1. Prims algorithm

import heapq

class Graph:

def \_init\_(self, vertices):

self.V = vertices

self.graph = defaultdict(list)

def add\_edge(self, src, dest, weight):

self.graph[src].append((dest, weight))

self.graph[dest].append((src, weight)) # Assuming undirected graph

def prim\_mst(self):

min\_heap = []

heapq.heapify(min\_heap)

visited = [False] \* self.V

mst\_cost = 0

mst\_edges = []

# Start Prim's algorithm from vertex 0 (can start from any vertex)

heapq.heappush(min\_heap, (0, 0)) # (weight, vertex)

while min\_heap:

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

if visited[u]:

continue

# Mark vertex as visited

visited[u] = True

mst\_cost += weight

if u != 0:

mst\_edges.append((parent[u], u, weight))

# Add adjacent vertices of u to the min-heap if not visited

for v, w in self.graph[u]:

if not visited[v]:

heapq.heappush(min\_heap, (w, v))

return mst\_edges, mst\_cost

# Example usage:

g = Graph(4)

g.add\_edge(0, 1, 10)

g.add\_edge(0, 2, 6)

g.add\_edge(0, 3, 5)

g.add\_edge(1, 3, 15)

g.add\_edge(2, 3, 4)

mst\_edges, mst\_cost = g.prim\_mst()

print("Minimum Spanning Tree Edges:")

for edge in mst\_edges:

print(edge)

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

1. Backtracking

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

# Check this row on the left side

for i in range(col):

if board[row][i] == 'Q':

return False

# Check upper diagonal on left side

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

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

return False

# Check lower diagonal on left side

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

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

return False

return True

def solve\_n\_queens(N):

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

result = []

def backtrack(board, col):

if col == N:

result.append([''.join(row) for row in board])

return

for i in range(N):

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

board[i][col] = 'Q'

backtrack(board, col + 1)

board[i][col] = '.'

backtrack(board, 0)

return result

# Example 1

N1 = 4

result1 = solve\_n\_queens(N1)

print(f"Solution for Example 1 with N = {N1}:")

for solution in result1:

print(solution)

print()

# Example 2

N2 = 1

result2 = solve\_n\_queens(N2)

print(f"Solution for Example 2 with N = {N2}:")

for solution in result2:

print(solution)

1. N queens problem

#include <stdio.h>

#include <stdbool.h>

#define N 8

int board[N][N];

bool isSafe(int row, int col) {

int i, j;

for (i = 0; i < col; i++)

if (board[row][i])

return false;

for (i = row, j = col; i >= 0 && j >= 0; i--, j--)

if (board[i][j])

return false;

for (i = row, j = col; j >= 0 && i < N; i++, j--)

if (board[i][j])

return false;

return true;

}

bool solveNQueensUtil(int col) {

if (col >= N)

return true;

for (int i = 0; i < N; i++) {

if (isSafe(i, col)) {

board[i][col] = 1;

if (solveNQueensUtil(col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

void solveNQueens() {

if (solveNQueensUtil(0)) {

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++)

printf("%d ", board[i][j]);

printf("\n");

}

} else {

printf("Solution does not exist");

}

}

int main() {

solveNQueens();

return 0;

}

1. Sudoku solve

return !isPresentInRow(row, num) && !isPresentInCol(col, num) && !isPresentInBox(row - row%3 , col - col%3, num);

}

int solveSudoku() {

int row, col;

//when all places are filled

if (!findEmptyPlace(&row, &col))

return 1;

//valid numbers are 1 - 9

for (int num = 1; num <= 9; num++) {

//check validation, if yes, put the number in the grid

if (isValidPlace(row, col, num)) {

grid[row][col] = num;

//recursively go for other rooms in the grid

if (solveSudoku())

return 1;

//turn to unassigned space when conditions are not satisfied

grid[row][col] = 0;

}

}

return 0;

}

int main() {

if (solveSudoku() == 1)

sudokuGrid();

else

printf("Can't get a solution");

}