**ADS LAB ASSIGNMENT-06**

**NAME:** Manasvi M. Patil

**CLASS:** SY-IT-C

**ROLL NO.:** 03

**AIM:** Write C/C++ program for representation of graphs using adjacency matrix and adjacency lists.

**THEORY:**

The problem statement revolves around representing graphs using two fundamental data structures: adjacency matrix and adjacency lists.

**Graphs:** Graphs are mathematical structures consisting of vertices (nodes) connected by edges (links). They can be categorized into directed and undirected graphs based on the presence or absence of directionality in edges. Graphs can also be further classified based on their density: dense graphs have many edges, while sparse graphs have relatively fewer edges compared to the maximum possible edges.

**Adjacency Matrix Representation:** An adjacency matrix is a 2D array where each cell `adjMatrix[i][j]` represents whether an edge exists between vertices `i` and `j`. For an undirected graph, the matrix is symmetric across the diagonal. This representation allows for straightforward edge queries in constant time but consumes more memory, especially for sparse graphs, as it requires space proportional to the square of the number of vertices.

**Adjacency List Representation:** Adjacency lists use an array of linked lists where each element represents a vertex, and its associated linked list contains all its adjacent vertices. This representation is memory-efficient for sparse graphs as it only stores information about existing edges. However, it might take longer to check for edge existence, requiring traversal through the linked lists. Nonetheless, it enables quicker iteration through adjacent vertices.

Choosing between these representations depends on various factors like the graph's density, memory efficiency, and the operations predominantly performed on the graph, such as querying edge existence, traversing neighbours, or memory constraints. The choice involves a trade-off between memory usage and time complexity for operations, and the most suitable representation depends on the specific characteristics and requirements of the graph being modelled or analysed.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

// Graph representation using Adjacency Matrix

typedef struct {

    int V; // Number of vertices

    int \*\*adjMatrix; // Adjacency Matrix

} GraphMatrix;

// Function to create a graph with V vertices

GraphMatrix\* createGraphMatrix(int vertices) {

    GraphMatrix\* graph = (GraphMatrix\*)malloc(sizeof(GraphMatrix));

    graph->V = vertices;

    // Allocate memory for the adjacency matrix and initialize with 0 (no edges)

    graph->adjMatrix = (int\*\*)malloc(vertices \* sizeof(int\*));

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

        graph->adjMatrix[i] = (int\*)calloc(vertices, sizeof(int));

    }

    return graph;

}

// Function to add an edge between vertices u and v in adjacency matrix

void addEdgeMatrix(GraphMatrix\* graph, int u, int v) {

    graph->adjMatrix[u][v] = 1;

    graph->adjMatrix[v][u] = 1; // If undirected graph

}

// Function to print the adjacency matrix representation of the graph

void printGraphMatrix(GraphMatrix\* graph) {

    printf("Adjacency Matrix representation of the graph:\n");

    for (int i = 0; i < graph->V; ++i) {

        for (int j = 0; j < graph->V; ++j) {

            printf("%d ", graph->adjMatrix[i][j]);

        }

        printf("\n");

    }

}

// Graph representation using Adjacency List

typedef struct Node {

    int dest;

    struct Node\* next;

} Node;

typedef struct {

    Node\* head;

} LinkedList;

typedef struct {

    int V; // Number of vertices

    LinkedList\* array; // Array of linked lists

} GraphList;

// Function to create a graph with V vertices

GraphList\* createGraphList(int vertices) {

    GraphList\* graph = (GraphList\*)malloc(sizeof(GraphList));

    graph->V = vertices;

    // Allocate memory for an array of linked lists

    graph->array = (LinkedList\*)malloc(vertices \* sizeof(LinkedList));

    // Initialize each linked list as empty by setting head as NULL

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

        graph->array[i].head = NULL;

    }

    return graph;

}

// Function to add an edge between vertices u and v in adjacency list

void addEdgeList(GraphList\* graph, int u, int v) {

    // Add an edge from u to v

    Node\* newNode = (Node\*)malloc(sizeof(Node));

    newNode->dest = v;

    newNode->next = graph->array[u].head;

    graph->array[u].head = newNode;

    // For undirected graph, add an edge from v to u as well

    newNode = (Node\*)malloc(sizeof(Node));

    newNode->dest = u;

    newNode->next = graph->array[v].head;

    graph->array[v].head = newNode;

}

// Function to print the adjacency list representation of the graph

void printGraphList(GraphList\* graph) {

    printf("Adjacency List representation of the graph:\n");

    for (int i = 0; i < graph->V; ++i) {

        Node\* currentNode = graph->array[i].head;

        printf("Vertex %d :", i);

        while (currentNode != NULL) {

            printf(" -> %d", currentNode->dest);

            currentNode = currentNode->next;

        }

        printf("\n");

    }

}

int main() {

    // Creating a graph with 5 vertices

    int vertices = 5;

    // Graph using Adjacency Matrix

    GraphMatrix\* graphMatrix = createGraphMatrix(vertices);

    addEdgeMatrix(graphMatrix, 0, 1);

    addEdgeMatrix(graphMatrix, 0, 2);

    addEdgeMatrix(graphMatrix, 1, 3);

    addEdgeMatrix(graphMatrix, 2, 4);

    printGraphMatrix(graphMatrix);

    printf("\n");

    // Graph using Adjacency List

    GraphList\* graphList = createGraphList(vertices);

    addEdgeList(graphList, 0, 1);

    addEdgeList(graphList, 0, 2);

    addEdgeList(graphList, 1, 3);

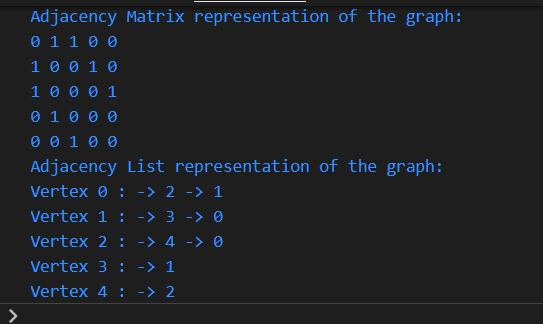
    addEdgeList(graphList, 2, 4);

    printGraphList(graphList);

    return 0;

}

**OUTPUT:**

****

**CONCLUSION:**

Graph representations via an adjacency matrix and adjacency lists offer distinct trade-offs based on the nature of the graph and the operations performed. The adjacency matrix, utilizing a 2D array, provides a straightforward approach for representing edges between vertices, enabling constant-time queries regarding edge existence. However, its memory usage grows quadratically with the number of vertices, making it less efficient for sparse graphs where edges are limited, leading to potential memory wastage. On the other hand, the adjacency list, employing an array of linked lists, efficiently uses memory by only storing information about existing edges, making it suitable for sparse graphs. Although it requires more time to check edge existence due to traversing linked lists, it facilitates faster iteration through adjacent vertices. The choice between these representations depends on the graph's characteristics; adjacency matrices prove useful for dense graphs where quick edge queries are crucial, while adjacency lists shine in scenarios with sparse graphs, optimizing memory usage and traversal operations. Overall, the decision involves a trade-off between space efficiency, query speed, and the graph's density.

**THANK YOU!**