1 Write the python program to solve 8-Puzzle problem

import copy

from heapq import heappush, heappop

n = 3

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

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

class priorityQueue:

    def \_\_init\_\_(self):

        self.heap = []

    def push(self, k):

        heappush(self.heap, k)

    def pop(self):

        return heappop(self.heap)

    def empty(self):

        if not self.heap:

            return True

        else:

            return False

class node:

    def \_\_init\_\_(self, parent, mat, empty\_tile\_pos,

                cost, level):

        self.parent = parent

        self.mat = mat

        self.empty\_tile\_pos = empty\_tile\_pos

        self.cost = cost

        self.level = level

    def \_\_lt\_\_(self, nxt):

        return self.cost < nxt.cost

def calculateCost(mat, final) -> int:

    count = 0

    for i in range(n):

        for j in range(n):

            if ((mat[i][j]) and

                (mat[i][j] != final[i][j])):

                count += 1

    return count

def newNode(mat, empty\_tile\_pos, new\_empty\_tile\_pos,

            level, parent, final) -> node:

    new\_mat = copy.deepcopy(mat)

    x1 = empty\_tile\_pos[0]

    y1 = empty\_tile\_pos[1]

    x2 = new\_empty\_tile\_pos[0]

    y2 = new\_empty\_tile\_pos[1]

    new\_mat[x1][y1], new\_mat[x2][y2] = new\_mat[x2][y2], new\_mat[x1][y1]

    cost = calculateCost(new\_mat, final)

    new\_node = node(parent, new\_mat, new\_empty\_tile\_pos,

                    cost, level)

    return new\_node

def printMatrix(mat):

    for i in range(n):

        for j in range(n):

            print("%d " % (mat[i][j]), end = " ")

        print()

def isSafe(x, y):

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

def printPath(root):

    if root == None:

        return

    printPath(root.parent)

    printMatrix(root.mat)

    print()

def solve(initial, empty\_tile\_pos, final):

    pq = priorityQueue()

    cost = calculateCost(initial, final)

    root = node(None, initial,

                empty\_tile\_pos, cost, 0)

    pq.push(root)

    while not pq.empty():

        minimum = pq.pop()

        if minimum.cost == 0:

            printPath(minimum)

            return

        for i in range(4):

            new\_tile\_pos = [

                minimum.empty\_tile\_pos[0] + row[i],

                minimum.empty\_tile\_pos[1] + col[i], ]

            if isSafe(new\_tile\_pos[0], new\_tile\_pos[1]):

                child = newNode(minimum.mat,

                                minimum.empty\_tile\_pos,

                                new\_tile\_pos,

                                minimum.level + 1,

                                minimum, final,)

                pq.push(child)

initial = [ [ 1, 2, 3 ],

            [ 4, 5, 8 ],

            [ 7, 0, 6 ] ]

final = [ [ 1, 2, 3 ],

        [ 4, 5, 6 ],

        [ 7, 8, 0 ] ]

empty\_tile\_pos = [ 1, 2 ]

solve(initial, empty\_tile\_pos, final)

2 Write the python program to solve 8-Queen problem

def is\_safe(board, row, col):

    # Check if there's a queen in the same column

    for i in range(row):

        if board[i][col] == 1:

            return False

    # Check upper left diagonal

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

        if board[i][j] == 1:

            return False

    # Check upper right diagonal

    for i, j in zip(range(row, -1, -1), range(col, len(board))):

        if board[i][j] == 1:

            return False

    return True

def solve\_nqueens(board, row):

    if row >= len(board):

        return True

    for col in range(len(board)):

        if is\_safe(board, row, col):

            board[row][col] = 1

            if solve\_nqueens(board, row + 1):

                return True

            board[row][col] = 0

    return False

def print\_board(board):

    for row in board:

        print(" ".join("Q" if cell == 1 else "-" for cell in row))

def solve\_8queens():

    board = [[0 for \_ in range(8)] for \_ in range(8)]

    if solve\_nqueens(board, 0):

        print\_board(board)

    else:

        print("No solution found.")

solve\_8queens()

3 Write the python program for Water Jug Problem

from collections import defaultdict

jug1, jug2, aim = 4, 3, 2

visited = defaultdict(lambda: False)

def waterJugSolver(amt1, amt2):

    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):

        print(amt1, amt2)

        return True

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

        print(amt1, amt2)

        # Changes the boolean value of

        # the combination as it is visited.

        visited[(amt1, amt2)] = True

        return (waterJugSolver(0, amt2) or

                waterJugSolver(amt1, 0) or

                waterJugSolver(jug1, amt2) or

                waterJugSolver(amt1, jug2) or

                waterJugSolver(amt1 + min(amt2, (jug1-amt1)),

                amt2 - min(amt2, (jug1-amt1))) or

                waterJugSolver(amt1 - min(amt1, (jug2-amt2)),

                amt2 + min(amt1, (jug2-amt2))))

    else:

        return False

print("Steps: ")

waterJugSolver(0, 0)

4 Write the python program for Cript-Arithmetic problem

from itertools import permutations

def is\_solution\_valid(mapping, words, result):

    values = []

    for word in words + [result]:

        value = 0

        for char in word:

            value = value \* 10 + mapping[char]

        values.append(value)

    return sum(values[:-1]) == values[-1]

def solve\_cryptarithmetic(words, result):

    all\_letters = set(''.join(words + [result]))

    if len(all\_letters) > 10:

        return None  # There are more than 10 unique characters, not solvable

    for perm in permutations(range(10), len(all\_letters)):

        mapping = dict(zip(all\_letters, perm))

        if is\_solution\_valid(mapping, words, result):

            return mapping

    return None  # No valid mapping found

# Example: SEND + MORE = MONEY

words = ['SEND', 'MORE']

result = 'MONEY'

solution = solve\_cryptarithmetic(words, result)

if solution:

    print("Solution found:")

    for char, digit in solution.items():

        print(f"{char}: {digit}")

else:

    print("No solution found.")

5 Write the python program for Missionaries Cannibal problem

def missionaries\_and\_cannibals(l\_m, l\_c, r\_m, r\_c):

  """

  Solves the missionaries and cannibals problem using a recursive backtracking algorithm.

  Args:

    l\_m: The number of missionaries on the left bank.

    l\_c: The number of cannibals on the left bank.

    r\_m: The number of missionaries on the right bank.

    r\_c: The number of cannibals on the right bank.

  Returns:

    True if the problem is solved, False otherwise.

  """

  print("Left Bank:", l\_m, l\_c, "Right Bank:", r\_m, r\_c)

  if (l\_m == 0 and l\_c == 0):

    return True

  # Check if the current state is unsafe.

  if (l\_c > l\_m):

    return False

  # Try moving 2 missionaries and 1 cannibal to the right bank.

  if (l\_m >= 2 and l\_c >= 1):

    print("Move 2 m and 1 c to the right bank.")

    if missionaries\_and\_cannibals(l\_m - 2, l\_c - 1, r\_m + 2, r\_c):

      return True

  # Try moving 1 missionary and 1 cannibal to the right bank.

  if (l\_m >= 1 and l\_c >= 1):

    print("Move 1 m and 1 c to the right bank.")

    if missionaries\_and\_cannibals(l\_m - 1, l\_c - 1, r\_m + 1, r\_c):

      return True

  # Try moving 2 cannibals to the right bank.

  if (l\_c >= 2):

    print("Move 2 c to the right bank.")

    if missionaries\_and\_cannibals(l\_m, l\_c - 2, r\_m, r\_c + 2):

      return True

  return False

def main():

  """

  Solves the missionaries and cannibals problem.

  """

  l\_m = 3

  l\_c = 3

  r\_m = 0

  r\_c = 0

  if missionaries\_and\_cannibals(l\_m, l\_c, r\_m, r\_c):

    print("The problem is solved.")

  else:

    print("The problem cannot be solved.")

if \_\_name\_\_ == "\_\_main\_\_":

  main()

6 Write the python program for Vacuum Cleaner problem

#Author A.Safarji

import random

def display(room):

    print(room)

room = [

    [1, 1, 1, 1],

    [1, 1, 1, 1],

    [1, 1, 1, 1],

    [1, 1, 1, 1],

]

print("All the rooom are dirty")

display(room)

x =0

y= 0

while x < 4:

    while y < 4:

        room[x][y] = random.choice([0,1])

        y+=1

    x+=1

    y=0

print("Before cleaning the room I detect all of these random dirts")

display(room)

x =0

y= 0

z=0

while x < 4:

    while y < 4:

        if room[x][y] == 1:

            print("Vaccum in this location now,",x, y)

            room[x][y] = 0

            print("cleaned", x, y)

            z+=1

        y+=1

    x+=1

    y=0

pro= (100-((z/16)\*100))

print("Room is clean now, Thanks for using : 3710933")

display(room)

print('performance=',pro,'%')

7 Write the python program to implement BFS.

from collections import defaultdict, deque

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(list)

    def add\_edge(self, u, v):

        self.graph[u].append(v)

    def bfs(self, start):

        visited = set()

        queue = deque([start])

        visited.add(start)

        while queue:

            vertex = queue.popleft()

            print(vertex, end=" ")

            for neighbor in self.graph[vertex]:

                if neighbor not in visited:

                    queue.append(neighbor)

                    visited.add(neighbor)

# Create a graph

graph = Graph()

graph.add\_edge(0, 1)

graph.add\_edge(0, 2)

graph.add\_edge(1, 2)

graph.add\_edge(2, 0)

graph.add\_edge(2, 3)

graph.add\_edge(3, 3)

print("Breadth-First Traversal (starting from vertex 2):")

graph.bfs(2)

8 Write the python program to implement DFS.

def dfs(graph, start):

    visited = [False] \* len(graph)

    stack = [start]

    traversal\_order = []

    total\_cost = 0

    while stack:

        vertex = stack.pop()

        if not visited[vertex]:

            visited[vertex] = True

            traversal\_order.append(vertex)

            for neighbor, cost in enumerate(graph[vertex]):

                if cost > 0 and not visited[neighbor]:

                    stack.append(neighbor)

                    total\_cost += cost

    return traversal\_order, total\_cost

# Example graph represented by an adjacency matrix

graph = [

    [0, 10, 15, 20],

    [10, 0, 35, 25],

    [15, 35, 0, 30],

    [20, 25, 30, 0]

]

start\_vertex = 0  # Starting vertex for DFS

traversal\_order, total\_cost = dfs(graph, start\_vertex)

# Print traversal order and total cost

print("Traversal Order:", traversal\_order)

print("Total Cost:", total\_cost)

# Visualization of traversal

print("Graph Traversal:")

for i in range(len(graph)):

    row\_str = " -> ".join(["{:<3}".format(graph[i][j]) for j in range(len(graph[i]))])

    print(row\_str)

# Display traversal path

print("\nTraversal Path:")

for i in range(len(traversal\_order) - 1):

    print(f"{traversal\_order[i]} -> ", end="")

print(traversal\_order[-1])

9 Write the python to implement Travelling Salesman Problem 1

from itertools import permutations

def tsp\_brute\_force(graph):

    nodes = list(graph.keys())

    route\_length = len(nodes)

    minimum\_distance = None

    minimum\_route = None

    for route in permutations(range(1, route\_length)):

        current\_distance = graph[0][route[0]]

        for i in range(route\_length - 2):

            current\_distance += graph[route[i]][route[i+1]]

        current\_distance += graph[route[-1]][0]

        if minimum\_distance is None or current\_distance < minimum\_distance:

            minimum\_distance = current\_distance

            minimum\_route = route

    return minimum\_distance, minimum\_route

# Example graph

graph = {

    0: {0: 0, 1: 10, 2: 15, 3: 20},

    1: {0: 10, 1: 0, 2: 35, 3: 25},

    2: {0: 15, 1: 35, 2: 0, 3: 30},

    3: {0: 20, 1: 25, 2: 30, 3: 0}

}

# Call the tsp\_brute\_force function with the example graph

distance, route = tsp\_brute\_force(graph)

print("Shortest route:", [0] + [x+1 for x in route] + [0])

print("Travel cost:", distance)

10 Write the python program to implement A\* algorithm

import heapq

class Node:

    def \_\_init\_\_(self, vertex, cost, heuristic):

        self.vertex = vertex

        self.cost = cost

        self.heuristic = heuristic

    def \_\_lt\_\_(self, other):

        return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def a\_star(graph, start, goal):

    open\_set = []

    closed\_set = set()

    heapq.heappush(open\_set, Node(start, 0, heuristic[start]))

    parent = {}

    g\_cost = {vertex: float('inf') for vertex in graph}

    g\_cost[start] = 0

    while open\_set:

        current = heapq.heappop(open\_set).vertex

        if current == goal:

            path = []

            while current in parent:

                path.insert(0, current)

                current = parent[current]

            path.insert(0, start)

            return path, g\_cost[goal]

        closed\_set.add(current)

        for neighbor, cost in graph[current].items():

            if neighbor in closed\_set:

                continue

            tentative\_g\_cost = g\_cost[current] + cost

            if tentative\_g\_cost < g\_cost[neighbor]:

                parent[neighbor] = current

                g\_cost[neighbor] = tentative\_g\_cost

                heapq.heappush(open\_set, Node(neighbor, tentative\_g\_cost, heuristic[neighbor]))

    return None, float('inf')

# Example graph represented by a dictionary (adjacency list)

graph = {

    'A': {'B': 10, 'C': 5},

    'B': {'A': 10, 'C': 15, 'D': 7},

    'C': {'A': 5, 'B': 15, 'D': 8},

    'D': {'B': 7, 'C': 8}

}

# Heuristic values for each vertex (manhattan distance to the goal)

heuristic = {

    'A': 15,

    'B': 10,

    'C': 5,

    'D': 0

}

start\_vertex = 'A'

goal\_vertex = 'D'

path, total\_cost = a\_star(graph, start\_vertex, goal\_vertex)

if path:

    print("Shortest Path:", path)

    print("Total Cost:", total\_cost)

else:

    print("No path found from", start\_vertex, "to", goal\_vertex)

11 Write the python program for Map Coloring to implement CSP

def is\_valid\_assignment(assignment, var, color, neighbors):

    for neighbor in neighbors[var]:

        if neighbor in assignment and assignment[neighbor] == color:

            return False

    return True

def backtrack(assignment, variables, colors, neighbors):

    if len(assignment) == len(variables):

        return assignment

    var = select\_unassigned\_variable(assignment, variables)

    for color in colors:

        if is\_valid\_assignment(assignment, var, color, neighbors):

            assignment[var] = color

            result = backtrack(assignment, variables, colors, neighbors)

            if result is not None:

                return result

            del assignment[var]

    return None

def select\_unassigned\_variable(assignment, variables):

    for var in variables:

        if var not in assignment:

            return var

    return None

# Map with regions and their neighbors

regions = {

    'WA': ['NT', 'SA'],

    'NT': ['WA', 'SA', 'Q'],

    'SA': ['WA', 'NT', 'Q', 'NSW', 'V'],

    'Q': ['NT', 'SA', 'NSW'],

    'NSW': ['Q', 'SA', 'V'],

    'V': ['SA', 'NSW']

}

# Available colors

colors = ['R', 'G', 'B']

# Initialize assignment and variables

assignment = {}

variables = list(regions.keys())

solution = backtrack(assignment, variables, colors, regions)

if solution:

    print("Map coloring solution:")

    for region, color in solution.items():

        print(f"{region}: {color}")

else:

    print("No solution found")

. 12 Write the python program for Tic Tac Toe game

def print\_board(board):

    for row in board:

        print(" | ".join(row))

        print("-" \* 9)

def check\_winner(board, player):

    for row in board:

        if all(cell == player for cell in row):

            return True

    for col in range(3):

        if all(board[row][col] == player for row in range(3)):

            return True

    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):

        return True

    return False

def is\_board\_full(board):

    return all(all(cell != " " for cell in row) for row in board)

def main():

    board = [[" " for \_ in range(3)] for \_ in range(3)]

    current\_player = "X"

    while True:

        print\_board(board)

        row = int(input(f"Player {current\_player}, choose row (0-2): "))

        col = int(input(f"Player {current\_player}, choose column (0-2): "))

        if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == " ":

            board[row][col] = current\_player

            if check\_winner(board, current\_player):

                print\_board(board)

                print(f"Player {current\_player} wins!")

                break

            if is\_board\_full(board):

                print\_board(board)

                print("It's a draw!")

                break

            current\_player = "X" if current\_player == "O" else "O"

        else:

            print("Invalid move. Try again.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

13 Write the python program to implement Minimax algorithm for gaming

def evaluate(board):

    # Check rows, columns, and diagonals for a win

    for i in range(3):

        if board[i][0] == board[i][1] == board[i][2]:

            if board[i][0] == 'X':

                return 10

            elif board[i][0] == 'O':

                return -10

        if board[0][i] == board[1][i] == board[2][i]:

            if board[0][i] == 'X':

                return 10

            elif board[0][i] == 'O':

                return -10

    if board[0][0] == board[1][1] == board[2][2]:

        if board[0][0] == 'X':

            return 10

        elif board[0][0] == 'O':

            return -10

    if board[0][2] == board[1][1] == board[2][0]:

        if board[0][2] == 'X':

            return 10

        elif board[0][2] == 'O':

            return -10

    return 0  # No winner

def is\_full(board):

    for i in range(3):

        for j in range(3):

            if board[i][j] == ' ':

                return False

    return True

def minimax(board, depth, is\_maximizing):

    score = evaluate(board)

    if score == 10:

        return score - depth

    if score == -10:

        return score + depth

    if is\_full(board):

        return 0

    if is\_maximizing:

        best\_score = -float('inf')

        for i in range(3):

            for j in range(3):

                if board[i][j] == ' ':

                    board[i][j] = 'X'

                    best\_score = max(best\_score, minimax(board, depth + 1, not is\_maximizing))

                    board[i][j] = ' '

        return best\_score

    else:

        best\_score = float('inf')

        for i in range(3):

            for j in range(3):

                if board[i][j] == ' ':

                    board[i][j] = 'O'

                    best\_score = min(best\_score, minimax(board, depth + 1, not is\_maximizing))

                    board[i][j] = ' '

        return best\_score

def find\_best\_move(board):

    best\_move = (-1, -1)

    best\_score = -float('inf')

    for i in range(3):

        for j in range(3):

            if board[i][j] == ' ':

                board[i][j] = 'X'

                move\_score = minimax(board, 0, False)

                board[i][j] = ' '

                if move\_score > best\_score:

                    best\_score = move\_score

                    best\_move = (i, j)

    return best\_move

def print\_board(board):

    for row in board:

        print(" | ".join(row))

        print("-" \* 9)

def main():

    board = [[' ' for \_ in range(3)] for \_ in range(3)]

    while True:

        print\_board(board)

        if is\_full(board) or evaluate(board) != 0:

            break

        row, col = find\_best\_move(board)

        board[row][col] = 'X'

        print("Computer's move:")

        print\_board(board)

        if is\_full(board) or evaluate(board) != 0:

            break

        row = int(input("Enter row (0, 1, 2): "))

        col = int(input("Enter col (0, 1, 2): "))

        board[row][col] = 'O'

    print\_board(board)

    result = evaluate(board)

    if result == 10:

        print("Computer wins!")

    elif result == -10:

        print("You win!")

    else:

        print("It's a draw!")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

14 Write the python program to implement Apha & Beta pruning algorithm for gaming

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

if \_\_name\_\_ == "\_\_main\_\_":

    values = [3, 5, 6, 9, 1, 2, 0, -1]

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

15 Write the python program to implement Decision Tree

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

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

# Input data with moderate separability

data = [

    [5.1, 3.5, 1.4, 0.2, 0],  # Iris Setosa

    [4.9, 3.0, 1.4, 0.2, 0],  # Iris Setosa

    [6.3, 2.5, 4.9, 1.5, 1],  # Iris Versicolor

    [6.5, 3.0, 5.2, 2.0, 2],  # Iris Virginica

    [5.7, 2.6, 3.5, 1.0, 1],  # Iris Versicolor

    [6.0, 2.2, 5.0, 1.5, 2],  # Iris Virginica

    [5.9, 3.0, 4.2, 1.5, 1],  # Iris Versicolor

    [6.2, 2.8, 4.8, 1.8, 2],  # Iris Virginica

]

# Separate features and target labels

X = [row[:-1] for row in data]

y = [row[-1] for row in data]

# Split data into training and testing sets

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

# Create a Decision Tree Classifier

clf = DecisionTreeClassifier()

# Fit the model to the training data

clf.fit(X\_train, y\_train)

# Make predictions on the test data

y\_pred = clf.predict(X\_test)

# Calculate accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2f}")

# New data point for prediction

new\_data = [5.8, 2.7, 5.1, 1.9]  # You can change this data point

# Make a prediction using the trained model

prediction = clf.predict([new\_data])

# Mapping of target labels

target\_labels = ['Iris Virginica', 'Iris Versicolor', 'Iris Setosa']

# Print the prediction

predicted\_label = target\_labels[prediction[0]]

print(f"Predicted Label: {predicted\_label}")

16 Write the python program to implement Feed forward neural Network

import numpy as np

# Define the sigmoid activation function

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

# Define the derivative of the sigmoid function

def sigmoid\_derivative(x):

    return x \* (1 - x)

# Input data

X = np.array([

    [0, 0],

    [0, 1],

    [1, 0],

    [1, 1]

])

# Target labels

y = np.array([

    [0],

    [1],

    [1],

    [0]

])

# Seed the random number generator for reproducibility

np.random.seed(42)

# Initialize the weights randomly with mean 0

input\_neurons = 2

hidden\_neurons = 4

output\_neurons = 1

# Initialize weights for the first and second layers

weights\_input\_hidden = 2 \* np.random.random((input\_neurons, hidden\_neurons)) - 1

weights\_hidden\_output = 2 \* np.random.random((hidden\_neurons, output\_neurons)) - 1

# Learning rate

learning\_rate = 0.1

# Training

for iteration in range(10000):

    # Forward propagation

    hidden\_layer\_input = np.dot(X, weights\_input\_hidden)

    hidden\_layer\_output = sigmoid(hidden\_layer\_input)

    output\_layer\_input = np.dot(hidden\_layer\_output, weights\_hidden\_output)

    output\_layer\_output = sigmoid(output\_layer\_input)

    # Calculate the error

    error = y - output\_layer\_output

    # Backpropagation

    d\_output = error \* sigmoid\_derivative(output\_layer\_output)

    error\_hidden\_layer = d\_output.dot(weights\_hidden\_output.T)

    d\_hidden\_layer = error\_hidden\_layer \* sigmoid\_derivative(hidden\_layer\_output)

    # Update weights

    weights\_hidden\_output += hidden\_layer\_output.T.dot(d\_output) \* learning\_rate

    weights\_input\_hidden += X.T.dot(d\_hidden\_layer) \* learning\_rate

# Test the neural network

test\_input = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

test\_output = sigmoid(np.dot(sigmoid(np.dot(test\_input, weights\_input\_hidden)), weights\_hidden\_output))

print("Predicted Output:")

print(test\_output)

17 Write a Prolog Program to Sum the Integers from 1 to n.

sum(0, 0).

sum(N, Result) :-

    N > 0,

    N1 is N - 1,

    sum(N1, SumN1),

    Result is N + SumN1.

%sum(5, Result).

18 Write a Prolog Program for A DB WITH NAME, DOB.

born(jan, 20,3,1977).

born(jeroen, 2,2,1992).

born(joris, 17,3,1995).

born(jelle, 1,1,2004).

born(jesus, 24,12,2000).

born(joop, 30,4,1989).

born(jannecke, 17,3,1993).

born(jaap, 16,11,1995).

19 Write a Prolog Program for STUDENT-TEACHER-SUB-CODE.

takes(jane\_doe, his201).

takes(jane\_doe, cs245).

takes(ajit\_chandra, art302).

takes(ajit\_chandra, cs254).

classmates(X, Y) :- takes(X, Z), takes(Y, Z).

20 Write a Prolog Program for PLANETS DB.

orbits(mercury, sun).

orbits(venus,   sun).

orbits(earth,   sun).

orbits(mars,    sun).

orbits(moon, earth).

orbits(phobos, mars).

orbits(deimos, mars).

planet(P) :- orbits(P,sun).

satellite(S) :- orbits(S,P), planet(P).

%orbits(venus, sun).

21 Write a Prolog Program to implement Towers of Hanoi.

move(1, X, Y, \_) :-

    write('Move top disk from '),

    write(X),

    write(' to '),

    write(Y),

    nl.

move(N, X, Y, Z) :-

    N > 1,

    M is N-1,

    move(M, X, Z, Y),

    move(1, X, Y, \_),

    move(M, Z, Y, X).

    %move(3, 'A', 'C', 'B').

22 Write a Prolog Program to print particular bird can fly or not. Incorporate required queries.

bird(eagle, can\_fly).

bird(penguin, cannot\_fly).

bird(ostrich, cannot\_fly).

bird(sparrow, can\_fly).

can\_fly(Bird) :- bird(Bird, can\_fly).

cannot\_fly(Bird) :- bird(Bird, cannot\_fly).

%can\_fly(eagle).

23 Write the prolog program to implement family tree.

female(pam).

female(liz).

female(pat).

female(ann).

male(jim).

male(bob).

male(tom).

male(peter).

parent(pam,bob).

parent(tom,bob).

parent(tom,liz).

parent(bob,ann).

parent(bob,pat).

parent(pat,jim).

parent(bob,peter).

parent(peter,jim).

mother(X,Y):- parent(X,Y), female(X).

father(X,Y):- parent(X,Y), male(X).

% Define sister relationship

sister(X, Y) :- parent(Z, X), parent(Z, Y), female(X), X \== Y.

% Query for all sisters of "pat"

?- sister(X, pat).

24 Write a Prolog Program to suggest Dieting System based on Disease.

symptom(amit,fever).

    symptom(amit,rash).

    symptom(amit,headache).

    symptom(amit,runny\_nose).

    symptom(kaushal,chills).

    symptom(kaushal,fever).

    symptom(kaushal,hedache).

    symptom(dipen,runny\_nose).

    symptom(dipen,rash).

    symptom(dipen,flu).

    hypothesis(Patient,measels):-

        symptom(Patient,fever),

        symptom(Patient,cough),

        symptom(Patient,conjunctivitis),

        symptom(Patient,rash),

        write('Eat Salad').

    hypothesis(Patient,german\_measles) :-

        symptom(Patient,fever),

        symptom(Patient,headache),

        symptom(Patient,runny\_nose),

        symptom(Patient,rash),

        write('Avoid Oily Food').

    hypothesis(Patient,flu) :-

        symptom(Patient,fever),

        symptom(Patient,headache),

        symptom(Patient,body\_ache),

        symptom(Patient,chills),

        write('Avoid cold food').

    hypothesis(Patient,common\_cold) :-

        symptom(Patient,headache),

        symptom(Patient,sneezing),

        symptom(Patient,sore\_throat),

        symptom(Patient,chills),

        write('Take green vegetables').

        %hypothesis(amit, Disease), symptom(amit, fever), symptom(amit, cough), symptom(amit, conjunctivitis).

25 Write a Prolog program to implement Monkey Banana Problem

on(floor,monkey).

on(floor,chair).

in(room,monkey).

in(room,chair).

in(room,banana).

at(ceiling,banana).

strong(monkey).

grasp(monkey).

climb(monkey,chair).

push(monkey,chair):-

    strong(monkey).

under(banana,chair):-

    push(monkey,chair).

canreach(banana,monkey):-

    at(floor,banana);

    at(ceiling,banana),

    under(banana,chair),

    climb(monkey,chair).

canget(banana,monkey):-

    canreach(banana,monkey),grasp(monkey).

    %canget(banana, monkey).

26 Write a Prolog Program for fruit and its color using Back Tracking.

colour(cherry, red).

colour(banana, yellow).

colour(apple, red).

colour(apple, green).

colour(orange, orange).

colour(x, unknown).

27 Write a Prolog Program to implement Best First Search algorithm

connected(a, b).

connected(a, c).

connected(b, d).

connected(b, e).

connected(c, f).

connected(d, g).

goal\_state(g).

bfs(Start, Path) :-

    bfs([[Start]], Path).

bfs([[Goal | Rest] | \_], [Goal | Rest]) :- goal\_state(Goal).

bfs([[Curr | Path] | Queue], Result) :-

    findall([Next, Curr | Path],

            (connected(Curr, Next), dif(Next, Curr)),

            NextPaths),

    append(Queue, NextPaths, NewQueue),

    bfs(NewQueue, Result).

28 Write the prolog program for Medical Diagnosis

symptom('Flu').

symptom('Yellowish eyes and skin').

symptom('Dark color urine').

symptom('Pale bowel movement').

symptom('Fatigue').

symptom('Vomitting').

symptom('Fever').

symptom('Pain in joints').

symptom('Weakness').

symptom('Stomach Pain').

treatment('Flu', 'Drink hot water, avoid cold eatables.').

treatment('Yellowish eyes and skin', 'Put eye drops, have healthy sleep, do not strain your eyes.').

treatment('Dark color urine', 'Drink lots of water, juices and eat fruits. Avoid alcohol consumption.').

treatment('Pale bowel movement', 'Drink lots of water and exercise regularly.').

treatment('Fatigue', 'Drink lots of water, juices and eat fruits.').

treatment('Vomitting', 'Drink salt and water.').

treatment('Fever', 'Put hot water cloth on head and take crocin.').

treatment('Pain in Joints', 'Apply pain killer and take crocin.').

treatment('Weakness', 'Drink salt and water, eat fruits.').

treatment('Stomach Pain', 'Avoid outside food and eat fruits.').

diagnose(Symptoms, Disease, Treatment) :-

    disease(Disease),

    forall(member(Symptom, Symptoms), has\_symptom(Disease, Symptom)),

    treatment(Disease, Treatment).

has\_symptom(Disease, Symptom) :-

    disease\_symptom(Disease, Symptoms),

    member(Symptom, Symptoms).

disease\_symptom('Flu', ['Flu', 'Fever']).

disease\_symptom('Yellowish eyes and skin', ['Yellowish eyes and skin']).

disease\_symptom('Dark color urine', ['Dark color urine']).

disease\_symptom('Pale bowel movement', ['Pale bowel movement']).

disease\_symptom('Fatigue', ['Fatigue']).

disease\_symptom('Vomitting', ['Vomitting']).

disease\_symptom('Fever', ['Fever']).

disease\_symptom('Pain in Joints', ['Pain in joints']).

disease\_symptom('Weakness', ['Weakness']).

disease\_symptom('Stomach Pain', ['Stomach Pain']).

disease('Flu').

disease('Yellowish eyes and skin').

disease('Dark color urine').

disease('Pale bowel movement').

disease('Fatigue').

disease('Vomitting').

disease('Fever').

disease('Pain in Joints').

disease('Weakness').

disease('Stomach Pain').

%diagnose(['Flu', 'Fever'], Disease, Treatment).

29 Write a Prolog Program for forward Chaining. Incorporate required queries.

% Define the rules and facts

% Animal properties

has\_fur(cat).

has\_feathers(bird).

has\_scales(snake).

% Animal types

animal\_type(X, mammal) :- has\_fur(X).

animal\_type(X, bird) :- has\_feathers(X).

animal\_type(X, reptile) :- has\_scales(X).

% Query

animal(cat).

animal(parrot).

animal(snake).

% Forward chaining

animal\_type(X, Type) :- animal(X), has\_fur(X), Type = mammal.

animal\_type(X, Type) :- animal(X), has\_feathers(X), Type = bird.

animal\_type(X, Type) :- animal(X), has\_scales(X), Type = reptile.

%animal\_type(cat, Type).

30 Write a Prolog Program for backward Chaining. Incorporate required queries.

% Define the rules and facts

% Animal properties

has\_fur(cat).

has\_feathers(bird).

has\_scales(snake).

% Animal types

mammal(X) :- has\_fur(X).

bird(X) :- has\_feathers(X).

reptile(X) :- has\_scales(X).

% Inference rules

is\_mammal(X) :- mammal(X).

is\_mammal(X) :- animal(X), not(bird(X)), not(reptile(X)).

is\_bird(X) :- bird(X).

is\_bird(X) :- animal(X), not(mammal(X)), not(reptile(X)).

is\_reptile(X) :- reptile(X).

is\_reptile(X) :- animal(X), not(mammal(X)), not(bird(X)).

% Query

animal(cat).

animal(parrot).

animal(snake).

% Backward chaining

animal\_type(X, mammal) :- is\_mammal(X).

animal\_type(X, bird) :- is\_bird(X).

animal\_type(X, reptile) :- is\_reptile(X).

%animal\_type(cat, Type).