**ARTIFICIAL INTELLIGENCE (Code 18CSC305J)**

# B.Tech (CSE) – 3nd year/6th Semester

## Name: Aditya Saxena

**Registration No.:RA2011003030223**



## DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING FACULTY OF ENGINEERING & TECHNOLOGY

### SRM INSTITUTE OF SCIENCE & TECHNOLOGY, DELHINCR CAMPUS, MODINAGAR

SIKRI KALAN, DELHI MEERUT ROAD, DIST. – GHAZIABAD - 201204

**https://**[**www.srmup.in/**](http://www.srmup.in/)

**Even Semester (2022-2023)**



**BONAFIDE CERTIFICATE**

**Registration no.RA2011003030223** *Certified to be the bonafide record of work done by* ***Aditya Saxena*** *of 6th semester 3rd year B.TECH degree course in SRM INSTITUTE OF SCIENCE*

*AND TECHNOLOGY, NCR Campus of Department of Computer Science &*

*Engineering, in ARTIFICIAL INTELLIGENCE LAB, during the academic year 2022-2023.*

**Assistant Professor Head of the Department (CSE)**

**(Dr. R. P. Mahapatra)**

*Submitted for university examination held on\_\_\_/\_\_\_/\_\_\_ at SRM IST, NCR Campus.*

**Internal Examiner-I Internal Examiner-II**

**Lab-1**

**Aim**- Write a python program for Water Jug and Tic Tac Toe problem.

### **Water Jug Problem problem**

### **Code:-**

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

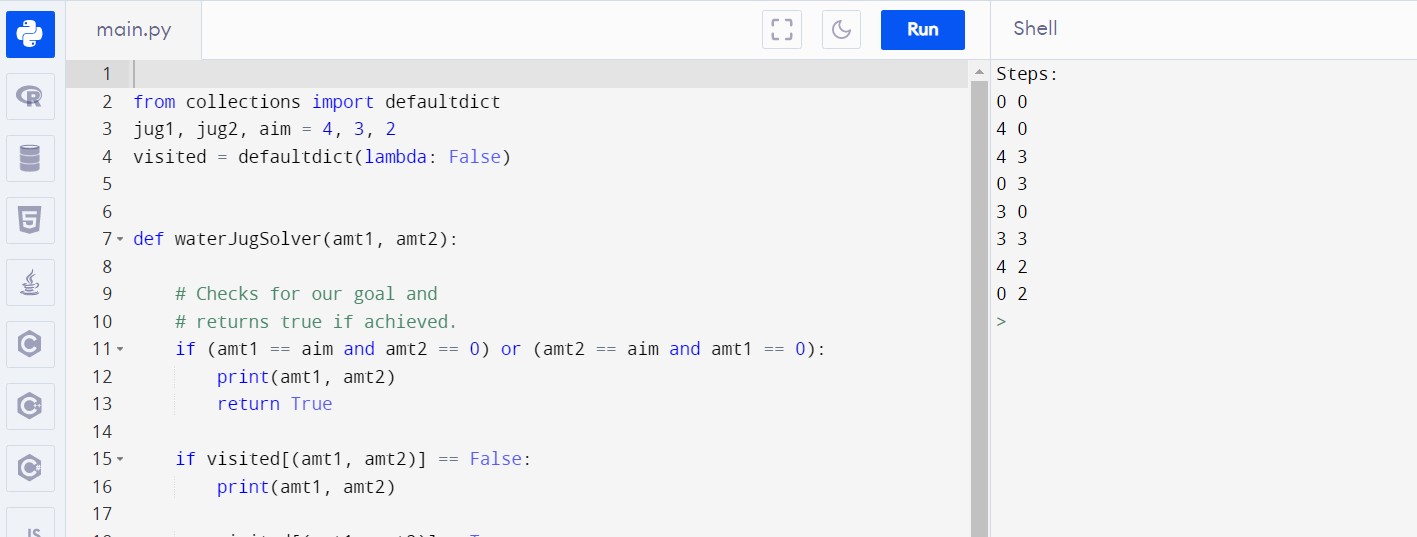
waterJugSolver(amt1, 0) or waterJugSolver(jug1, amt2) or waterJugSolver(amt1, jug2) or waterJugSolver(amt1 + min(amt2, (jug1-amt1)), amt2 - min(amt2, (jug1-amt1))) or waterJugSolver(amt1 - min(amt1, (jug2-amt2)), amt2 + min(amt1, (jug2-amt2))))

else:

return False print("Steps: ")

waterJugSolver(0, 0)

**Output:-**



### **Tic tac toe problem**

### **Code:-**

import numpy as np import random

from time import sleep def create\_board():

return(np.array([[0, 0, 0],[0, 0, 0],[0, 0, 0]]))

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)

def random\_place(board, player): selection = possibilities(board)

current\_loc = random.choice(selection) board[current\_loc] = 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

if win == True:

return(win)

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

if win == True:

return(win)

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)

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

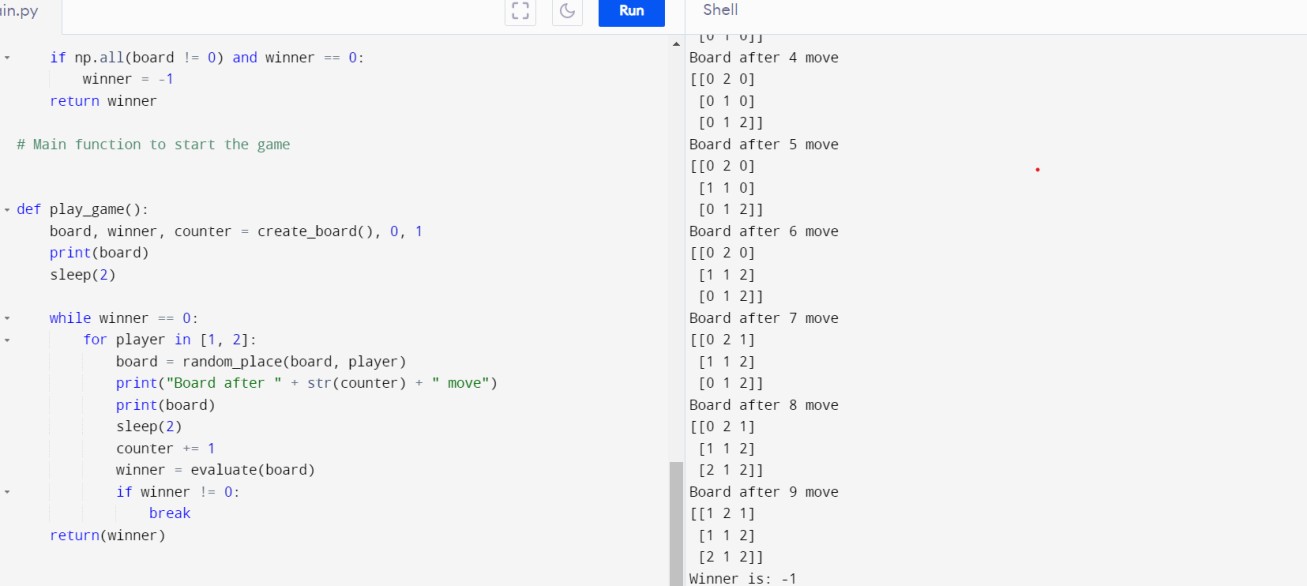
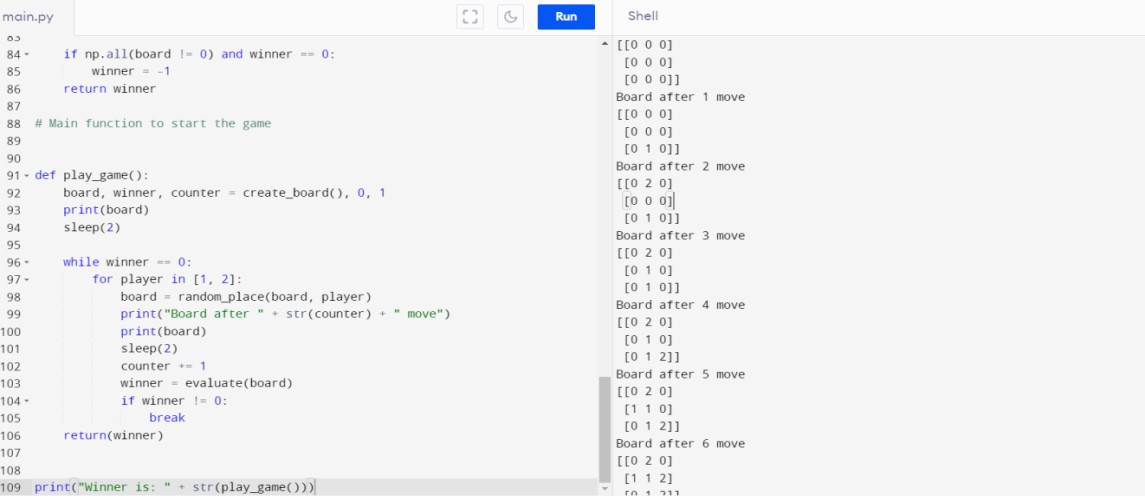
winner = evaluate(board) if winner != 0:

break

return(winner)

print("Winner is: " + str(play\_game()))

**Output:-**



**Result:-** Hence, Water Jug and Tic Tac Toe problem has been solved successfully.

# Lab-2

### **AIM-** Write a python program for Travelling salesman problem & Missionaries and cannibals.

**Travelling Salesman Problem**

**Code:-**

import sys import math

def nearest\_neighbor(cities, start):

n = len(cities)

unvisited = [i for i in range(n) if i != start]

current = start

tour = [current]

while unvisited:

next\_city = unvisited[0]

min\_distance = sys.maxsize

for i in range(1, len(unvisited)):

city = unvisited[i]

distance = math.sqrt((cities[current][0] - cities[city][0]) \*\* 2 + (cities[current][1] - cities[city][1]) \*\* 2)

if distance < min\_distance:

min\_distance = distance next\_city = city

tour.append(next\_city)

unvisited.remove(next\_city)

current = next\_city

return tour

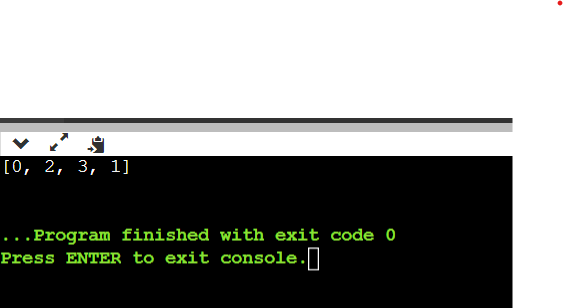
cities = [(0, 0), (1, 1), (1, 0), (0, 1)]

start = 0

tour = nearest\_neighbor(cities, start)

print(tour)

**Output:-**



**Missionaries and Cannibals**

**Code:-**

from colletions import deque

def is\_safe(state):

m, c = state

return m >= 0 and c >= 0 and (m == 0 or m >= c)

def bfs(start, goal, q):

q.append(start) while q:

state = q.popleft() if state == goal:

return state m, c = state

for (mm, cc) in [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]:

next\_state = (m - mm, c - cc) if is\_safe(next\_state):

q.append(next\_state) return None

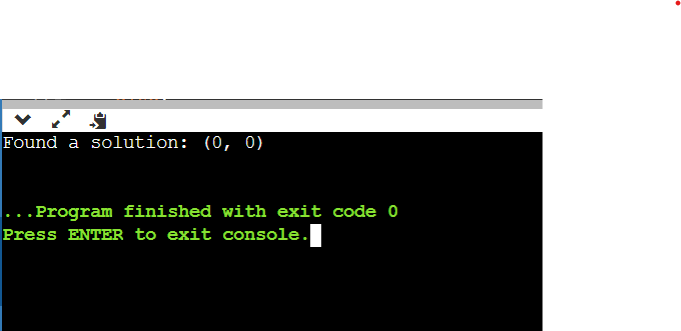
if name == ' main ': start = (3, 3)

goal = (0, 0) q = deque()

result = bfs(start, goal, q) if result:

print("Found a solution:", result) else:

print("No solution found")

**Output:-**

### **Result:-** Hence, Travelling salesman problem & Missionaries and cannibals has been solved successfully.

**Lab-3**

**AIM-**Write a python program for crypto-arithmetic problem and Sudoku problem.

**Crypto-arithmetic problem**

**Code:-**

import re

solved = False

def solve(letters, values, visited, words):

global solved

if len(set) == len(values):

map = {}

for letter, val in zip(letters,values):

map[letter] = val

if map[words[0][0]] == 0 or map[words[1][0]] == 0 or map[words[2][0]] == 0:

return

word1, word2, res = "", "", ""

for c in words[0]:

word1 += str(map[c])

for c in words[1]:

word2 += str(map[c])

for c in words[2]:

res += str(map[c])

if int(word1) + int(word2) == int(res):

print("{} + {} = {}\t{}".format(word1, word2, res, map))

solved = True

return

for i in range(10):

if not visited[i]:

visited[i] = True

values.append(i)

solve(letters, values, visited, words)

values.pop()

visited[i] = False

print("\nCRYPTARITHMETIC PUZZLE SOLVER")

print("WORD1 + WORD2 = RESULT")

word1 = input("Enter WORD1: ").upper()

word2 = input("Enter WORD2: ").upper()

result = input("Enter RESULT: ").upper()

if len(result) > (max(len(word1), len(word2)) + 1):

print("\n0 Solutions!")

else:

set = []

for c in word1:

if c not in set:

set.append(c)

for c in word2:

if c not in set:

set.append(c)

for c in result:

if c not in set:

set.append(c)

if len(set) > 10:

print("\nNo solutions!")

exit()

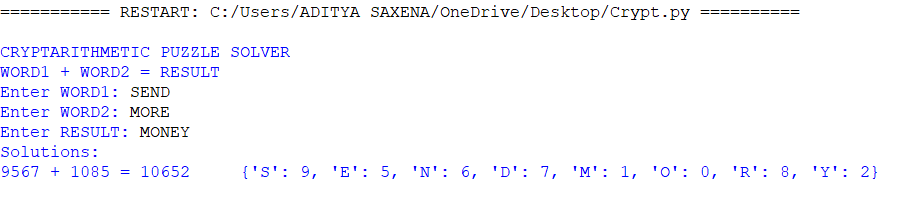
print("Solutions:")

solve(set, [], [False for \_ in range(10)], [word1, word2, result])

if not solved:

print("\n0 solutions!")

**Output:-**

****

**Sudoku Solver**

**Code:-**

M = 9

def puzzle(a):

for i in range(M):

for j in range(M):

print(a[i][j],end = " ")

print()

def solve(grid, row, col, num):

for x in range(9):

if grid[row][x] == num:

return False

for x in range(9):

if grid[x][col] == num:

return False

startRow = row - row % 3

startCol = col - col % 3

for i in range(3):

for j in range(3):

if grid[i + startRow][j + startCol] == num:

return False

return True

def Suduko(grid, row, col):

if (row == M - 1 and col == M):

return True

if col == M:

row += 1

col = 0

if grid[row][col] > 0:

return Suduko(grid, row, col + 1)

for num in range(1, M + 1, 1):

if solve(grid, row, col, num):

grid[row][col] = num

if Suduko(grid, row, col + 1):

return True

grid[row][col] = 0

return False

grid = [[2, 5, 0, 0, 3, 0, 9, 0, 1],

[0, 1, 0, 0, 0, 4, 0, 0, 0],

[4, 0, 7, 0, 0, 0, 2, 0, 8],

[0, 0, 5, 2, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 9, 8, 1, 0, 0],

[0, 4, 0, 0, 0, 3, 0, 0, 0],

[0, 0, 0, 3, 6, 0, 0, 7, 2],

[0, 7, 0, 0, 0, 0, 0, 0, 3],

[9, 0, 3, 0, 0, 0, 6, 0, 4]]

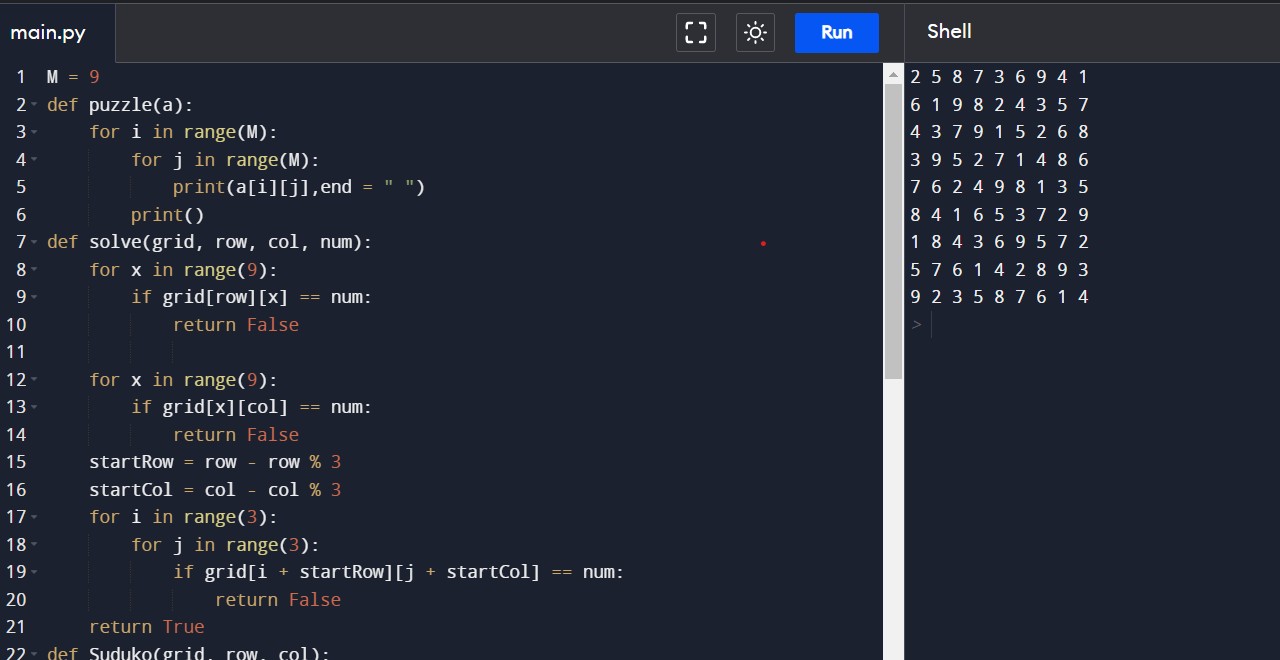
if (Suduko(grid, 0, 0)):

puzzle(grid)

else:

print("Solution does not exist:(")

**Output:-**



### **Result:-** Hence, implementation of Crypto-arithmetic problem and Sudoku problem has been done successfully.

**Lab-4**

**AIM:-** Write a python program for Implementation and Analysis of DFS and BFS for an application.

**Water Jug problem using BFS**

**Code:-**

from collections import deque

def BFS(a, b, target):

m = {}

isSolvable = False

path = []

q = deque()

q.append((0, 0))

while (len(q) > 0):

u = q.popleft()

if ((u[0], u[1]) in m):

continue

if ((u[0] > a or u[1] > b or u[0] < 0 or u[1] < 0)):

continue

path.append([u[0], u[1]])

m[(u[0], u[1])] = 1

if (u[0] == target or u[1] == target):

isSolvable = True

if (u[0] == target):

if (u[1] != 0):

path.append([u[0], 0])

else:

if (u[0] != 0):

path.append([0, u[1]])

sz = len(path)

for i in range(sz):

print("(", path[i][0], ",",path[i][1], ")")

break

q.append([u[0], b])

q.append([a, u[1]])

for ap in range(max(a, b) + 1):

c = u[0] + ap

d = u[1] - ap

if (c == a or (d == 0 and d >= 0)):

q.append([c, d])

c = u[0] - ap

d = u[1] + ap

if ((c == 0 and c >= 0) or d == b):

q.append([c, d])

q.append([a, 0])

q.append([0, b])

if (not isSolvable):

print ("No solution")

Jug1 = int(input("Jug 1 quantity : "))

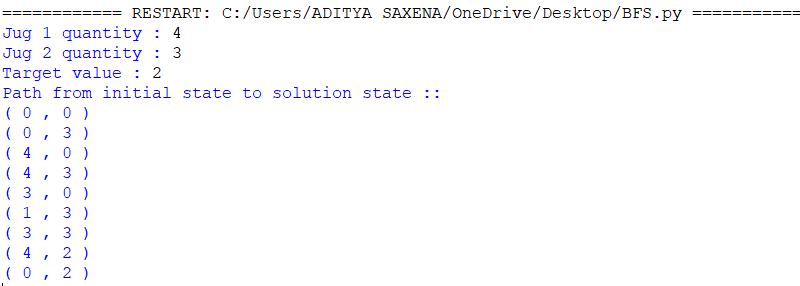
Jug2 = int(input("Jug 2 quantity : "))

target = int(input("Target value : "))

print("Path from initial state " "to solution state ::")

BFS(Jug1, Jug2, target)

**Output:-**

****

**Binary Search Tree Implementation using DFS**

**Code:-**

class TreeNode:

def \_\_init\_\_(self, val=0, left=None, right=None):

self.val = val

self.left = left

self.right = right

def rangeSumBST(root, L, R):

ans = 0

stack = [root]

while stack:

node = stack.pop()

if node:

if L <= node.val <= R:

ans += node.val

if L < node.val:

stack.append(node.left)

if node.val < R:

stack.append(node.right)

return ans

bst = TreeNode(10)

bst.left = TreeNode(5)

bst.right = TreeNode(15)

bst.left.left = TreeNode(3)

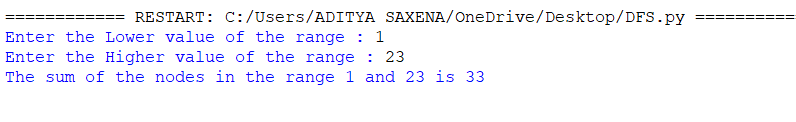
min = int(input("Enter the Lower value of the range : "))

max = int(input("Enter the Higher value of the range : "))

sol = rangeSumBST(bst, min, max)

print(f"The sum of the nodes in the range {min} and {max} is {sol}")

**Output:-**



**Result:-** Hence, implementation and analysis of DFS and BFS has been done successfully.

**Lab-5**

**AIM:-** Write a python program to implement best first search algorithm and A\* algorithm.

**Best First Search Algorithm**

**Code:-**

from queue import PriorityQueue

import networkx as nx

def best\_first\_search(source, target, n):

visited = [0] \* n

visited[source] = True

pq = PriorityQueue()

pq.put((0, source))

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))

G = nx.Graph()

v = int(input("Enter the number of nodes: "))

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

e = int(input("Enter the number of edges: "))

print("Enter the edges along with their weights:")

for i in range(e):

x, y, z = list(map(int, input().split()))

addedge(x, y, z)

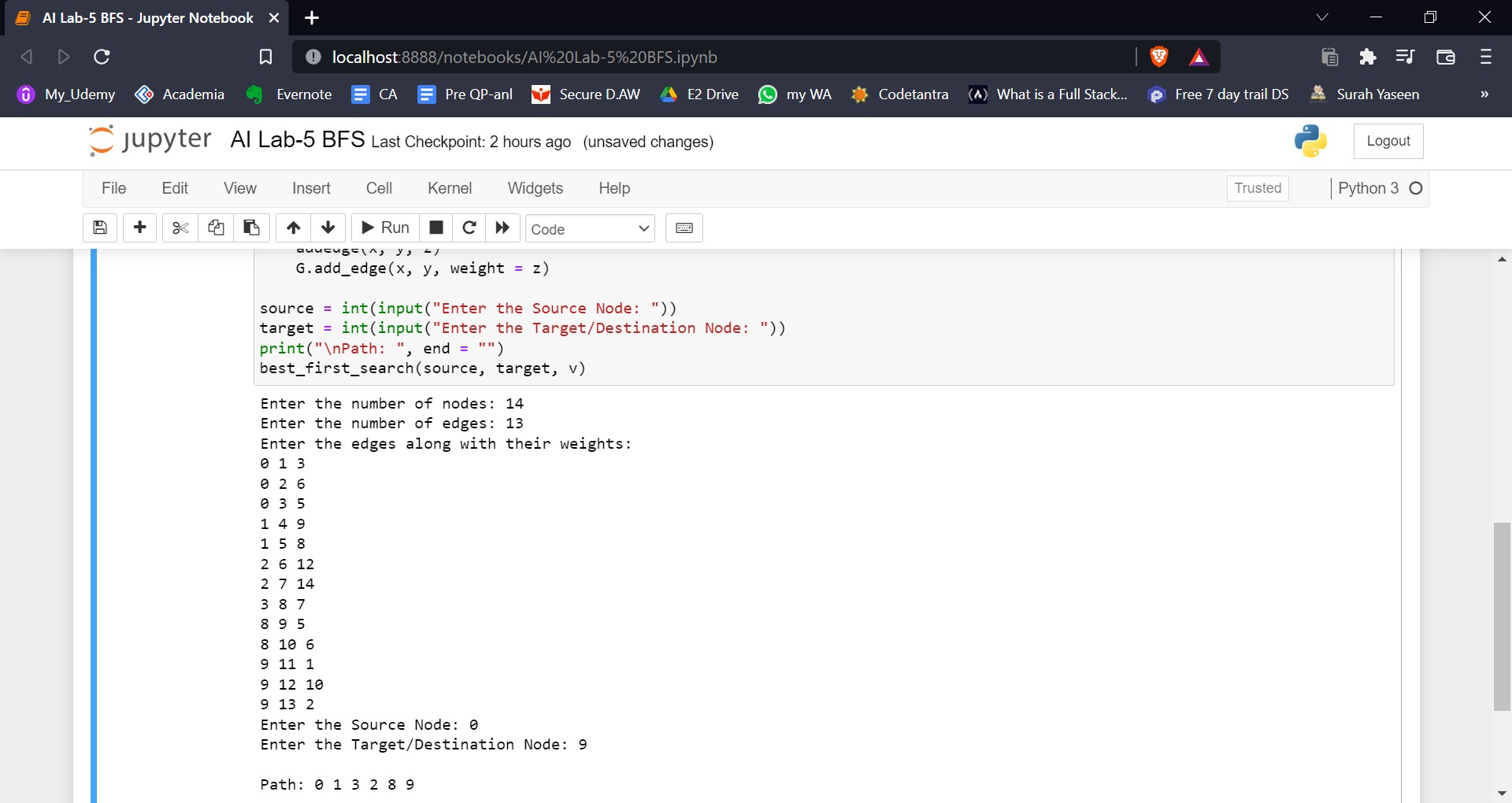
G.add\_edge(x, y, weight = z)

source = int(input("Enter the Source Node: "))

target = int(input("Enter the Target/Destination Node: "))

print("\nPath: ", end = "")

best\_first\_search(source, target, v)

**Output:-**

**A\* Algorithm**

**Code:-**

from collections import deque

class Graph:

def \_\_init\_\_(self, adjacency\_list):

self.adjacency\_list = adjacency\_list

def get\_neighbors(self, v):

return self.adjacency\_list[v]

def h(self, n):

H = {'A': 1,'B': 1,'C': 1,'D': 1}

return H[n]

def a\_star\_algorithm(self, start\_node, stop\_node):

open\_list = set([start\_node])

closed\_list = set([])

g = {}

g[start\_node] = 0

parents = {}

parents[start\_node] = start\_node

while len(open\_list) > 0:

n = None

for v in open\_list:

if n == None or g[v] + self.h(v) < g[n] + self.h(n):

n = v;

if n == None:

print('Path does not exist!')

return None

if n == stop\_node:

reconst\_path = []

while parents[n] != n:

reconst\_path.append(n)

n = parents[n]

reconst\_path.append(start\_node)

reconst\_path.reverse()

print('Path found: {}'.format(reconst\_path))

return reconst\_path

for (m, weight) in self.get\_neighbors(n):

if m not in open\_list and m not in closed\_list:

open\_list.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n

if m in closed\_list:

closed\_list.remove(m)

open\_list.add(m)

open\_list.remove(n)

closed\_list.add(n)

print('Path does not exist!')

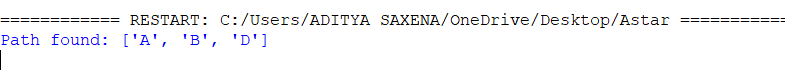
return None

adjacency\_list = {'A': [('B', 1), ('C', 3), ('D', 7)],'B': [('D', 5)],'C': [('D', 12)]}

graph1 = Graph(adjacency\_list)

graph1.a\_star\_algorithm('A', 'D')

**Output:-**

****

**Result-** Hence, implementation of best first search algorithm and A\* algorithm has been done successfully.

**Lab-6**

**AIM:-** Write a python program for implementation of minimax algorithm for an application.

**Minimax Algorithm**

**Code:-**

import math

def minimax (curDepth, nodeIndex,maxTurn, scores,targetDepth):

if (curDepth == targetDepth):

return scores[nodeIndex]

if (maxTurn):

return max(minimax(curDepth + 1, nodeIndex \* 2,

False, scores, targetDepth),

minimax(curDepth + 1, nodeIndex \* 2 + 1,

False, scores, targetDepth))

else:

return min(minimax(curDepth + 1, nodeIndex \* 2,

True, scores, targetDepth),

minimax(curDepth + 1, nodeIndex \* 2 + 1,

True, scores, targetDepth))

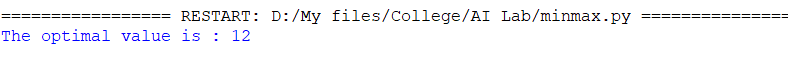
scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")

print(minimax(0, 0, True, scores, treeDepth))

**Output:-**

****

**Alpha-Beta Pruning**

**Code:-**

MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer,values, alpha, beta):

if depth == 3:

return values[nodeIndex]

if maximizingPlayer:

best = MIN

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)

if beta <= alpha:

break

return best

else:

best = MAX

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)

if beta <= alpha:

break

return best

values = []

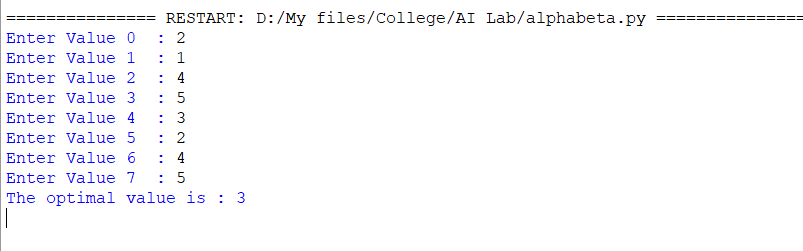
for i in range(0, 8):

x = int(input(f"Enter Value {i} : "))

values.append(x)

print ("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))

**Output:-**

****

**Result:-**Hence, minimax algorithm for an application has been implemented successfully.

**Lab-7**

**AIM:** Write a python program to implement unification and resolution for a given set of statements.

**Unification**

**Code:-**

def get\_index\_comma(string):

index\_list = list()

par\_count = 0

for i in range(len(string)):

if string[i] == ',' and par\_count == 0:

index\_list.append(i)

elif string[i] == '(':

par\_count += 1

elif string[i] == ')':

par\_count -= 1

return index\_list

def is\_variable(expr):

for i in expr:

if i == '(' or i == ')':

return False

return True

def process\_expression(expr):

expr = expr.replace(' ', '')

index = None

for i in range(len(expr)):

if expr[i] == '(':

index = i

break

predicate\_symbol = expr[:index]

expr = expr.replace(predicate\_symbol, '')

expr = expr[1:len(expr) - 1]

arg\_list = list()

indices = get\_index\_comma(expr)

if len(indices) == 0:

arg\_list.append(expr)

else:

arg\_list.append(expr[:indices[0]])

for i, j in zip(indices, indices[1:]):

arg\_list.append(expr[i + 1:j])

arg\_list.append(expr[indices[len(indices) - 1] + 1:])

return predicate\_symbol, arg\_list

def get\_arg\_list(expr):

\_, arg\_list = process\_expression(expr)

flag = True

while flag:

flag = False

for i in arg\_list:

if not is\_variable(i):

flag = True

\_, tmp = process\_expression(i)

for j in tmp:

if j not in arg\_list:

arg\_list.append(j)

arg\_list.remove(i)

return arg\_list

def check\_occurs(var, expr):

arg\_list = get\_arg\_list(expr)

if var in arg\_list:

return True

return False

def unify(expr1, expr2):

if is\_variable(expr1) and is\_variable(expr2):

if expr1 == expr2:

return 'Null'

else:

return False

elif is\_variable(expr1) and not is\_variable(expr2):

if check\_occurs(expr1, expr2):

return False

else:

tmp = str(expr2) + '/' + str(expr1)

return tmp

elif not is\_variable(expr1) and is\_variable(expr2):

if check\_occurs(expr2, expr1):

return False

else:

tmp = str(expr1) + '/' + str(expr2)

return tmp

else:

predicate\_symbol\_1, arg\_list\_1 = process\_expression(expr1)

predicate\_symbol\_2, arg\_list\_2 = process\_expression(expr2)

if predicate\_symbol\_1 != predicate\_symbol\_2:

return False

elif len(arg\_list\_1) != len(arg\_list\_2):

return False

else:

sub\_list = list()

for i in range(len(arg\_list\_1)):

tmp = unify(arg\_list\_1[i], arg\_list\_2[i])

if not tmp:

return False

elif tmp == 'Null':

pass

else:

if type(tmp) == list:

for j in tmp:

sub\_list.append(j)

else:

sub\_list.append(tmp)

return sub\_list

f1 = 'Q(a, g(x, a), f(y))'

f2 = 'Q(a, g(f(b), a), x)'

result = unify(f1, f2)

if not result:

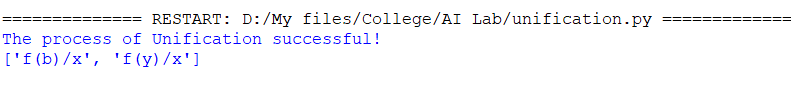
print('The process of Unification failed!')

else:

print('The process of Unification successful!')

print(result)

**Output:-**

****

**Resolution**

**Code:-**

import copy

import time

class Parameter:

variable\_count = 1

def \_\_init\_\_(self, name=None):

if name:

self.type = "Constant"

self.name = name

else:

self.type = "Variable"

self.name = "v" + str(Parameter.variable\_count)

Parameter.variable\_count += 1

def isConstant(self):

return self.type == "Constant"

def unify(self, type\_, name):

self.type = type\_

self.name = name

def \_\_eq\_\_(self, other):

return self.name == other.name

def \_\_str\_\_(self):

return self.name

class Predicate:

def \_\_init\_\_(self, name, params):

self.name = name

self.params = params

def \_\_eq\_\_(self, other):

return self.name == other.name and all(a == b for a, b in zip(self.params, other.params))

def \_\_str\_\_(self):

return self.name + "(" + ",".join(str(x) for x in self.params) + ")"

def getNegatedPredicate(self):

return Predicate(negatePredicate(self.name), self.params)

class Sentence:

sentence\_count = 0

def \_\_init\_\_(self, string):

self.sentence\_index = Sentence.sentence\_count

Sentence.sentence\_count += 1

self.predicates = []

self.variable\_map = {}

local = {}

for predicate in string.split("|"):

name = predicate[:predicate.find("(")]

params = []

for param in predicate[predicate.find("(") + 1: predicate.find(")")].split(","):

if param[0].islower():

if param not in local: # Variable

local[param] = Parameter()

self.variable\_map[local[param].name] = local[param]

new\_param = local[param]

else:

new\_param = Parameter(param)

self.variable\_map[param] = new\_param

params.append(new\_param)

self.predicates.append(Predicate(name, params))

def getPredicates(self):

return [predicate.name for predicate in self.predicates]

def findPredicates(self, name):

return [predicate for predicate in self.predicates if predicate.name == name]

def removePredicate(self, predicate):

self.predicates.remove(predicate)

for key, val in self.variable\_map.items():

if not val:

self.variable\_map.pop(key)

def containsVariable(self):

return any(not param.isConstant() for param in self.variable\_map.values())

def \_\_eq\_\_(self, other):

if len(self.predicates) == 1 and self.predicates[0] == other:

return True

return False

def \_\_str\_\_(self):

return "".join([str(predicate) for predicate in self.predicates])

class KB:

def \_\_init\_\_(self, inputSentences):

self.inputSentences = [x.replace(" ", "") for x in inputSentences]

self.sentences = []

self.sentence\_map = {}

def prepareKB(self):

self.convertSentencesToCNF()

for sentence\_string in self.inputSentences:

sentence = Sentence(sentence\_string)

for predicate in sentence.getPredicates():

self.sentence\_map[predicate] = self.sentence\_map.get(

predicate, []) + [sentence]

def convertSentencesToCNF(self):

for sentenceIdx in range(len(self.inputSentences)):

# Do negation of the Premise and add them as literal

if "=>" in self.inputSentences[sentenceIdx]:

self.inputSentences[sentenceIdx] = negateAntecedent(

self.inputSentences[sentenceIdx])

def askQueries(self, queryList):

results = []

for query in queryList:

negatedQuery = Sentence(negatePredicate(query.replace(" ", "")))

negatedPredicate = negatedQuery.predicates[0]

prev\_sentence\_map = copy.deepcopy(self.sentence\_map)

self.sentence\_map[negatedPredicate.name] = self.sentence\_map.get(

negatedPredicate.name, []) + [negatedQuery]

self.timeLimit = time.time() + 40

try:

result = self.resolve([negatedPredicate], [False]\*(len(self.inputSentences) + 1))

except:

result = False

self.sentence\_map = prev\_sentence\_map

if result:

results.append("TRUE")

else:

results.append("FALSE")

return results

def resolve(self, queryStack, visited, depth=0):

if time.time() > self.timeLimit:

raise Exception

if queryStack:

query = queryStack.pop(-1)

negatedQuery = query.getNegatedPredicate()

queryPredicateName = negatedQuery.name

if queryPredicateName not in self.sentence\_map:

return False

else:

queryPredicate = negatedQuery

for kb\_sentence in self.sentence\_map[queryPredicateName]:

if not visited[kb\_sentence.sentence\_index]:

for kbPredicate in kb\_sentence.findPredicates(queryPredicateName):

canUnify, substitution = performUnification(

copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))

if canUnify:

newSentence = copy.deepcopy(kb\_sentence)

newSentence.removePredicate(kbPredicate)

newQueryStack = copy.deepcopy(queryStack)

if substitution:

for old, new in substitution.items():

if old in newSentence.variable\_map:

parameter = newSentence.variable\_map[old]

newSentence.variable\_map.pop(old)

parameter.unify(

"Variable" if new[0].islower() else "Constant", new)

newSentence.variable\_map[new] = parameter

for predicate in newQueryStack:

for index, param in enumerate(predicate.params):

if param.name in substitution:

new = substitution[param.name]

predicate.params[index].unify(

"Variable" if new[0].islower() else "Constant", new)

for predicate in newSentence.predicates:

newQueryStack.append(predicate)

new\_visited = copy.deepcopy(visited)

if kb\_sentence.containsVariable() and len(kb\_sentence.predicates) > 1:

new\_visited[kb\_sentence.sentence\_index] = True

if self.resolve(newQueryStack, new\_visited, depth + 1):

return True

return False

return True

def performUnification(queryPredicate, kbPredicate):

substitution = {}

if queryPredicate == kbPredicate:

return True, {}

else:

for query, kb in zip(queryPredicate.params, kbPredicate.params):

if query == kb:

continue

if kb.isConstant():

if not query.isConstant():

if query.name not in substitution:

substitution[query.name] = kb.name

elif substitution[query.name] != kb.name:

return False, {}

query.unify("Constant", kb.name)

else:

return False, {}

else:

if not query.isConstant():

if kb.name not in substitution:

substitution[kb.name] = query.name

elif substitution[kb.name] != query.name:

return False, {}

kb.unify("Variable", query.name)

else:

if kb.name not in substitution:

substitution[kb.name] = query.name

elif substitution[kb.name] != query.name:

return False, {}

return True, substitution

def negatePredicate(predicate):

return predicate[1:] if predicate[0] == "~" else "~" + predicate

def negateAntecedent(sentence):

antecedent = sentence[:sentence.find("=>")]

premise = []

for predicate in antecedent.split("&"):

premise.append(negatePredicate(predicate))

premise.append(sentence[sentence.find("=>") + 2:])

return "|".join(premise)

def getInput(filename):

with open(filename, "r") as file:

noOfQueries = int(file.readline().strip())

inputQueries = [file.readline().strip() for \_ in range(noOfQueries)]

noOfSentences = int(file.readline().strip())

inputSentences = [file.readline().strip()

for \_ in range(noOfSentences)]

return inputQueries, inputSentences

def printOutput(filename, results):

print(results)

with open(filename, "w") as file:

for line in results:

file.write(line)

file.write("\n")

file.close()

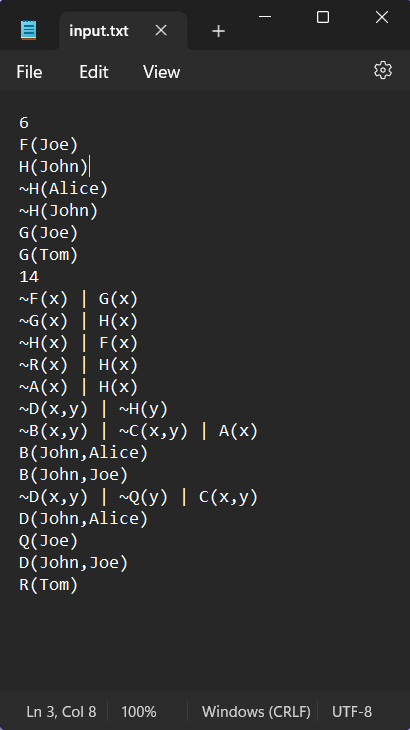
inputQueries\_, inputSentences\_ = getInput('input.txt')

knowledgeBase = KB(inputSentences\_)

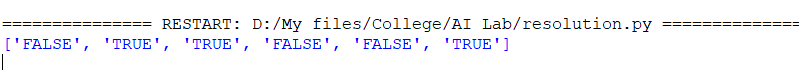
knowledgeBase.prepareKB()

results\_ = knowledgeBase.askQueries(inputQueries\_)

printOutput("output.txt", results\_)



**Output:-**

****

**Result:** Hence unification and resolution have been implemented successfully.

**Lab-8**

**AIM:** Write a python program to implement the knowledge representation schemes using test cases.

**Code:-**

import sys

def definiteNoun(s):

s = s.lower().strip()

if s in ['a', 'e', 'i', 'o', 'u', 'y']:

return "an " + s

else:

return "a " + s

def removeArticle(s):

s = s.lower().strip()

if s[0:3] == "an ": return s[3:]

if s[0:2] == "a ": return s[2:]

return s

def makeQuestion(question, yes, no):

return [question, yes, no]

def isQuestion(p):

return type(p).\_\_name\_\_ == "list"

def askQuestion(question):

print ("\r%s " % question,)

return sys.stdin.readline().strip().lower()

def getAnswer(question):

if isQuestion(question):

return askQuestion(question[0])

else:

return askQuestion("Were you thinking about %s?" % definiteNoun(question))

def answeredYes(answer):

if len(answer) > 0:

return answer.lower()[0] == "y"

return False

def gameOver(message):

global tries

print ("")

print ("\r%s" % message)

print ("")

def playAgain():

return answeredYes(askQuestion("Do you want to play again?"))

def correctGuess(message):

global tries

gameOver(message)

if playAgain():

print ("")

tries = 0

return Q

else:

sys.exit(0)

def nextQuestion(question, answer):

global tries

tries += 1

if isQuestion(question):

if answer:

return question[1]

else:

return question[2]

else:

if answer:

return correctGuess("I knew it!")

else:

return makeNewQuestion(question)

def replaceAnswer(tree, find, replace):

if not isQuestion(tree):

if tree == find:

return replace

else:

return tree

else:

return makeQuestion(tree[0],

replaceAnswer(tree[1], find, replace),

replaceAnswer(tree[2], find, replace))

def makeNewQuestion(wrongAnimal):

global Q, tries

correctAnimal = removeArticle(askQuestion("I give up. What did you think about?"))

newQuestion = askQuestion("Enter a question that would distinguish %s from %s:"

% (definiteNoun(correctAnimal), definiteNoun(wrongAnimal))).capitalize()

yesAnswer = answeredYes(askQuestion("If I asked you this question " +

"and you thought about %s, what would the correct answer be?" % definiteNoun(correctAnimal)))

# Create new question node

if yesAnswer:

q = makeQuestion(newQuestion, correctAnimal, wrongAnimal)

else:

q = makeQuestion(newQuestion, wrongAnimal, correctAnimal)

Q = replaceAnswer(Q, wrongAnimal, q)

tries = 0

return Q

def addNewQuestion(wrongAnimal, newques, correct):

global Q

q = makeQuestion(newques, correct, wrongAnimal)

Q = replaceAnswer(Q, wrongAnimal, q)

return Q

tries = 0

Q = (makeQuestion('Does it have fur?', 'Tiger', 'Penguin'))

q = addNewQuestion('Tiger', 'Does it have dark spots?', 'Leopard')

q = addNewQuestion('Leopard', 'Is it the fastest animal?', 'Cheetah')

q = addNewQuestion('Penguin', 'Can it fly?', 'Parrot')

q = Q

print ("Imagine an animal. I will try to guess which one.")

print ("You are only allowed to answer YES or NO.")

print ("")

try:

while True:

ans = answeredYes(getAnswer(q))

q = nextQuestion(q, ans)

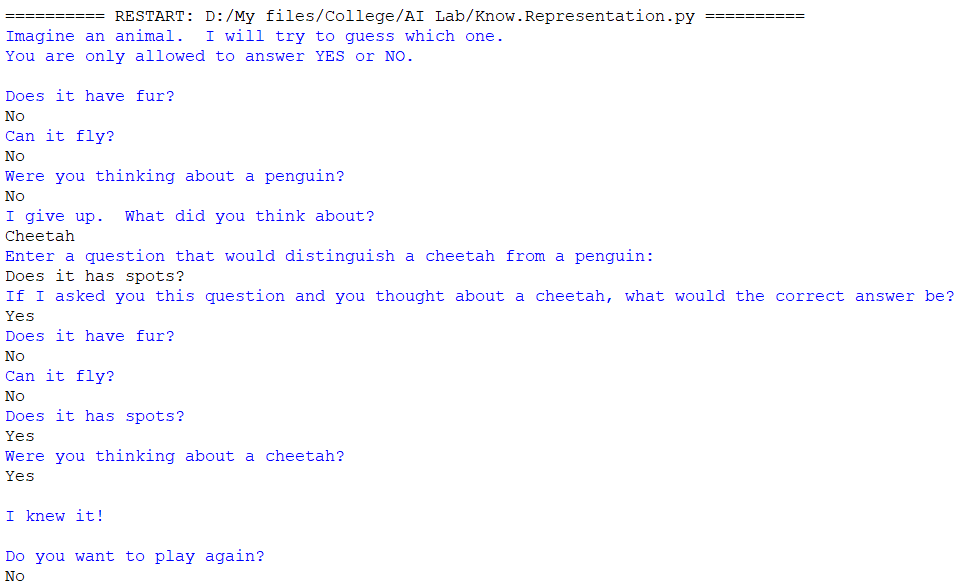
except KeyboardInterrupt:

sys.exit(0)

except Exception:

sys.exit(1)

**Output:-**



**Result:** We have successfully implemented knowledge representation schemes – use cases and output.

**Lab-9**

**AIM:** Write a python program to implement uncertain methods for an application.

(Implementing Fuzzy logic using matplotlib in python and find the graph of temperature, humidity and speed in different conditions.)

**Code:-**

import numpy as np

import skfuzzy as fuzz

from skfuzzy import control as ctrl

import matplotlib.pyplot as plt

temp = ctrl.Antecedent(np.arange(0, 101, 1), 'Temperature')

humidity = ctrl.Antecedent(np.arange(0, 101, 1), 'Humidity')

speed = ctrl.Consequent(np.arange(0, 101, 1), 'Speed')

temp['cold'] = fuzz.trimf(temp.universe, [0, 0, 50])

temp['hot'] = fuzz.trimf(temp.universe, [50, 100, 100])

humidity['dry'] = fuzz.trimf(humidity.universe, [0, 0, 50])

humidity['wet'] = fuzz.trimf(humidity.universe, [50, 100, 100])

speed['slow'] = fuzz.trimf(speed.universe, [0, 0, 50])

speed['fast'] = fuzz.trimf(speed.universe, [50, 100, 100])

rule1 = ctrl.Rule(temp['cold'] | humidity['dry'], speed['slow'])

rule2 = ctrl.Rule(temp['hot'] | humidity['wet'], speed['fast'])

rule3 = ctrl.Rule(humidity['dry'] & temp['hot'], speed['fast'])

rule4 = ctrl.Rule(humidity['wet'] & temp['cold'], speed['slow'])

speed\_ctrl = ctrl.ControlSystem([rule1, rule2, rule3, rule4])

speed\_simulation = ctrl.ControlSystemSimulation(speed\_ctrl)

speed\_simulation.input['Temperature'] = 30

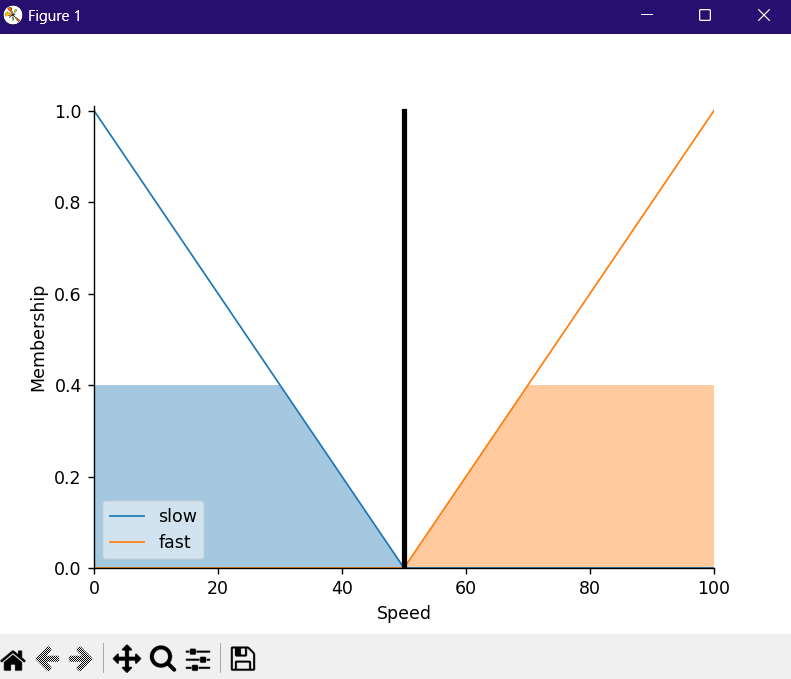
speed\_simulation.input['Humidity'] = 70

speed\_simulation.compute()

speed.view(sim=speed\_simulation)

plt.show()

**Output:-**

****

**Result:** We have successfully implemented fuzzy uncertainty problem using matplotlib and output is received.

**Lab-10**

**AIM:** Write a python program to implement the block world problem using correct artificial intelligence optimization techniques.

**Code:-**

import time

class Node:

def \_\_init\_\_(self, data=None):

self.data = data

self.next = None

class Stack:

def \_\_init\_\_(self):

print("Stack created")

self.stack\_pointer = None

def push(self, x):

if not isinstance(x, Node):

x = Node(x)

print(f"Adding {x.data} to the top of stack")

if self.is\_empty():

self.stack\_pointer = x

else:

x.next = self.stack\_pointer

self.stack\_pointer = x

def pop(self):

if not self.is\_empty():

print(f"Removing node on top of stack")

curr = self.stack\_pointer

self.stack\_pointer = self.stack\_pointer.next

curr.next = None

return curr.data

else:

return "Stack is empty"

def is\_empty(self):

return self.stack\_pointer == None

def peek(self):

if not self.is\_empty():

return self.stack\_pointer.data

def \_\_str\_\_(self):

print("Printing Stack state...")

to\_print = ""

curr = self.stack\_pointer

while curr is not None:

to\_print += str(curr.data) + "->"

curr = curr.next

if to\_print:

print("Stack Pointer")

print(" |")

print(" V")

return "[" + to\_print[:-2] + "]"

return "[]"

print ("INITIAL STATE : {[1], [2], [3], [4], [5]}")

print("-"\*70)

print ("FINAL STATE :[4->3->2->1]")

my\_stack = Stack()

print("Checking if stack is empty:", my\_stack.is\_empty())

my\_stack.push(1)

time.sleep(1)

my\_stack.push(2)

print(my\_stack)

time.sleep(1)

my\_stack.push(3)

time.sleep(1)

my\_stack.push(4)

time.sleep(1)

print("Checking item on top of stack:", my\_stack.peek())

time.sleep(1)

my\_stack.push(5)

print(my\_stack)

time.sleep(1)

print(my\_stack.pop())

time.sleep(1)

print(my\_stack.pop())

print(my\_stack)

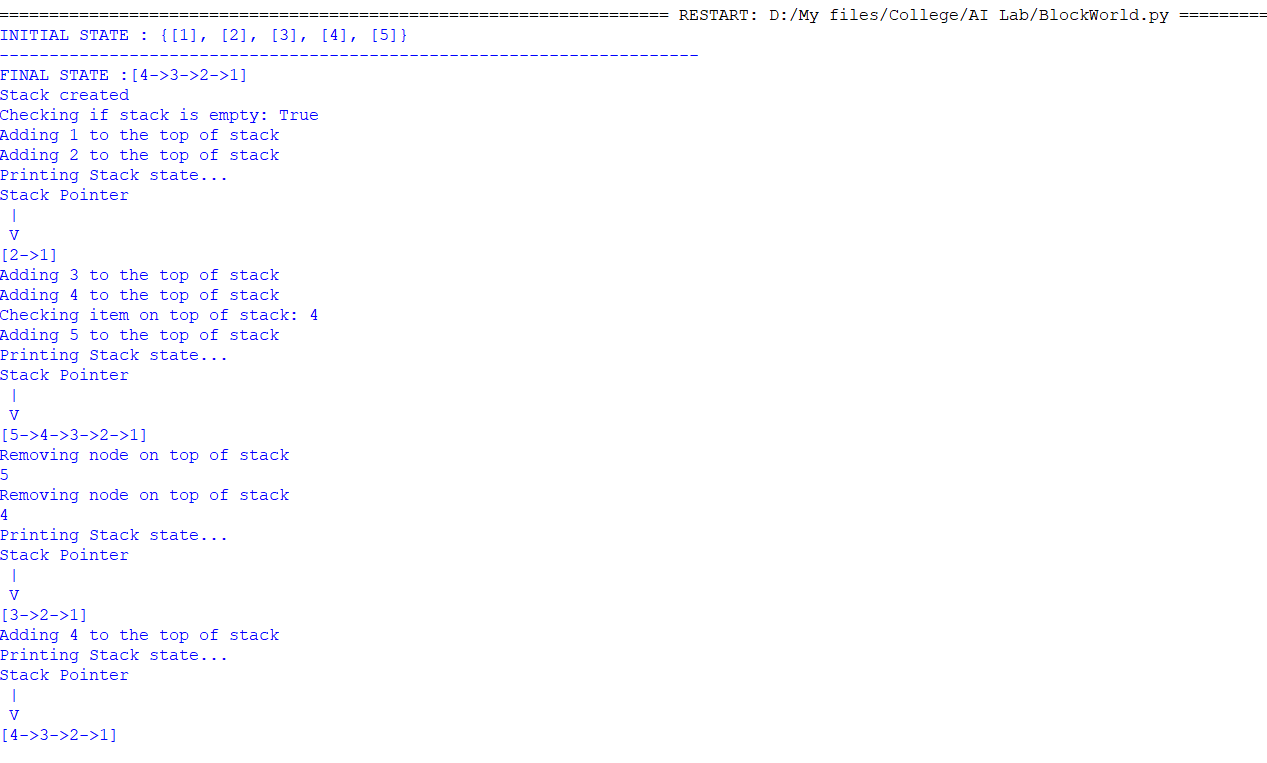
time.sleep(1)

my\_stack.push(4)

print(my\_stack)

time.sleep(1)

**Output:-**

****

**Result:** We have successfully implemented block world problem using stacks and searching algorithm.

**Lab-11**

**AIM:** Write a python program to implement a learning algorithm for an application.

The learning models used are:-

1.Logistic Regression

2.SVM

3.Naive Bayes

Logistic Regression

Logistic regression predicts the output of a categorical dependent variable. Therefore, the outcome must be a categorical or discrete value. It can be either Yes or No, 0 or 1, True or False, etc. but instead of giving the exact value as 0 and 1, it gives the probabilistic values which lie between 0 and 1.

**Code:-**

from sklearn.datasets import load\_iris

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

iris = load\_iris()

X = iris.data

Y = iris.target

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=42)

lr = LogisticRegression()

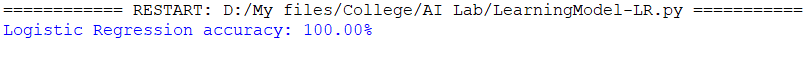
lr.fit(X\_train, Y\_train)

Y\_pred\_lr = lr.predict(X\_test)

accuracy\_lr = accuracy\_score(Y\_test, Y\_pred\_lr)

print("Logistic Regression accuracy: {:.2f}%".format(accuracy\_lr \* 100))

**Output:-**



SVM (Support Vector Machine)

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

**Code:-**

from sklearn.datasets import load\_iris

from sklearn.svm import SVC

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

iris = load\_iris()

X = iris.data

Y = iris.target

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=42)

svm = SVC(kernel='linear')

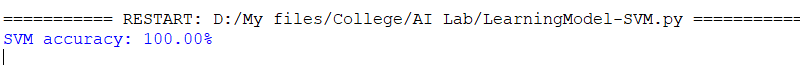
svm.fit(X\_train, Y\_train)

Y\_pred\_svm = svm.predict(X\_test)

accuracy\_svm = accuracy\_score(Y\_test, Y\_pred\_svm)

print("SVM accuracy: {:.2f}%".format(accuracy\_svm \* 100))

**Output:-**



Naive Bayes

Naive Bayes algorithm is a supervised learning algorithm, which is based on Bayes theorem and used for solving classification problems. It is mainly used in text classification that includes a high-dimensional training dataset.

**Code:-**

from sklearn.datasets import load\_iris

from sklearn.naive\_bayes import GaussianNB

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

iris = load\_iris()

X = iris.data

Y = iris.target

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=42)

nb = GaussianNB()

nb.fit(X\_train, Y\_train)

Y\_pred\_nb = nb.predict(X\_test)

accuracy\_nb = accuracy\_score(Y\_test, Y\_pred\_nb)

print("Naive Bayes accuracy: {:.2f}%".format(accuracy\_nb \* 100))

**Output:-**

****

**Result:** We have successfully implemented leaning algorithms.

**Lab-12**

**AIM:** Write a python program to develop an ensemble model for an application.

**Code:-**

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import VotingClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.linear\_model import LogisticRegression

from sklearn.neighbors import KNeighborsClassifier

data = pd.DataFrame({

'feature1': [0, 1, 0, 1, 0, 1, 1, 0],

'feature2': [0, 1, 1, 1, 0, 1, 0, 0],

'target': [0, 1, 1, 1, 0, 1, 0, 0]

})

X\_train, X\_test, y\_train, y\_test = train\_test\_split(data.iloc[:, :-1], data.iloc[:, -1], test\_size=0.3)

model1 = DecisionTreeClassifier()

model2 = LogisticRegression()

model3 = KNeighborsClassifier()

#ensemble model

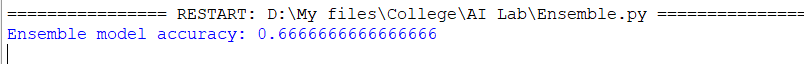
ensemble\_model = VotingClassifier(estimators=[('dt', model1), ('lr', model2), ('knn', model3)], voting='hard')

ensemble\_model.fit(X\_train, y\_train)

accuracy = ensemble\_model.score(X\_test, y\_test)

print(f"Ensemble model accuracy: {accuracy}")

**Output:-**



**Result:** We have successfully developed an ensemble model for an application.

**Lab-13**

**AIM:** Write a python program to implement an NLP program.

**Code:-**

import nltk

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

nltk.download('punkt')

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

nltk.download('averaged\_perceptron\_tagger')

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

text = input("Enter a sentence: ")

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

tokens = nltk.word\_tokenize(text)

tagged = nltk.pos\_tag(tokens)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

for word, tag in tagged:

if tag.startswith('N'):

print(word, '--> noun')

elif tag.startswith('V'):

print(word, '--> verb')

elif tag.startswith('J'):

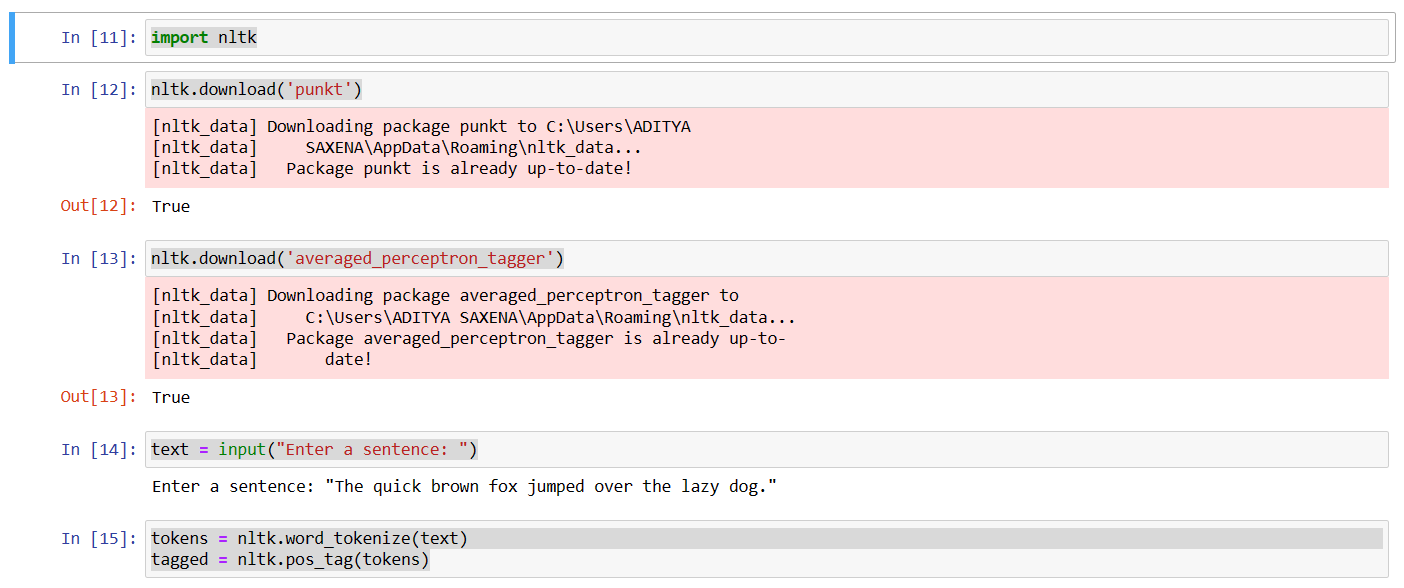
print(word, '--> adjective')

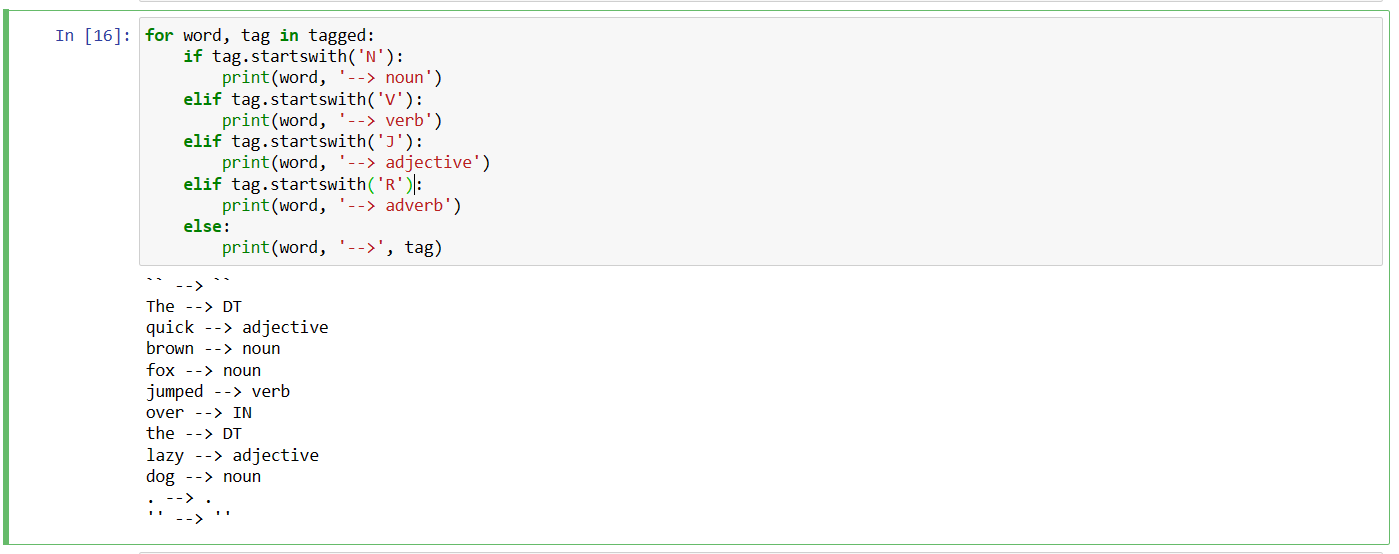
elif tag.startswith('R'):

print(word, '--> adverb')

else:

print(word, '-->', tag)

**Output:-**



**Result :** We have successfully implemented a NLP program.

**Lab-14**

**Aim:** To apply deep learning to solve a problem.

**Code:-**

#K Nearest Neighbors

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score

from sklearn.datasets import load\_digits

from sklearn.model\_selection import train\_test\_split

digits = load\_digits()

X, Y = digits.data, digits.target

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.3, random\_state=42)

knn = KNeighborsClassifier(n\_neighbors=7)

knn.fit(X\_train, Y\_train)

Y\_pred\_knn = knn.predict(X\_test)

score\_knn = round(accuracy\_score(Y\_pred\_knn,Y\_test)\*100,2)

print("The accuracy score achieved using K-Nearest Neighbors is: "+str(score\_knn)+" %")

#Decision Tree

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

from sklearn.datasets import load\_digits

from sklearn.model\_selection import train\_test\_split

digits = load\_digits()

X, Y = digits.data, digits.target

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.3, random\_state=42)

dt = DecisionTreeClassifier()

dt.fit(X\_train, Y\_train)

Y\_pred\_dt = dt.predict(X\_test)

score\_dt = round(accuracy\_score(Y\_pred\_dt,Y\_test)\*100,2)

print("The accuracy score achieved using Decision Tree is: "+str(score\_dt)+" %")

**Output:-**

****

****

**Result :** Hence, deep learning methods have been implemented and compared to find the best for precision decision making.