```python
#!/usr/bin/env python
# coding: utf-8

# In[6]:


#program 1
def solveWaterJugProblem(capacity_jug1, capacity_jug2, desired_quantity):
    stack = [(0, 0)]  # Initial state: both jugs empty
    visited = set()   # Track visited states

    while stack:
        current_state = stack.pop()

        # Check if current state meets the desired quantity
        if current_state[0] == desired_quantity or current_state[1] == desired_quantity:
            return current_state

        # Mark current state as visited
        if current_state in visited:
            continue
        visited.add(current_state)

        # Generate and add next possible states
        next_states = generateNextStates(current_state, capacity_jug1, capacity_jug2)
        stack.extend(next_states)

    return "No solution found"


def generateNextStates(state, capacity_jug1, capacity_jug2):
    next_states = []

    jug1, jug2 = state

    # Fill Jug 1
    next_states.append((capacity_jug1, jug2))

    # Fill Jug 2
    next_states.append((jug1, capacity_jug2))

    # Empty Jug 1
    next_states.append((0, jug2))

    # Empty Jug 2
    next_states.append((jug1, 0))

    # Pour water from Jug 1 to Jug 2
    pour_amount = min(jug1, capacity_jug2 - jug2)
    next_states.append((jug1 - pour_amount, jug2 + pour_amount))

    # Pour water from Jug 2 to Jug 1
    pour_amount = min(jug2, capacity_jug1 - jug1)
    next_states.append((jug1 + pour_amount, jug2 - pour_amount))

    return next_states


# Run the fixed program
solution = solveWaterJugProblem(4, 3, 2)
print("Solution:", solution)


# In[7]:


#program 2
```

```python
from collections import deque

def is_valid(state):
    """
    Checks if a given state is valid.
    Conditions:
    1. No negative values.
    2. Missionaries should never be outnumbered by cannibals on either side.
    """
    leftM, leftC, rightM, rightC, boat = state

    # No negative values allowed
    if leftM < 0 or leftC < 0 or rightM < 0 or rightC < 0:
        return False

    # Missionaries should never be outnumbered by cannibals on either side
    if (leftM > 0 and leftC > leftM) or (rightM > 0 and rightC > rightM):
        return False

    return True

def solve():
    """
    Solves the Missionaries and Cannibals puzzle using BFS.
    Returns the shortest path from the initial state to the goal state.
    """
    initial_state = (3, 3, 0, 0, 'left')  # (leftM, leftC, rightM, rightC, boat_position)
    goal_state = (0, 0, 3, 3, 'right')  # Goal: All on the right side

    visited = set()  # To track visited states
    queue = deque()  # BFS queue

    queue.append([initial_state])  # Start with the initial state

    while queue:
        path = queue.popleft()  # Get the first path from the queue
        current = path[-1]  # Get the last state in the path

        if current[:4] == goal_state[:4]:  # Check if we reached the goal
            return path

        if current in visited:  # Skip already visited states
            continue
        visited.add(current)

        lm, lc, rm, rc, boat = current  # Extract current state values
        transitions = []  # Store possible next moves

        # Possible moves (1M, 0C), (2M, 0C), (0M, 1C), (0M, 2C), (1M, 1C)
        moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]

        for m, c in moves:
            if boat == 'left':  # Move to the right side
                new_state = (lm - m, lc - c, rm + m, rc + c, 'right')
            else:  # Move back to the left side
                new_state = (lm + m, lc + c, rm - m, rc - c, 'left')

            if is_valid(new_state) and new_state not in visited:
                queue.append(path + [new_state])  # Append new state to the path

    return None  # No solution found

def print_solution(path):
    """
    Prints the steps of the optimal solution.
    """
    print("\nOptimal Solution Steps:")
```

```python
    for i, state in enumerate(path):
        lm, lc, rm, rc, boat = state
        print(f"Step {i}:")
        print(f"Left: {lm}M {lc}C | Boat: {boat} | Right: {rm}M {rc}C")
        if i < len(path) - 1:
            moved_m = abs(path[i+1][0] - lm)
            moved_c = abs(path[i+1][1] - lc)
            print(f"Action: Move {moved_m}M and {moved_c}C {'right' if boat == 'left' else 'left'}")

# Solve the puzzle and print the solution
solution = solve()
if solution:
    print_solution(solution)
else:
    print("No solution found")


# In[8]:


class Graph:
    def __init__(self, adjac_list):
        self.adjac_list = adjac_list
        print("Input Graph:\n", self.adjac_list)

    def get_neighbors(self, v):
        return self.adjac_list[v]

    def h(self, n):
        H = {
            'A': 11,
            'B': 6,
            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0,
        }
        return H[n]

    def AStar(self, start, stop):
        open_list = set([start])
        closed_list = set([])
        g = {}
        g[start] = 0
        parents = {}
        parents[start] = start

        while len(open_list) > 0:
            n = None

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

            if n is None:
                print('Path does not exist!')
                return None

            if n == stop:
                reconst_path = []

                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]

                reconst_path.append(start)
                reconst_path.reverse()
```

```python
                print('Path found: {}'.format(reconst_path))
                print('Cost of the path is:', g[stop])
                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


# Example graph input
adjac_list = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1), ('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
    'G': None
}

graph1 = Graph(adjac_list)
graph1.AStar('A', 'G')


# In[9]:


#4th program AO*
class Graph:
    def __init__(self, graph, hVals, startNode):
        self.graph = graph
        self.H = hVals
        self.start = startNode
        self.parent = {}
        self.status = {}
        self.solutionGraph = {}

    def getNeighbors(self, v):
        return self.graph.get(v, '')

    def getStatus(self, v):
        return self.status.get(v, 0)

    def setStatus(self, v, val):
        self.status[v] = val

    def getHval(self, n):
        return self.H.get(n, float('inf'))

    def setHval(self, n, value):
        self.H[n] = value
```

```python
    def printSolution(self):
        print("\nFinal Heuristic Values:")
        for node, val in self.H.items():
            print(f"{node}: {val}")

        print("\nBest Path to Goal State:")
        if self.solutionGraph:
            for node, path in self.solutionGraph.items():
                print(f"{node} -> {path}")
        else:
            print("No valid solution path found.")

        print(f"\nMinimum Cost: {self.H[self.start]}")

    def computeMinCost(self, v):
        neighbors = self.getNeighbors(v)

        if not neighbors:
            # If no neighbors, return infinite cost and empty path
            return float('inf'), []

        minimumCost = float('inf')
        costList = {}

        for nodes in neighbors:
            cost = 0
            nodeList = []

            for c, weight in nodes:
                cost += self.getHval(c) + weight
                nodeList.append(c)

            if cost < minimumCost:
                minimumCost = cost
                costList[minimumCost] = nodeList

        return minimumCost, costList.get(minimumCost, [])

    def AOStar(self, v, backTracking):
        if self.getStatus(v) >= 0:
            minimumCost, childList = self.computeMinCost(v)

            if not childList and minimumCost == float('inf'):
                return

            self.setHval(v, minimumCost)
            self.setStatus(v, len(childList))

            solved = all(self.getStatus(child) == -1 for child in childList)

            if solved:
                self.setStatus(v, -1)
                self.solutionGraph[v] = childList

            if v != self.start:
                self.AOStar(self.parent.get(v, self.start), True)

            if not backTracking:
                for child in childList:
                    self.parent[child] = v
                    self.setStatus(child, 0)
                    self.AOStar(child, False)


# Example Usage
h1 = {'A': 0, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}
```

```python
graph1 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'C': [[('J', 1)]],
    'D': [[('E', 1), ('F', 1)]],
    'G': [[('I', 1)]]
}

# Instantiate and run the AO* algorithm
G1 = Graph(graph1, h1, 'A')
G1.AOStar('A', False)
G1.printSolution()


# In[10]:


#5th program Nqueen
N = 8
def print_board(board):
    for row in board:
        print(" ".join("Q" if cell else "." for cell in row))
    print("\n")

def is_safe(board, row, col):
    for i in range(row):  # Check column
        if board[i][col]:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):  # Check left diagonal
        if board[i][j]:
            return False
    for i, j in zip(range(row, -1, -1), range(col, N)):  # Check right diagonal
        if board[i][j]:
            return False
    return True

def solve_n_queens(board, row=0):
    if row == N:  # If all queens are placed, print the board
        print_board(board)
        return True

    for col in range(N):  # Try placing the Queen in every column
        if is_safe(board, row, col):  # Check if it's safe
            board[row][col] = 1  # Place Queen
            if solve_n_queens(board, row + 1):
                return True  # If successful, return True
            board[row][col] = 0  # Backtrack if needed

    return False  # No solution found

# Initialize an empty chessboard
chessboard = [[0] * N for _ in range(N)]
solve_n_queens(chessboard)


# In[3]:


#6th program
import sys

def nearest_neighbor(graph):
    """Find a near-optimal path using the nearest neighbor algorithm."""
    num_cities = len(graph)
    visited = [False] * num_cities
    path = []
```

```python
    # Start from the first city
    current_city = 0
    visited[current_city] = True
    path.append(current_city)

    # Visit each city exactly once
    for _ in range(num_cities - 1):
        nearest_city = None
        min_distance = sys.maxsize

        # Find the nearest unvisited city
        for next_city in range(num_cities):
            if not visited[next_city] and graph[current_city][next_city] < min_distance:
                nearest_city = next_city
                min_distance = graph[current_city][next_city]

        # Move to the nearest unvisited city
        if nearest_city is not None:
            current_city = nearest_city
            visited[current_city] = True
            path.append(current_city)

    # Return to the starting city
    path.append(path[0])

    return path


def take_input():
    """Function to take input from the user."""
    graph = []

    # Get the number of cities
    num_cities = int(input("Enter the number of cities: "))

    print("Enter the distance matrix (one row at a time):")

    for _ in range(num_cities):
        row = list(map(int, input().split()))

        # Ensure the row length matches the number of cities
        if len(row) != num_cities:
            print("Invalid row length. Please enter exactly", num_cities, "values.")
            return take_input()

        graph.append(row)

    return graph


# Example usage
if __name__ == "__main__":
    # Take input from the user
    graph = take_input()

    # Validate the matrix dimensions
    if len(graph) != len(graph[0]):
        print("Error: Distance matrix must be square.")
    else:
        # Find a near-optimal path using the nearest neighbor algorithm
        optimal_path = nearest_neighbor(graph)

        print("\nOptimal Path:", " -> ".join(map(str, optimal_path)))
```