem in Python

Program

```
def knapSack(W, wt, val, n):
   # initial conditions
   if n == 0 or W == 0 :
      return 0
   # If weight is higher than capacity then it is not included
   if (wt[n-1] > W):
      return knapSack(W, wt, val, n-1)
   # return either nth item being included or not
   else:
      return max(val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
         knapSack(W, wt, val, n-1))

val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print (knapSack(W, wt, val, n))
```

Output

```
220
PS C:\Users\Devanshu\Desktop\testing_folds\python_testing>
```

Devanshu Kumar Singh
A023119820024

Experiment 5

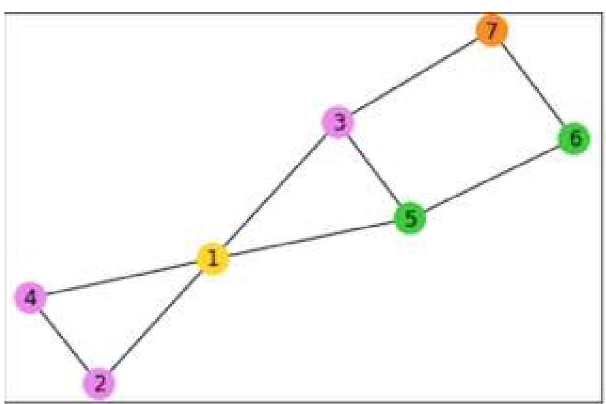
Aim

Implement Graph colouring problem using python

Program

```
import networkx as nx

# creating the graph
network = nx.Graph()

# adding nodes
network.add_nodes_from([4,5,6,7])
network.add_edge(1,4)
network.add_edge(1,5)
network.add_edge(2,4)
network.add_edge(3,5)
network.add_edge(5,6)
network.add_edge(7,3)
network.add_edge(7,6)

# defining color list
color_list = ["gold", "violet", "violet", "violet",
              "limegreen", "limegreen", "darkorange"]

# final colored graph
nx.draw_networkx(network,node_color=color_list, with_labels=True)
```

Output



Devanshu Kumar Singh
A023119820024

Experiment 6

Aim

Write a program to implement BFS for water jug problem using Python

Program

```
x_capacity = int(input("Enter Jug 1 capacity:"))
y_capacity = int(input("Enter Jug 2 capacity:"))
end = int(input("Enter target volume:"))

def bfs(start, end, x_capacity, y_capacity):
      path = []
      front = []
      front.append(start)
      visited = []
      #visited.append(start)
      while(not (not front)):
            current = front.pop()
            x = current[0]
            y = current[1]
            path.append(current)
            if x == end or y == end:
                  print("Found!")
                  return path
            # rule 1
            if current[0] < x_capacity and ([x_capacity, current[1]] not
in visited):
                  front.append([x_capacity, current[1]])
                  visited.append([x_capacity, current[1]])

            # rule 2
            if current[1] < y_capacity and ([current[0], y_capacity] not
in visited):
                  front.append([current[0], y_capacity])
                  visited.append([current[0], y_capacity])

            # rule 3
            if current[0] > x_capacity and ([0, current[1]] not in
visited):
                  front.append([0, current[1]])
                  visited.append([0, current[1]])

            # rule 4
            if current[1] > y_capacity and ([x_capacity, 0] not in
visited):
                  front.append([x_capacity, 0])
                  visited.append([x_capacity, 0])

            # rule 5
```

```
            #(x, y) -> (min(x + y, x_capacity), max(0, x + y - 
x_capacity)) if y > 0
            if current[1] > 0 and ([min(x + y, x_capacity), max(0, x + y 
- x_capacity)] not in visited):
                  front.append([min(x + y, x_capacity), max(0, x + y - 
x_capacity)])
                  visited.append([min(x + y, x_capacity), max(0, x + y - 
x_capacity)])

            # rule 6
            # (x, y) -> (max(0, x + y - y_capacity), min(x + y, 
y_capacity)) if x > 0
            if current[0] > 0  and ([max(0, x + y - y_capacity), min(x + 
y, y_capacity)] not in visited):
                  front.append([max(0, x + y - y_capacity), min(x + y, 
y_capacity)])
                  visited.append([max(0, x + y - y_capacity), min(x + y, 
y_capacity)])

      return "Not found"

def gcd(a, b):
      if a == 0:
            return b
      return gcd(b%a, a)

# start state: x = 0 , y = 0
start = [0, 0]
#end = 2
#x_capacity = 4
#y_capacity = 3

# condition for getting a solution:
# the target volume 'end' should be a multiple of gcd(a,b)

if end % gcd(x_capacity,y_capacity) == 0:
      print(bfs(start, end, x_capacity, y_capacity))
else:
      print("No solution possible for this combination.")

input()
```

Output

```
Solution for water jug problem
Enter Jug 1 capacity:4
Enter Jug 2 capacity:3
Enter target volume:2
Found!
[[0, 0], [0, 3], [3, 0], [3, 3], [4, 2]]
```

Devanshu Kumar Singh
A023119820024

Experiment 7

Aim

Write a program to implement DFS using Python

Program

```
NodeData = {
    "A": ["B", "C"],
    "B": ["D", "E", "A"],
    "C": ["F", "G", "A"],
    "D": ["B"],
    "E": ["B", "F"],
    "F": ["C", "H", "E"],
    "G": ["C"],
    "H": ["F"]
}
```

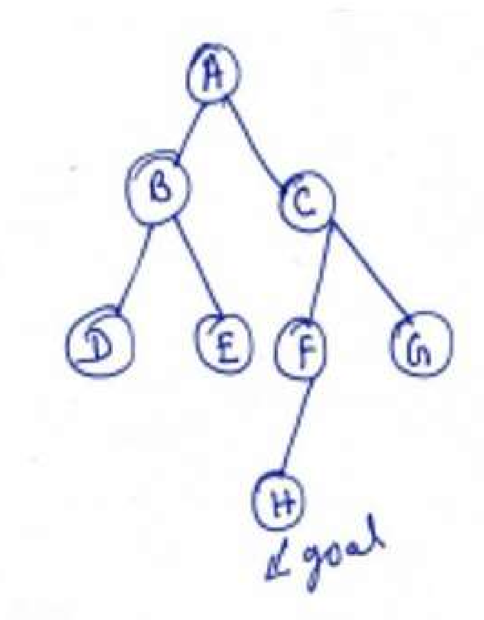
```
# WHEN NodeData WAS KEPT IN A SEPARATE FILE, IMPORT
from jsonNodeData import NodeData

# CLASS FOR NODE OBJECT
class Node:
    def __init__(self, data, children) -> None:
        self.data = data
        self.children = children

nodes = []

# ADDING NODES TO THE GRAPH
for node in NodeData:
    new = None
    if NodeData[node]:
        new = Node(
            data=node,
            children=NodeData[node]
        )
    else:
        new = Node(
            data=node,
            children=None
        )
    nodes.append(new)

open = nodes
waiting = [None]
visited = []

# TERMINATION DATA
start = input("Enter start node: ").upper()
goal = input("Enter end node: ").upper()
currentNode = Node(data=None, children=None) #PLACEHOLDER NODE
```

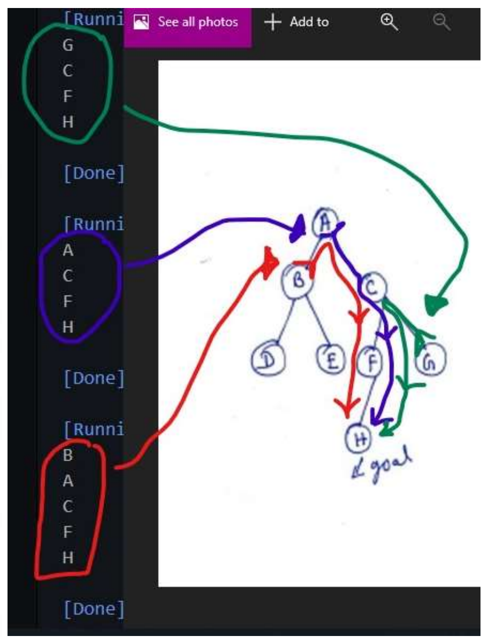
```
# CHECKING IF START NODE EXISTS
if start not in NodeData:
    print("Start node doesn't exist")
    exit

# GET THE NEW NODE TO WORK WITH
for node in open:
    if node.data == start:
        waiting.append(node)
        open.remove(node)

# TRAVERSING THE GRAPH
while currentNode:
    if currentNode.data == goal:
        # insert at top
        visited.append(currentNode)
        break
    else:
        # ADDING THE CURRENT NODE TO THE VISITED STACK
        visited.append(currentNode)
        # NOW ADDING CHILDREN OF CURRENTNODE
        # TO WAITING
        if currentNode.children:
            for child in currentNode.children:
                for node in open:
                    if node.data == child:
                        waiting.append(node)
                        open.remove(node)

        currentNode = waiting.pop()

# REMOVING THE PLACEHOLDER NODE
visited = visited[1:]

# ROUTINE TO CHECK IF AN ARRAY CONTAINS
# ANY OF ITS ELEMENTS IN ANOTHER ARRAY
def subCheck(arr1, arr2):
    flag = False
    for i in arr1:
        if i in arr2:
            flag = True
    return flag

# ROUTINE TO REMOVE NODE FROM AN ARRAY
# RETURNS THE NEW ARRAY
def removeNode(node, array):
    for n in array:
        if node.data == n.data:
            array.remove(node)
    return array

# TRACING BACK THE PATH FROM START NODE TO GOAL NODE
visitedCopy = visited[:]
```

```
for i in range(len(visitedCopy)):
    node = visitedCopy[i]
    subVisitedData = [chnode.data for chnode in visitedCopy[i+1:]]
    if not subCheck(node.children, subVisitedData):
        if node.data != goal:
            visited = removeNode(node, visited)

if visited[-1].data == goal:
    print("Goal node reached!")
    # THE PATH FROM START NODE TO GOAL NODE
    for node in visited:
        print(node.data)
else:
    print("Goal doesn't exist.")
```

Output

Problem graph (Left), Solution paths (Right)





Experiment 8

Aim

Tokenization of word and Sentences with the help of NLTK package

Program

```
from nltk.tokenize import word_tokenize
s = "Good muffins cost $3.88 in New York.  Please buy me two of them.
Thanks."
Word_tokenize(s)
```

Output

```
['Good',
 'muffins',
 'cost',
 '$',
 '3.88',
 'in',
 'New',
 'York',
 '.',
 'Please',
 'buy',
 'me',
 'two',
 'of',
 'them',
 '.',
 'Thanks',
 '.']
```

Devanshu Kumar Singh
A023119820024

Experiment 9

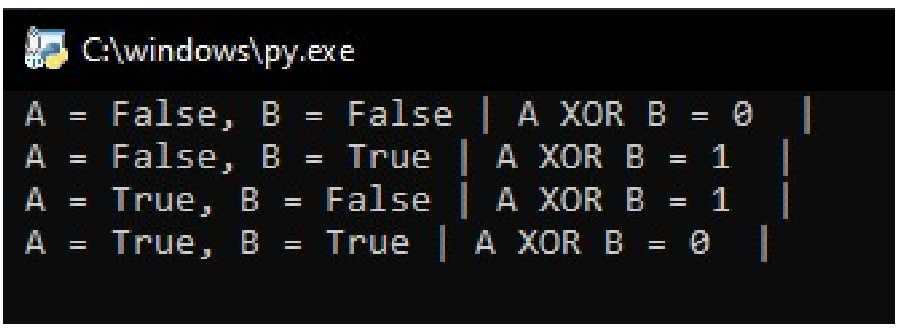
Aim

Design an XOR truth table using Python

Program

```
def XOR (a, b):
    if a != b:
        return 1
    else:
        return 0

print(" A = False, B = False | A XOR B =",XOR(False,False)," | ")
print(" A = False, B = True | A XOR B =",XOR(False,True)," | ")
print(" A = True, B = False | A XOR B =",XOR(True,False)," | ")
print(" A = True, B = True | A XOR B =",XOR(True,True)," | ")
```

Output



Devanshu Kumar Singh
A023119820024

Experiment 10

Aim

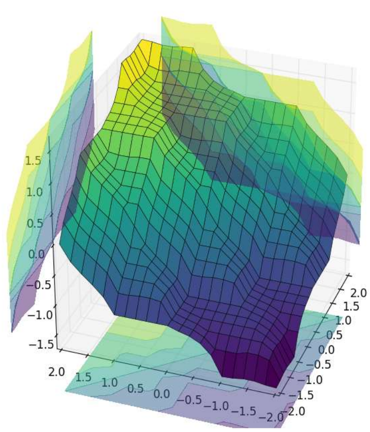
Study of scikit fuzzy

Program

```
import numpy as np
import skfuzzy.control as ctrl

universe = np.linspace(-2, 2, 5)

error = ctrl.Antecedent(universe, 'error')
delta = ctrl.Antecedent(universe, 'delta')
output = ctrl.Consequent(universe, 'output')

names = ['nb', 'ns', 'ze', 'ps', 'pb']
error.automf(names=names)
delta.automf(names=names)
output.automf(names=names)

rule0 = ctrl.Rule(antecedent=((error['nb'] & delta['nb']) |
                              (error['ns'] & delta['nb']) |
                              (error['nb'] & delta['ns'])),
                  consequent=output['nb'], label='rule nb')

rule1 = ctrl.Rule(antecedent=((error['nb'] & delta['ze']) |
                              (error['nb'] & delta['ps']) |
                              (error['ns'] & delta['ns']) |
                              (error['ns'] & delta['ze']) |
                              (error['ze'] & delta['ns']) |
                              (error['ze'] & delta['nb']) |
                              (error['ps'] & delta['nb'])),
                  consequent=output['ns'], label='rule ns')

rule2 = ctrl.Rule(antecedent=((error['nb'] & delta['pb']) |
                              (error['ns'] & delta['ps']) |
                              (error['ze'] & delta['ze']) |
                              (error['ps'] & delta['ns']) |
                              (error['pb'] & delta['nb'])),
                  consequent=output['ze'], label='rule ze')

rule3 = ctrl.Rule(antecedent=((error['ns'] & delta['pb']) |
                              (error['ze'] & delta['pb']) |
                              (error['ze'] & delta['ps']) |
                              (error['ps'] & delta['ps']) |
                              (error['ps'] & delta['ze']) |
                              (error['pb'] & delta['ze']) |
                              (error['pb'] & delta['ns'])),
                  consequent=output['ps'], label='rule ps')
```

```
rule4 = ctrl.Rule(antecedent=((error['ps'] & delta['pb']) |
                              (error['pb'] & delta['pb']) |
                              (error['pb'] & delta['ps'])),
                  consequent=output['pb'], label='rule pb')

upsampled = np.linspace(-2, 2, 21)
x, y = np.meshgrid(upsampled, upsampled)
z = np.zeros_like(x)

for i in range(21):
    for j in range(21):
        sim.input['error'] = x[i, j]
        sim.input['delta'] = y[i, j]
        sim.compute()
        z[i, j] = sim.output['output']

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')

surf = ax.plot_surface(x, y, z, rstride=1, cstride=1, cmap='viridis',
                       linewidth=0.4, antialiased=True)

cset = ax.contourf(x, y, z, zdir='z', offset=-2.5, cmap='viridis',
alpha=0.5)
cset = ax.contourf(x, y, z, zdir='x', offset=3, cmap='viridis',
alpha=0.5)
cset = ax.contourf(x, y, z, zdir='y', offset=3, cmap='viridis',
alpha=0.5)

ax.view_init(30, 200)
```

Output



Devanshu Kumar Singh
A023119820024