**EXERCISE 11**

 **CASE STUDIES**

 **Case study 1:** **Find Hamiltonian cycle is present in the graph or not using suitable algorithm design technique. Write the documentation for the problem**

 **Documentation**: Hamiltonian Cycle Problem

**Problem Statement**

A \*Hamiltonian cycle\* in a graph is a cycle that visits each vertex exactly once and returns to the starting vertex. Given an undirected graph \( G = (V, E) \), determine whether a Hamiltonian cycle exists in \( G \). If such a cycle exists, output the cycle; otherwise, state that no Hamiltonian cycle exists.

 **Input**

- A graph \( G = (V, E) \) where \( V \) is a set of vertices and \( E \) is a set of edges.

 - Represented as an adjacency matrix or adjacency list.

**Output**

- \*Yes\*: If a Hamiltonian cycle exists, along with the vertices in the order of traversal.

- \*No\*: If a Hamiltonian cycle does not exist.

**Algorithm Design Technique**

\*Backtracking\* is the most suitable algorithm design technique for solving this problem. It explores all possible paths in the graph to determine whether a Hamiltonian cycle exists while pruning invalid paths.

**Algorithm: Backtracking for Hamiltonian Cycle**

**1. Start**:

 - Choose an arbitrary vertex as the starting point.

 - Mark the vertex as visited and initialize the path.

2**. Recursive Exploration**:

 - Explore all unvisited neighbors of the current vertex.

 - Add a neighbor to the path and mark it as visited.

 - Recur to find if the cycle continues.

**3. Base Case**:

 - If all vertices are visited:

 - Check if the last vertex in the path connects back to the starting vertex. If yes, a Hamiltonian cycle is found.

**4. Backtrack:**

 - If a vertex does not lead to a solution, remove it from the path and mark it as unvisited.

**5. Output:**

 - If all possibilities are exhausted and no Hamiltonian cycle is found, return \*No\*.

 **Pseudocode**

function isHamiltonianCycle(graph, path, pos):

 if pos == len(graph): # Base case: All vertices visited

 return graph[path[pos-1]][path[0]] == 1 # Check if cycle is complete

 for vertex in range(1, len(graph)): # Explore neighbors

 if isSafe(vertex, graph, path, pos):

 path[pos] = vertex

 if isHamiltonianCycle(graph, path, pos+1): # Recur for next vertex

 return True

 path[pos] = -1 # Backtrack

 return False

function isSafe(vertex, graph, path, pos):

 if graph[path[pos-1]][vertex] == 0: # No edge exists

 return False

 if vertex in path: # Vertex already visited

 return False

 return True

function findHamiltonianCycle(graph):

 path = [-1] \* len(graph)

 path[0] = 0 # Start from vertex 0

 if isHamiltonianCycle(graph, path, 1):

 return path + [path[0]] # Return the cycle

 else:

 return "No Hamiltonian Cycle"

 **Complexity**

1. \*Time Complexity\*: \( O(V!) \) in the worst case since it explores all permutations of vertices.

2. \*Space Complexity\*: \( O(V) \) for the path array and recursion stack.

This algorithm is practical for small graphs. For larger graphs, heuristic or approximation algorithms like Genetic Algorithms or Dynamic Programming-based approaches (e.g., Held-Karp) can be considered.



**Program code:**

#include <stdio.h>

#include <stdbool.h>

#define V 10 // Maximum number of vertices (can be adjusted)

// Function to check if it is safe to add the current vertex to the path

bool isSafe(int v, int graph[V][V], int path[], int pos, int n) {

 // Check if there is an edge from the last vertex in the path to the current vertex

 if (graph[path[pos - 1]][v] == 0)

 return false;

 // Check if the vertex is already in the path

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

 if (path[i] == v)

 return false;

 }

 return true;

}

// Recursive utility function to find the Hamiltonian cycle

bool isHamiltonianCycleUtil(int graph[V][V], int path[], int pos, int n) {

 // Base case: If all vertices are included in the path

 if (pos == n) {

 // Check if the last vertex is connected to the first vertex

 return graph[path[pos - 1]][path[0]] == 1;

 }

 // Try different vertices as the next candidate

 for (int v = 1; v < n; v++) {

 if (isSafe(v, graph, path, pos, n)) {

 path[pos] = v;

 // Recur to construct the rest of the path

 if (isHamiltonianCycleUtil(graph, path, pos + 1, n))

 return true;

 // Backtrack

 path[pos] = -1;

 }

 }

 return false;

}

// Function to find and print the Hamiltonian cycle if it exists

void findHamiltonianCycle(int graph[V][V], int n) {

 int path[n];

 for (int i = 0; i < n; i++)

 path[i] = -1;

 path[0] = 0; // Start from the first vertex

 if (isHamiltonianCycleUtil(graph, path, 1, n)) {

 printf("Hamiltonian Cycle exists: ");

 for (int i = 0; i < n; i++)

 printf("%d ", path[i]);

 printf("%d\n", path[0]); // Complete the cycle by returning to the start

 } else {

 printf("No Hamiltonian Cycle exists.\n");

 }

}

int main() {

 int n;

 int graph[V][V];

 printf("Enter the number of vertices: ");

 scanf("%d", &n);

 printf("Enter the adjacency matrix of the graph (enter 0 or 1):\n");

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

 for (int j = 0; j < n; j++) {

 scanf("%d", &graph[i][j]);

 }

 }

 findHamiltonianCycle(graph, n);

 return 0;

}

**Output:**

**Case study 2:  A 3 x 3 board with 8 tiles (each tile has a number ranging from 1 to 8) and a single empty space is provided. The goal is to use the vacant space to arrange the numbers on the tiles so that they match the final arrangement. Four neighboring (left, right, above, and below) tiles can be slid into the available area.  Solve this problem using suitable algorithm design technique
The sample puzzle shown in img1put**



To solve the \*8-Puzzle Problem, we will use the \*\*A Search Algorithm\*\*—a popular algorithm design technique for solving such pathfinding problems. Below is a detailed solution approach for this problem.

**Problem Statement**

Given a 3x3 board with tiles numbered 1 through 8 and one empty space, the objective is to slide tiles into the empty space until the board matches the final arrangement.

**Initial State:**

2 8 3

1 6

7 4 5

**Final State**:

1 2 3

8 4

7 6 5

**Algorithm Design Technique: A Search Algorithm**

**1. State Representation**:

 - Each state of the board is represented as a 2D matrix.

 - The position of the empty space is tracked for generating possible moves.

**2. Heuristic Function (h(n)):**

 - Use the \*Manhattan Distance\* as the heuristic. It calculates the sum of the vertical and horizontal distances of each tile from its goal position.

**3. Cost Function (f(n)):**

 

**4. Priority Queue**:

 - Use a priority queue (min-heap) to explore states based on the lowest \( f(n) \).

**5. Goal Test:**

 - Check if the current state matches the final arrangement.

**Steps of the Algorithm**

**1. Initialization:**

 - Start with the initial state.

 - Calculate its \( f(n) \) and push it into the priority queue.

**2. Expand Nodes:**

 - Pop the state with the lowest \( f(n) \) from the queue.

 - Generate all possible moves by sliding tiles into the empty space.

**3. Heuristic Calculation**:

 - For each new state, calculate \( f(n) \) and add it to the queue if it hasn’t been visited before.

**4. Repeat:**

 - Continue expanding nodes until the goal state is reached.

**5. Output:**

 - Print the sequence of moves leading to the goal state.

**Program code**:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

#define N 3 // Size of the puzzle (3x3)

// Structure to represent a state of the puzzle

typedef struct {

 int board[N][N];

 int x, y; // Position of the empty tile

 int g; // Cost to reach this state

 int h; // Heuristic value

 int f; // f = g + h

} State;

// Goal state of the puzzle

int goal[N][N] = {

 {1, 2, 3},

 {8, 0, 4},

 {7, 6, 5}

};

// Moves for the empty tile (up, down, left, right)

int dx[] = {-1, 1, 0, 0};

int dy[] = {0, 0, -1, 1};

// Function to calculate Manhattan Distance heuristic

int calculateHeuristic(int board[N][N]) {

 int h = 0;

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

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

 if (board[i][j] != 0) {

 int value = board[i][j];

 int targetX = (value - 1) / N;

 int targetY = (value - 1) % N;

 h += abs(i - targetX) + abs(j - targetY);

 }

 }

 }

 return h;

}

// Function to check if the current board matches the goal state

bool isGoalState(int board[N][N]) {

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

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

 if (board[i][j] != goal[i][j]) return false;

 }

 }

 return true;

}

// Function to copy the board

void copyBoard(int dest[N][N], int src[N][N]) {

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

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

 dest[i][j] = src[i][j];

 }

 }

}

// Function to print the board

void printBoard(int board[N][N]) {

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

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

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

 }

 printf("\n");

 }

 printf("\n");

}

// Solve the puzzle using the A\* algorithm

void solvePuzzle(int initial[N][N]) {

 State queue[1000]; // Priority queue for states

 int front = 0, rear = 0;

 // Initial state

 State start;

 copyBoard(start.board, initial);

 start.g = 0;

 start.h = calculateHeuristic(initial);

 start.f = start.g + start.h;

 // Find the position of the empty tile

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

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

 if (initial[i][j] == 0) {

 start.x = i;

 start.y = j;

 break;

 }

 }

 }

 // Add initial state to the queue

 queue[rear++] = start;

 while (front < rear) {

 // Get the state with the lowest f value

 State current = queue[front++];

 if (isGoalState(current.board)) {

 printf("Solution found in %d moves:\n", current.g);

 printBoard(current.board);

 return;

 }

 // Generate all possible moves

 for (int i = 0; i < 4; i++) {

 int newX = current.x + dx[i];

 int newY = current.y + dy[i];

 // Check if the move is valid

 if (newX >= 0 && newX < N && newY >= 0 && newY < N) {

 State next;

 copyBoard(next.board, current.board);

 // Swap the empty tile with the neighboring tile

 next.board[current.x][current.y] = next.board[newX][newY];

 next.board[newX][newY] = 0;

 next.x = newX;

 next.y = newY;

 next.g = current.g + 1;

 next.h = calculateHeuristic(next.board);

 next.f = next.g + next.h;

 // Add the new state to the queue

 queue[rear++] = next;

 }

 }

 }

 printf("No solution found.\n");

}

int main() {

 int initial[N][N];

 printf("Enter the initial state of the puzzle (3x3 matrix, use 0 for the empty space):\n");

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

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

 scanf("%d", &initial[i][j]);

 }

 }

 printf("\nInitial State:\n");

 printBoard(initial);

 printf("Solving the puzzle...\n");

 solvePuzzle(initial);

 return 0;

}

**Output:**

