EXPERIMENT NO. 9

Author: Harsheet Chordiya

Roll No.: 37

Sem and Section: 3 CSEB

Source file: expt09.c

Date: 12/02/2021

==

Aim: To study a graph data structure and demonstrate different traversals on it - Depth First Search and Breadth First Search.

Problem Statement: Write a C routine to create an arbitrary graph [a minimum of 6 vertices] using adjacency matrix representation. Write menu driven C program that will include routines to - (1) display the graph [the adjacency matrix], (2) execute depth-first search, and (3) execute breadth-first search.

==

THEORY

==

Introduction to graphs:

A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.

A graph can be directed or undirected. In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A. In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).

Different data structures for the representation of graphs are:

Adjacency List and Adjacency matrix.

Adjacency Matrix Representation:

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them. In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of n ¥ n. In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry aij in the adjacency matrix will contain 1, if vertices vi and vj are adjacent to each other. However, if the nodes are not adjacent, aijwill be set to zero. Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix.

Adjacency List Representation:

An adjacency list is another way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it. The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node. It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

Breadth-First Search:

Breadth-first search (BFS) is a graph search algorithm that begins at the root node.That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine The neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

Space complexity: In the breadth-first search algorithm, all the nodes at a particular level must be saved until their child nodes in the next level have been generated. The space complexity is therefore proportional to the number of nodes at the deepest level of the graph. Given a graph with branching factor b (number of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level O(bd).

Time complexity: In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches O(bd). However, the time complexity can also be expressed as O(| E | + | V |), since every vertex and every edge will be explored in the worst case.

Applications of Breadth-First Search Algorithm:

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

Depth-first Search:

The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored. In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the currentnode. Otherwise, the unvisited (unprocessed) node becomes the current node.

Space complexity: The space complexity of a depth-first search is lower than that of a breadth-first search.

Time complexity: The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as (O(|V| + |E|)).

Applications of Depth-First Search Algorithm:

Depth-first search is useful for:

- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

==

ALGORITHMS

==

Algorithm of BFS:

```
Step 1: SET STATUS=1 (ready state)
        for each node in G
Step 2: Enqueue the starting node A
        and set its STATUS=2
(waiting state)
Step 3: Repeat Steps 4 and 5 until
        QUEUE is empty
Step 4: Dequeue a node N. Process it
      and set its STATUS=3 (processed state).
Step 5: Enqueue all the neighbours of
        N that are in the ready state
        (whose STATUS=1) and set
        their STATUS=2
        (waiting state)
         [END OF LOOP]
Step 6: EXIT
```

Algorithm of DFS:

```
Step 1: SET STATUS=1 (ready state) for each node in G

Step 2: Push the starting nodeAon the stack and set

its STATUS=2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its

STATUS=3 (processed state)

Step 5: Push on the stack all the neighbours of N that

are in the ready state (whose STATUS=1) and

set their STATUS=2 (waiting state)

[END OF LOOP]

Step 6: EXIT
```

==

PROGRAM

==

```
//Header file delecrations

#include <stdio.h>

#include <stdlib.h>

#define MX 10

#define SIZE 50//Function Declaration/Signatures/Prototypes

int createGraph(int[][MX]);

void initAdjacencyMatrix(int[][MX]);

void showAdjacencyMatrix(int[][MX], int);

void setUpDFSBFS(int[][MX], int[], int);

void dfsAdjMat(int[][MX], int[], int, int);

void bfsAdjMat(int[][MX], int[], int, int);

void initQueue(int *, int *);

int isEmptyQ(int);
int isFullQ(int);

int insert(int[], int *, int *, int);

int delete (int[], int *, int *);

void displayQ(int[], int, int);
```

The driver Function:

```
int main()

{

    int graph[MX][MX], vertices;

    int visited[MX];

    int choice, source, i;


    initAdjacencyMatrix(graph);

    vertices = createGraph(graph);

    showAdjacencyMatrix(graph, vertices);


    do{

        printf("\n=======================================================\n");

        printf( "Choose the operation to perform on Graph:\n"

                "1. display()    2.depthFirstSearch() 3. breadthFirstSearch()\n"

                "0. Exit\n");

        scanf("%d", &choice);


        switch (choice){

            case 1:

                    showAdjacencyMatrix(graph, vertices);

                    break;


            case 2:

                    do{
                        printf("\n\t\tSource Vertex for DFS?? [0 thru %d]",
vertices - 1);

                        scanf("%d", &source);

                    } while (source > vertices - 1 || source < 0);


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

                        visited[i] = 0;


                    printf("\n\t\tDFS := ");

                    dfsAdjMat(graph, visited, source, vertices);


                    break;


            case 3: do{

                        printf("\n\t\tSource Vertex for BFS?? [0 thru %d]",
vertices - 1);

                        scanf("%d", &source);

                    } while (source > vertices - 1 || source < 0);

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

                        visited[i] = 0;

                    printf("\n\t\tBFS := ");

                    bfsAdjMat(graph, visited, source, vertices);

                    break;


            default: break;

        }

    }while(choice != 0);

    return 0;

}
```

Function Definitions:

```
int createGraph(int graph[][MX])

{

    int i, j, vCnt = 0;

    int u, v, vertices, type;


    printf("\n\tGraph Creation [Undirected/Directed]...\n");


    printf("\t\tType of Graph [0: UnDirected] := ");

    scanf("%d", &type);


    if (type != 0)

        type = 1;


    do

    {

        printf("\t\tHow many vertices [upto %d vertices]?? ", MX);

        scanf("%d", &vertices);

    } while (vertices < 1 || vertices > MX);


    printf("\n\t...\n");


    printf("\n\tVerices starts at 0 and terminates at %d\n", vertices - 1);

    printf("\t\tVertex ID of -1 terminates Input\n");

    printf("\n\tEnter Existing Edges in the Graph\n\n");

    printf("\t\t--------------------------------------\n");

    printf("\t\tEdge# uVertex# vVertex# Remark\n");

    printf("\t\t--------------------------------------\n");


    do
    {
        printf("\t\t  %2d       ", vCnt + 1);
        scanf("%d%d", &u, &v); //&cost
        printf("  \t   ");

        if ((u != -1 || v != -1) && u < vertices && v < vertices)
        {
            if (graph[u][v] == 0)
            {
                if (type)
                    graph[u][v] = 1;
                else
                    graph[u][v] = graph[v][u] = 1;

                printf("\t\t\t\t\t\tEdge Taken\n");
            }
            else
                printf("\t\t\t\t\t\tEdge Exists\n");
        }
        else
            printf("\t\t\t\t\t\tInvalid Edge\n");
        vCnt++;
    } while (u != -1 || v != -1);
    return vertices;
}

void initAdjacencyMatrix(int graph[][MX])
{
    int i, j;
    for (i = 0; i < MX; i++)
```

```
        for (j = 0; j < MX; j++)
            graph[i][j] = 0;
}

void showAdjacencyMatrix(int graph[][MX], int vertices)
{
    int i, j;

    printf("\n\tAdjacency Matrix is...\n\n");
    printf("\t\t u|v |");
    for (i = 0; i < vertices; i++)
        printf("%4d", i);
    printf("\n");

    printf("\t\t-------");
    for (i = 0; i < vertices; i++)
        printf("----");
    printf("\n");

    for (i = 0; i < vertices; i++)
    {
        printf("\t\t%4d  |", i);
        for (j = 0; j < vertices; j++)
            printf("%4d", graph[i][j]);
        printf("\n");
    }
    printf("\n");
}

void setUpDFSBFS(int graph[][MX], int visited[], int vertices)
```

```
{
    int i, source;
    for (i = 0; i < vertices; i++)
        visited[i] = 0;
    do
    {
        printf("\n\t\tSource Vertex for DFS/BFS?? [0 thru %d]", vertices - 1);
        scanf("%d", &source);
    } while (source > vertices - 1 || source < 0);

    printf("\n\t\tDFS := ");
    dfsAdjMat(graph, visited, source, vertices); // Recursive Algo.

    for (i = 0; i < vertices; i++)
        visited[i] = 0;
    printf("\n\t\tBFS := ");
    bfsAdjMat(graph, visited, source, vertices); // Iterative Algo.
}

void dfsAdjMat(int graph[][MX], int visited[], int source, int vertices)
{
    int vCount;
    printf("%4d", source);
    visited[source] = 1;

    for (vCount = 0; vCount < vertices; vCount++)
        if (!visited[vCount] && graph[source][vCount] == 1)
            dfsAdjMat(graph, visited, vCount, vertices);
}
```

```
void bfsAdjMat(int graph[][MX], int visited[], int source, int vertices)
{
    int vCount;
    int queue[SIZE], front, rear;
    initQueue(&front, &rear);

    printf("%4d", source);
    visited[source] = 1;
    insert(queue, &front, &rear, source);
    // displayQ(queue, front, rear);

    while (isEmptyQ(front) == 0)
    {
        source = delete (queue, &front, &rear);
        // displayQ(queue, front, rear);
        for (vCount = 0; vCount < vertices; vCount++)
            if (!visited[vCount] && graph[source][vCount] == 1)
            {
                printf("%4d", vCount);
                visited[vCount] = 1;
                insert(queue, &front, &rear, vCount);
                // displayQ(queue, front, rear);
            }
    }
}

void initQueue(int *front, int *rear)
{
    *front = -1;
    *rear = -1;
```

```
}

int isEmptyQ(int front)
{
    if (front == -1)
        return 1;
    return 0;
}

int isFullQ(int rear)
{
    if (rear == SIZE - 1)
        return 1;
    return 0;
}

int insert(int Q[], int *front, int *rear, int key)
{
    if (*rear == SIZE - 1)
        return 999999; // Full
    *rear = *rear + 1;
    Q[(*rear)] = key;
    if (*front == -1) // initial case when initQueue()
        *front = 0;
}

int delete (int Q[], int *front, int *rear)
{
    int key;
    if (*front == -1)
```

```
        return -999999; // Empty
    key = Q[(*front)];
    if (*front == *rear)
        *front = *rear = -1;
    else
        *front = *front + 1;
    return key;
}

void displayQ(int Q[], int front, int rear)
{
    int i;
    if (rear != -1)
    {
        printf("\n\tThe Queue Contents are..\n");
        printf("\t\tFRONT: [%d]-->", front);

        for (i = front; i <= rear; i++)
            printf("%4d", Q[i]);
        printf(" -->[%d]:REAR\n", rear);
    }
    else
        printf("\n\tQueue is Empty....\n");
}
```

```
================================================================================
```

EXECUTION-TRAIL

```
================================================================================

Graph Creation [Undirected/Directed]...

                Type of Graph [0: UnDirected] := 0

                How many vertices [upto 10 vertices]?? 6


        ...


        Verices starts at 0 and terminates at 5

                Vertex ID of -1 terminates Input


        Enter Existing Edges in the Graph


                --------------------------------------

                Edge# uVertex# vVertex# Remark

                --------------------------------------

                   1       1

2

                                        Edge Taken

                   2       2

3

                                        Edge Taken

                   3       3

4

                                        Edge Taken

                   4       4

5

                                        Edge Taken

                   5       5
```

```
6
                                                        Invalid Edge
                   6       6
-1
                                                        Invalid Edge
                   7       -1
-1
                                                        Invalid Edge

        Adjacency Matrix is...

                 u|v |   0   1   2   3   4   5
                 -------------------------------
                   0  |   0   0   0   0   0   0
                   1  |   0   0   1   0   0   0
                   2  |   0   1   0   1   0   0
                   3  |   0   0   1   0   1   0
                   4  |   0   0   0   1   0   1
                   5  |   0   0   0   0   1   0


=======================================================
Choose the operation to perform on Graph:
1. display()    2.depthFirstSearch() 3. breadthFirstSearch()
0. Exit
1

        Adjacency Matrix is...

                 u|v |   0   1   2   3   4   5
```

```
                ------------------------------
                   0 |   0   0   0   0   0   0
                   1 |   0   0   1   0   0   0
                   2 |   0   1   0   1   0   0
                   3 |   0   0   1   0   1   0
                   4 |   0   0   0   1   0   1
                   5 |   0   0   0   0   1   0


=======================================================
Choose the operation to perform on Graph:
1. display()    2.depthFirstSearch() 3. breadthFirstSearch()
0. Exit
2

                Source Vertex for DFS?? [0 thru 5]5

                DFS :=    5   4   3   2   1
=======================================================
Choose the operation to perform on Graph:
1. display()    2.depthFirstSearch() 3. breadthFirstSearch()
0. Exit
3

                Source Vertex for BFS?? [0 thru 5]4

                BFS :=    4   3   5   2   1
=======================================================
Choose the operation to perform on Graph:
1. display()    2.depthFirstSearch() 3. breadthFirstSearch()
```

```
0. Exit

1


        Adjacency Matrix is...


                 u|v |   0   1   2   3   4   5

                ------------------------------

                   0  |   0   0   0   0   0   0

                   1  |   0   0   1   0   0   0

                   2  |   0   1   0   1   0   0

                   3  |   0   0   1   0   1   0

                   4  |   0   0   0   1   0   1

                   5  |   0   0   0   0   1   0



=======================================================

Choose the operation to perform on Graph:

1. display()    2.depthFirstSearch() 3. breadthFirstSearch()

0. Exit

0


Graph Creation [Undirected/Directed]...

                Type of Graph [0: UnDirected] := 1

                How many vertices [upto 10 vertices]?? 5


        ...


        Verices starts at 0 and terminates at 4

                Vertex ID of -1 terminates Input
        Enter Existing Edges in the Graph

                ------------------------------------
                Edge# uVertex# vVertex# Remark
                ------------------------------------
                   1       1
1
                                                        Edge Taken
                   2       2
2
                                                        Edge Taken
                   3       3
3
                                                        Edge Taken
                   4       4
4
                                                        Edge Taken
                   5       5
5
                                                        Invalid Edge
                   6       -1
-1
                                                        Invalid Edge

        Adjacency Matrix is...

                 u|v |   0   1   2   3   4
                --------------------------
                   0  |   0   0   0   0   0
                   1  |   0   1   0   0   0
```

```
                  2 |   0   0   1   0   0

                  3 |   0   0   0   1   0

                  4 |   0   0   0   0   1



=======================================================

Choose the operation to perform on Graph:

1. display()    2.depthFirstSearch() 3. breadthFirstSearch()

0. Exit

1


        Adjacency Matrix is...


                 u|v |   0   1   2   3   4

                --------------------------

                  0 |   0   0   0   0   0

                  1 |   0   1   0   0   0

                  2 |   0   0   1   0   0

                  3 |   0   0   0   1   0

                  4 |   0   0   0   0   1



=======================================================

Choose the operation to perform on Graph:

1. display()    2.depthFirstSearch() 3. breadthFirstSearch()

0. Exit

3


                Source Vertex for BFS?? [0 thru 4]1
                BFS :=    1
=======================================================
Choose the operation to perform on Graph:
1. display()    2.depthFirstSearch() 3. breadthFirstSearch()
0. Exit
2

                Source Vertex for DFS?? [0 thru 4]1

                DFS :=    1
=======================================================
Choose the operation to perform on Graph:
1. display()    2.depthFirstSearch() 3. breadthFirstSearch()
0. Exit
1

        Adjacency Matrix is...

                 u|v |   0   1   2   3   4
                ---------------------------
                   0  |   0   0   0   0   0
                   1  |   0   1   0   0   0
                   2  |   0   0   1   0   0
                   3  |   0   0   0   1   0
                   4  |   0   0   0   0   1
=======================================================
Choose the operation to perform on Graph:
1. display()    2.depthFirstSearch() 3. breadthFirstSearch()
0. Exit
0
```

==

VIVA-VOICE

==

1.Enlist advantages and limitations of the adjacency matrix.

Ans: Advantages of Adjacency Matrix Representation

● We can determine if two vertices are adjacent to each other in constant

time.

● We can add an edge in the graph in constant time

● We can delete an edge from the graph in constant time.

The most important disadvantage of the adjacency matrix representation is that it requires n2 storage, even if the graph has as few as O(n) edges.Traversing the graph using algorithms like DPS/BFS requires O(V2) time in case of an adjacency matrix whereas we can traverse the graph in O(V+E) time.

2. Enlist advantages and limitations of the adjacency list.

Ans: Adjacency lists, on the other hand, are a great option when we need to continuously access all the neighbors of some node u. In that case, we'll only be iterating over the needed nodes. Adjacency list limitations show when we need to check if two nodes have a direct edge or not.

3. List the data structures used in DFS and BFS.

Ans: DFS(Depth First Search) uses Stack data structure.BFS(Breadth First Search) uses Queue data structure for finding the shortest path.

4. Elaborate on self-loop,parallel edges,and a digraph.

Ans: A self-loop or loop is an edge between a vertex and itself. An undirected.A graph without loops or multiple edges is known as a simple graph. In this class we will assume graphs to be simple unless otherwise stated. In graph theory,parallel edges are, in an undirected graph, two or more edges that are incident to the same two vertices, or in a directed graph, two or more edges with both the same tail vertex and the same head vertex. A simple the graph has no multiple edges and no loops. Simply put, digraphs are multi-character sequences treated by the C preprocessor and/or compiler as other (normally single-character) sequences.

==

CONCLUSION

==

In this experiment we have studied what is graph and its various operations. We have also learnt how to write the algorithms for operations of graphs. Sir has explained us in detail what is bfs and dfs and its uses. Sir also explained us algorithm for bfs and dfs in all cases.