**CSA1751-Artificial Intelligence for Developing Generative AI Applications**

**1)8 QUEENS**

**AIM:**

Write a python code to solve 8queens problem.

**ALGORITHM:**

1Initialize the chessboard as an 8x8 matrix with all values set to 0, representing no queens placed.

2.Place a queen column by column, starting from the leftmost column.

3.Check if placing a queen in a specific row is safe (i.e., no other queen is present in the same row, column, or diagonals).

4.If safe, place the queen and recur for the next column.

5.If placing a queen leads to a solution, return true. If not, backtrack by removing the queen and try the next possible row.

6.Repeat the process until all queens are placed successfully or all possibilities are exhausted, in which case the solution doesn’t exist.

**CODE:**

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

for i in range(col):

if board[row][i] == 1:

return False

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

if board[i][j] == 1:

return False

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

if board[i][j] == 1:

return False

return True

def solve\_queens(board, col, n):

if col >= n:

return True

for i in range(n):

if is\_safe(board, i, col, n):

board[i][col] = 1

if solve\_queens(board, col + 1, n):

return True

board[i][col] = 0 # Backtrack

return False

def print\_board(board):

for row in board:

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

print()

def solve\_n\_queens(n):

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

if not solve\_queens(board, 0, n):

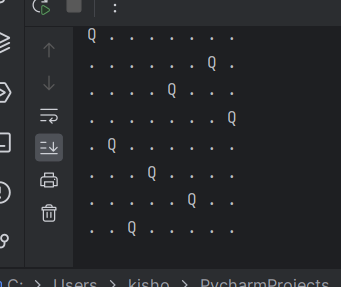
print("Solution does not exist")

return

print\_board(board)

solve\_n\_queens(8)

**OUTPUT:**



**2)WATER JUG PRBLM**

**AIM:**

Write a python code to solve water jug problem.

**ALGORITHM:**

1.Initialize the jugs with capacities for both jugs (jug1, jug2) and the target amount of water to be measured.

2.Use a Depth-First Search (DFS) approach, where you start from an initial state of (0, 0) for both jugs.

3.Check if the current state (jug1, jug2) equals the target in either jug.

Mark the current state as visited to avoid loops.

4.Explore all possible operations:

1.Fill jug1 or jug2.

2.Empty jug1 or jug2.

3.Pour water from one jug to the other until one is full or the other is empty.

5.Recur for each possible operation until the target amount is measured in either jug or all possibilities are explored.

**CODE:**

def waterjug(jug1cap,jug2cap,target,jug1=0,jug2=0,visited=None):

if visited is None:

visited=set()

if jug1==target or jug2==target:

print(f"solved jug1:{jug1} ,jug2:{jug2}")

return True

if (jug1,jug2) in visited:

return False

visited.add((jug1,jug2))

return(

waterjug(jug1cap,jug2cap,target,jug1cap,jug2,visited) or

waterjug(jug1cap,jug2cap,target,jug1,jug2cap,visited) or

waterjug(jug1cap,jug2cap,target,0,jug2,visited) or

waterjug(jug1cap,jug2cap,target,jug1,0,visited) or

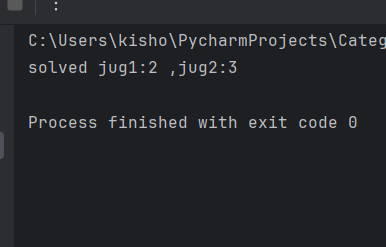
waterjug(jug1cap,jug2cap,target,max(0,jug1-(jug2cap-jug2)),min(jug2cap,jug1+jug2),visited) or

waterjug(jug1cap,jug2cap,target,max(0,jug2-(jug1cap-jug1)),min(jug1cap,jug1+jug2),visited)

)

waterjug(4,3,2)

**OUTPUT:**



**3)CRYPT ARITHMETIC PROBLEM:**

**AIM:**

Write a python code to Implement crypt arithmetic problem.

**ALGORITHM:**

1.Identify the unique letters involved in the cryptarithmetic equation (e.g., SEND + MORE = MONEY).

2.Generate all possible digit permutations for the unique letters (since there are 10 digits, each letter gets a digit between 0-9).

3.For each permutation, assign a digit to each letter.

4.Form the numbers using the assigned digits (e.g., SEND, MORE, and MONEY as numbers).

5.Check if the equation is valid (i.e., SEND + MORE == MONEY).

6.Return the solution if found, or continue searching until all possible digit assignments are checked.

**CODE:**

import itertools

def is\_solution(assignment, s1, s2, s3):

num1 = int("".join(str(assignment[ch]) for ch in s1))

num2 = int("".join(str(assignment[ch]) for ch in s2))

num3 = int("".join(str(assignment[ch]) for ch in s3))

return num1 + num2 == num3

def cryptarithmetic\_solver(s1, s2, s3):

unique\_chars = set(s1 + s2 + s3)

if len(unique\_chars) > 10:

return "Too many letters to map to digits."

for perm in itertools.permutations(range(10), len(unique\_chars)):

assignment = dict(zip(unique\_chars, perm))

if assignment[s1[0]] == 0 or assignment[s2[0]] == 0 or assignment[s3[0]] == 0:

continue

if is\_solution(assignment, s1, s2, s3):

return assignment

return "No solution found."

s1 = "SEND"

s2 = "MORE"

s3 = "MONEY"

result = cryptarithmetic\_solver(s1, s2, s3)

if isinstance(result, dict):

print("Solution found:")

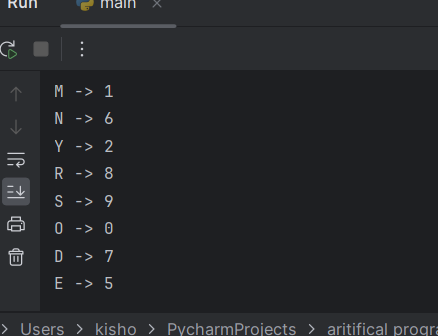
for letter, digit in result.items():

print(f"{letter} -> {digit}")

else:

print(result)

**OUTPUT:**



**4)BFS**

**AIM:**

Write a python code to solve BFS problem.

**ALGORITHM:**

1.Initialize Queue and Visit Start Node: Initialize a queue and add the start node to it. Mark the start node as visited.

2.Dequeue a Node: Dequeue the first node from the queue and process it (e.g., print its value).

3.Visit Unvisited Neighbors: For each neighbor of the dequeued node, if it hasn't been visited, mark it as visited and add it to the queue.

4.Repeat Until Queue is Empty: Continue dequeuing nodes and visiting their unvisited neighbors, adding them to the queue.

5.Terminate When Queue is Empty: The process continues until the queue becomes empty, indicating all reachable nodes have been visited.

6.Output Traversal: Once the queue is empty, output the BFS traversal order.

**CODE:**

from collections import deque

def bfs(graph,start):

queue=deque([start])

visited=set([start])

while queue:

node=queue.popleft()

print(node,end=" ")

for neighbour in graph[node]:

if neighbour not in visited:

visited.add(neighbour)

queue.append(neighbour)

graph= {

'A': ['B','C'],

'B': ['D','E'],

'C': ['F'],

'D': [],

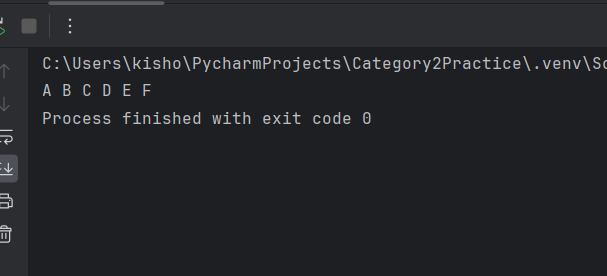
'E': ['F'],

'F': []

}

bfs(graph,'A')

**OUTPUT:**



**5)DFS**

**AIM:**

Write a python code to solve DFS problem.

**ALGORITHM:**

1.Start at Initial Node: Begin the search at the starting node, marking it as visited.

2.Mark Node as Visited: Add the current node to the visited set and process it (e.g., print its value).

3.Explore Neighbors Recursively: For each neighbor of the current node, recursively perform a DFS if the neighbor hasn't been visited yet.

4.Traverse Until All Neighbors are Visited: Continue exploring neighbors recursively until no more unvisited neighbors are found for the current path.

5.Backtrack: Once all the neighbors of a node are visited, backtrack to the previous node in the recursive stack.

6.Terminate When All Nodes are Visited: Repeat the process until all nodes in the graph have been visited.

**CODE:**

def dfs(graph,start,visited=None):

if visited is None:

visited=set()

visited.add(start)

print(start,end=" ")

for neighbour in graph[start]:

if neighbour not in visited:

dfs(graph,neighbour,visited)

graph ={

'A':['B','C'],

'B':['D','E'],

'C':['F'],

'D':[],

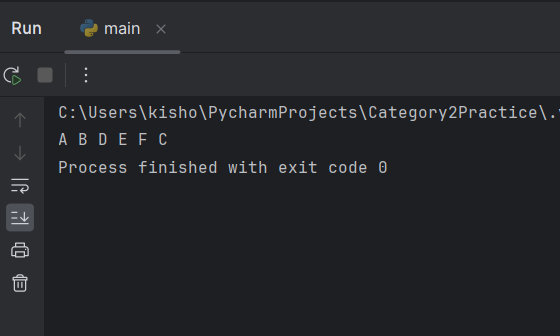
'E':['F'],

'F':[]

}

dfs(graph,'A')

**OUTPUT:**



**6) A\* search**

**AIM:**

Write a python code to solve a\* search problem.

**ALGORITHM:**

1.Initialize Open List: Start by initializing a priority queue (open\_list) with the start node and an empty dictionary to keep track of the path (came\_from). Initialize the g-scores (cost to reach each node) with 0 for the start node.

2.Select the Node with the Lowest f-score: Pop the node with the lowest f-score (the sum of g-score and heuristic) from the open list.

3.Check for Goal: If the current node is the goal node, reconstruct the path by tracing back from the goal to the start node using the came\_from dictionary

4.Expand Neighbors: For each neighboring node of the current node, calculate the tentative g-score (cost to reach the neighbor from the current node).

5.Update Costs: If the neighbor's g-score is lower than a previously recorded g-score (or if it hasn't been visited yet), update its g-score and record its parent in the came\_from dictionary. Compute the f-score and add the neighbor to the open list.

6.Repeat Until Goal is Found or List is Empty: Repeat steps 2-5 until the open list is empty or the goal is reached.

**CODE:**

def dfs(graph,start,visited=None):

if visited is None:

visited=set()

visited.add(start)

print(start,end=" ")

for neighbour in graph[start]:

if neighbour not in visited:

dfs(graph,neighbour,visited)

graph ={

'A':['B','C'],

'B':['D','E'],

'C':['F'],

'D':[],

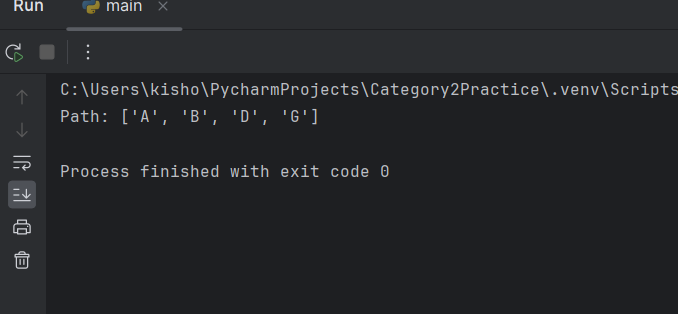
'E':['F'],

'F':[]

}

dfs(graph,'A')

**OUTPUT:**



**7) Map Coloring**

**AIM:**

Write a python code to solve Map coloring problem.

**ALGORITHM:**

1.Initialize Color Set: Start by defining available colors and an empty list for storing the colors assigned to each region (node)

2.Check Feasibility: For each node, check if the current color can be assigned by ensuring no adjacent node (neighbor) has the same color.

3.Assign Color: If a color is valid, assign it to the current node.

4.Recursive Assignment: Recursively assign colors to the next node, ensuring constraints are satisfied.

5.Backtrack if Needed: If no valid color can be assigned to a node, backtrack to the previous node and try a different color.

6.Solution or Failure: If all nodes are colored, return the solution. If not, report failure.

**CODE:**

def is\_safe(node, color, graph, colors):

for neighbor in graph[node]:

if color == colors[neighbor]:

return False

return True

def map\_coloring(graph, colors\_available, colors, node):

if node == len(graph):

return True

for color in colors\_available:

if is\_safe(node, color, graph, colors):

colors[node] = color

if map\_coloring(graph, colors\_available, colors, node + 1):

return True

colors[node] = None

return False

def solve\_map\_coloring(graph, colors\_available):

colors = [None] \* len(graph)

if map\_coloring(graph, colors\_available, colors, 0):

return colors

else:

return None

graph = {

0: [1, 2],

1: [0, 2, 3],

2: [0, 1, 3],

3: [1, 2]

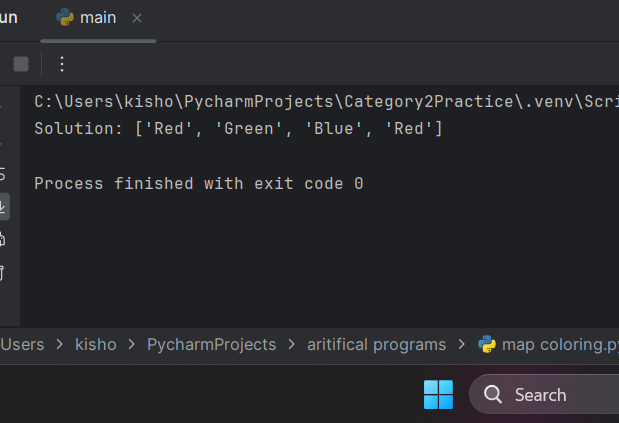
}

colors\_available = ['Red', 'Green', 'Blue']

solution = solve\_map\_coloring(graph, colors\_available)

print("Solution:", solution)

**OUTPUT:**

****

**8) Tic Tac Toe**

**AIM:**

Write a python code to solve Tic Tac Toe problem.

**ALGORITHM:**

1.Initialize Board: Create a 3x3 board (grid) initialized with empty spaces. Set the first player (X) to start.

2.Player Input: Prompt the current player to input their move (row and column).

3.Check Move Validity: Validate the move by checking if the selected cell is empty. If not, ask the player for another move.

4.Update Board: If the move is valid, update the board by placing the player's mark ('X' or 'O') in the chosen cell.

5.Check for Winner or Draw: After each move, check if the current player has won by completing a row, column, or diagonal. Also, check if the board is full (draw).

6.Switch Players: If no winner or draw, switch to the other player and repeat the process until the game ends.

**CODE:**

def print\_board(board):

for row in board:

print("|".join(row))

print("-" \* 5)

def check\_winner(board, player):

for row in board:

if all([spot == player for spot 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\_draw(board):

return all([spot != ' ' for row in board for spot in row])

def tictactoe():

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

current\_player = 'X'

while True:

print\_board(board)

row, col = map(int, input(f"Player {current\_player}, enter your move (row and column): ").split())

if 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\_draw(board):

print\_board(board)

print("It's a draw!")

break

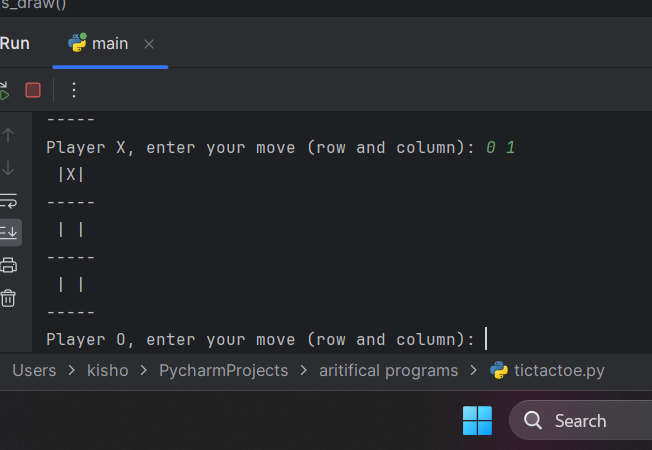
current\_player = 'O' if current\_player == 'X' else 'X'

else:

print("Spot already taken. Try again.")

tictactoe()

**OUTPUT:**



**9) Travelling sales person**

**AIM:**

Write a python code to solve Travelling Sales Person problem.

**ALGORITHM:**

1.Define Graph and Start Point: Create a distance matrix (graph) representing the distances between cities. Define the starting city.

2.Generate Permutations: Generate all possible permutations of the cities (excluding the start city) to consider all possible routes.

3.Calculate Path Cost: For each permutation (route), calculate the total cost by summing the distances between consecutive cities and returning to the start.

4.Track Minimum Path: Track the minimum path cost found during the exploration of all permutations.

5.Update Minimum Cost: Update the minimum path cost whenever a lower-cost path is found.

6.Return Solution: After exploring all routes, return the minimum path cost as the solution to the TSP.

**CODE:**

from itertools import permutations

def travelling\_salesman(graph, start):

vertices = list(range(len(graph)))

vertices.remove(start)

min\_path = float('inf')

for perm in permutations(vertices):

current\_pathweight = 0

k = start

for j in perm:

current\_pathweight += graph[k][j]

k = j

current\_pathweight += graph[k][start]

min\_path = min(min\_path, current\_pathweight)

return min\_path

graph = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]

start = 0

print("Minimum path:", travelling\_salesman(graph, start))

**OUTPUT:**

