DAY - 13

1.Prims algorithm

import heapq

from collections import defaultdict

def prim(graph):

if not graph:

return [], 0

start\_node = next(iter(graph))

mst = []

total\_weight = 0

visited = set([start\_node])

min\_heap = [(weight, start\_node, neighbor) for neighbor, weight in graph[start\_node].items()]

heapq.heapify(min\_heap)

while min\_heap:

weight, u, v = heapq.heappop(min\_heap)

if v not in visited:

visited.add(v)

mst.append((u, v, weight))

total\_weight += weight

for neighbor, weight in graph[v].items():

if neighbor not in visited:

heapq.heappush(min\_heap, (weight, v, neighbor))

return mst, total\_weight

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

graph = {'A': {'B': 2, 'C': 3},

'B': {'A': 2, 'C': 1, 'D': 1},

'C': {'A': 3, 'B': 1, 'D': 2},

'D': {'B': 1, 'C': 2}}

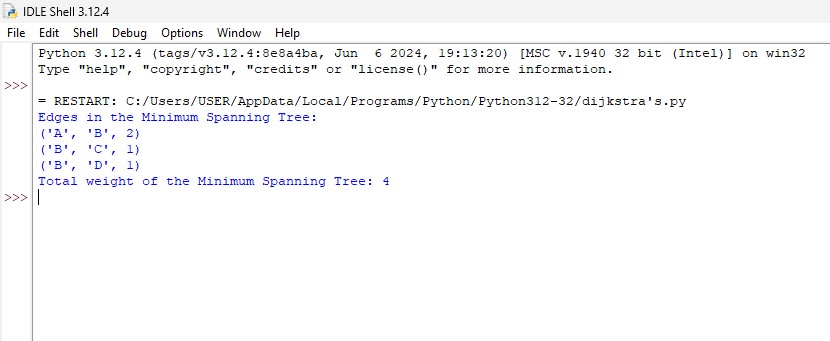
mst, total\_weight = prim(graph)

print("Edges in the Minimum Spanning Tree:")

for edge in mst:

print(edge)

print("Total weight of the Minimum Spanning Tree:", total\_weight)



2. Backtracking

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

for i in range(row):

if board[i][col] == 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, -1, -1), range(col, N)):

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

return False

return True

def solve\_n\_queens\_util(board, row, N, solution):

if row == N:

solution.append(["".join("Q" if x == 1 else "." for x in row) for row in board])

return True

for col in range(N):

if is\_safe(board, row, col, N):

board[row][col] = 1

solve\_n\_queens\_util(board, row + 1, N, solution)

board[row][col] = 0

return False

def solve\_n\_queens(N):

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

solution = []

solve\_n\_queens\_util(board, 0, N, solution)

return solution

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

N = 4

solutions = solve\_n\_queens(N)

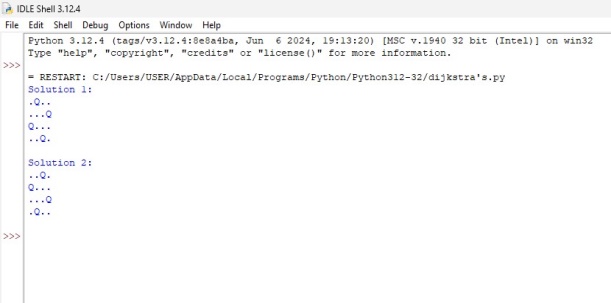
for idx, solution in enumerate(solutions, start=1):

print(f"Solution {idx}:")

for row in solution:

print(row)

print()



3. N queens problem

global N

N = 4

def printSolution(board):

for i in range(N):

for j in range(N):

print (board[i][j],end=' ')

print()

def isSafe(board, row, col):

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 solveNQUtil(board, col):

if col >= N:

return True

for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

if solveNQUtil(board, col + 1) == True:

return True

board[i][col] = 0

return False

def solveNQ():

board = [ [0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0]]

if solveNQUtil(board, 0) == False:

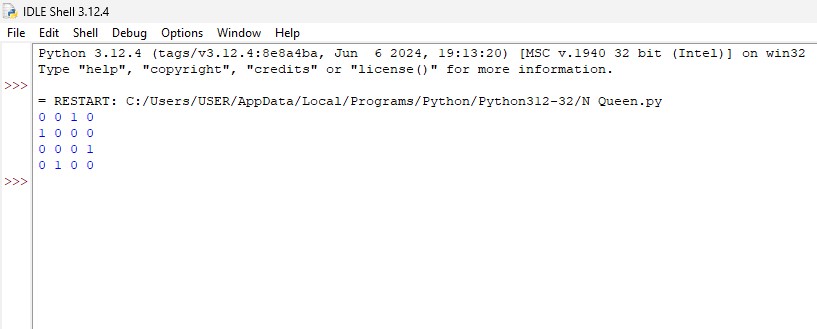
print ("Solution does not exist")

return False

printSolution(board)

return True

solveNQ()



4. Sudoku solver

def is\_valid(board, row, col, num):

if num in board[row]:

return False

for i in range(9):

if board[i][col] == num:

return False

start\_row, start\_col = 3 \* (row // 3), 3 \* (col // 3)

for i in range(start\_row, start\_row + 3):

for j in range(start\_col, start\_col + 3):

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

return False

return True

def solve\_sudoku(board):

empty\_cell = find\_empty\_cell(board)

if not empty\_cell:

return True

row, col = empty\_cell

for num in range(1, 10):

if is\_valid(board, row, col, num):

board[row][col] = num

if solve\_sudoku(board):

return True

board[row][col] = 0

return False

def find\_empty\_cell(board):

for i in range(9):

for j in range(9):

if board[i][j] == 0:

return (i, j)

return None

def print\_board(board):

for row in board:

print(" ".join(map(str, row)))

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

board = [[5, 3, 0, 0, 7, 0, 0, 0, 0],

[6, 0, 0, 1, 9, 5, 0, 0, 0],

[0, 9, 8, 0, 0, 0, 0, 6, 0],

[8, 0, 0, 0, 6, 0, 0, 0, 3],

[4, 0, 0, 8, 0, 3, 0, 0, 1],

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

[0, 6, 0, 0, 0, 0, 2, 8, 0],

[0, 0, 0, 4, 1, 9, 0, 0, 5],

[0, 0, 0, 0, 8, 0, 0, 7, 9]]

print("Original Sudoku board:")

print\_board(board)

print("\nSolving...\n")

if solve\_sudoku(board):

print("Solved Sudoku board:")

print\_board(board)

else:

print("No solution exists for the given Sudoku board.")

