Name	Shubhan Singh
UID no.	2022300118
Experiment No.	8

Program 1	
PROBLEM STATEMENT :	-> Implement BFS & DFS traversal for graphs. -> You need to make use of an adjacency matrix for representing the graph
ALGORITHM:	1. **init_graph** function: • Initialize a new graph with the given number of vertices and directed/undirected status. • Allocate memory for the adjacency matrix, distance, color, predecessor, and finish arrays. • Initialize adjacency matrix with zeros (no edges). • Initialize distance to -1 (indicating unconnected nodes), color to WHITE, predecessor to INT_MAX, and finish to NULL (not needed for BFS). • Set the number of vertices, number of edges, and graph type. • Return the initialized graph. 2. **insert_edge** function: • Check if the edge e is not already in the adjacency matrix (e.g., **graph->edges[e.u][e.v] == 0**). • If not, mark the edge as present (set to 1 in the adjacency matrix). • If the graph is undirected, also mark the reverse edge as present (bi-directional). • Update the edge count in the graph. 3. **remove_edge** function: • Check if the edge e is present in the adjacency matrix (e.g., **graph->edges[e.u][e.v] != 0**). • If it is present, mark the edge as removed (set to 0). • If the graph is undirected, also remove the reverse edge.

- Update the edge count in the graph.

4. **traverse_bfs** function:
    - Initialize BFS traversal attributes: source, type, color, distance, and predecessor.
    - Create a queue for BFS traversal.
    - Enqueue the source node and set its color to GRAY.
    - While the queue is not empty:
        - Dequeue a node (current vertex).
        - Process adjacent vertices and enqueue unvisited neighbors:
            - Set the color of unvisited neighbors to GRAY.
            - Update distance (hops) from the source.
            - Set the predecessor for each neighbor.
            - Enqueue unvisited neighbors.
        - Mark the current vertex as BLACK when all adjacent vertices are explored.
    - Print the BFS traversal path.
5. **traverse_dfs** function:
    - Initialize DFS traversal attributes: source, type, finish, and time.
    - Create an array to track the finish times of vertices.
    - Print the start of the DFS traversal.
    - Call the **dfs_visit** function for the source vertex and explore its neighbors.
    - After the first DFS traversal, explore any unvisited nodes in the graph.
    - Print the DFS traversal path.
6. **dfs_visit** function:
    - Recursive DFS traversal function.
    - Update the distance (time) of the current vertex.
    - Set the color of the current vertex to GRAY.
    - Print the current vertex.
    - Explore adjacent vertices:
        - If an adjacent vertex is unvisited (WHITE), set its predecessor and recursively call

dfs_visit.

- Update the finish time (time) of the current vertex.
- Mark the current vertex as BLACK.

7. **display_path** function:
   - Check if the destination node is reachable from the source.
   - Determine the traversal type (BFS or DFS).
   - Handle error cases where the destination is invalid or the same as the source.
   - Backtrack from the destination to the source using the predecessor array.
   - Print the path from source to destination.
8. **display_graph** function:
   - Display the graph in matrix form by iterating through the adjacency matrix and printing the values.

PROGRAM:

```c
// Implement BFS & DFS traversal for graphs.
// You need to make use of an adjacency matrix for representing the graph
// The structure below should allow you to handle both directed and
undirected graphs.
// For traversals, the respective function should accept starting node for
traversal and perform traversal (BFS/ DFS).

#include "queue.c"

GraphRep *init_graph(int num_vertices, bool is_directed){
    GraphRep* newgraph=malloc(sizeof(GraphRep));
    newgraph->edges=malloc(num_vertices*sizeof(bool*));
    newgraph->distance=malloc(num_vertices*sizeof(int));
    newgraph->color=malloc(num_vertices*sizeof(Color));
    newgraph->predecessor=malloc(num_vertices*sizeof(Vertex));
    newgraph->finish=NULL;//Finish is not initialised here itself as it is
only needed for DFS
    for(int i=0;i<num_vertices;i++){
        newgraph->edges[i]=calloc(num_vertices,sizeof(bool));//initialise
all positions in adjacency matrix to 0
        newgraph->distance[i]=-1;//distance==-1 -> node not connected
        newgraph->color[i]=WHITE;//initialise all nodes to white
        newgraph->predecessor[i]=INT_MAX;//predecessor>nV means no
predecessor
    }
    newgraph->nV=num_vertices;
    newgraph->nE=0;
    newgraph->is_directed=is_directed;
    return newgraph;
}

void insert_edge(GraphRep *graph, Edge e){
    if(graph->edges[e.u][e.v]==0){
        graph->edges[e.u][e.v]=1;
        if(!graph->is_directed){
            graph->edges[e.v][e.u]=1;//add two ones to adjacency matrix only
when graph is undirected
        }
        graph->nE++;
    }
}

void remove_edge(GraphRep *graph, Edge e){
    if(graph->edges[e.u][e.v]!=0){
        graph->edges[e.u][e.v]=0;
        if(!graph->is_directed){
            graph->edges[e.v][e.u]=0;//have to remove bith ones when graph
is undirected
        }
        graph->nE--;
    }
}


// NOTE: During both DFS and BFS traversals, when at a vertex that is
connected with multiple vertices, always pick the connecting vertex which
has the lowest value first
// Both traversals will always update the following attributes of the Graph:
// 1. source -> stores the value of the starting vertex for the traversal
// 2. type -> stores the traversal type (BFS or DFS)
// 3. color --> indicates if all vertices have been visited or not. Post
traversal, all vertices should be BLACK
// 4. predecessor --> this array would hold the predecessor for a given
vertex (indicated by the array index).
```

```
// NOTE: BFS Traversal should additionally update the following in the
graph:
// 1. distance --> this array would hold the number of hops it takes to
reach a given vertex (indicated by the array index) from the source.
void traverse_bfs(GraphRep *graph, Vertex source){
    graph->source=source;
    int bfsarr[graph->nV];
    graph->type=BFS;
    graph->color[source]=GRAY;
    graph->distance[source]=0;
    graph->predecessor[source]=INT_MAX;/*setting predecessor of source as
null, since we can't use null with integers, we are using a large number
instead,which is unlikely to be the an index*/
    Queue* bfsqueue=initialize_queue(graph->nV);
    Vertex temp;
    int index=0;
    enqueue(bfsqueue,source);
    while(!isEmpty(bfsqueue)){
        temp=dequeue(bfsqueue);//dequeue a node, this is our current grey
vertex
        bfsarr[index]=temp;
        index++;
        for(int i=0;i<graph->nV;i++){
            if(graph->edges[temp][i]==1){
```

```c
                if(graph->color[i]==WHITE){//process all adjacent vertices
and enqueue them
                    graph->color[i]=GRAY;
                    graph->distance[i]=graph->distance[temp]+1;
                    graph->predecessor[i]=temp;
                    enqueue(bfsqueue,i);
                }
            }
        }
        graph->color[temp]=BLACK;//paint the current vortex black as we have
explored all adjacent vertices
    }
    printf("The BFS traversal of the given graph originating from vertex %d
is\n[ ",source);
    for(int i=0;i<index;i++){
        printf("%d ",bfsarr[i]);//printing the BFS traversal
    }
    printf("]\n");
}

void dfs_visit(GraphRep* graph,Vertex source, int *time);
// NOTE: DFS Traversal should additionally update the following in the
graph:
// 1. distance --> Assuming 1 hop to equal 1 time unit, this array would
hold the time of discovery a given vertex (indicated by the array index)
```

```c
from the source.
// 2. finish --> Assuming 1 hop to equal 1 time unit, this array would hold
the time at which exploration concludes for a given vertex (indicated by the
array index).
void traverse_dfs(GraphRep *graph, Vertex source){
    graph->source=source;
    graph->type=DFS;
    graph->finish=calloc(graph->nV,sizeof(int));
    int time=0;
    printf("The DFS traversal of the given graph originating from vertex %d
is \n[ ",source);
    dfs_visit(graph,source,&time);/*since there is a source node specified
for the DFS function, we run DFS for that first, an then do it for any Ehite
nodes left */
    for(int i=0;i<graph->nV;i++){//doing DS for any unvisited nodes
        if(graph->color[i]==WHITE){
            dfs_visit(graph,i,&time);
        }
    }
    printf("]\n");
}

void dfs_visit(GraphRep* graph,Vertex source, int *time){/*this is a helper
function for the traverse_dfs function, with only those parts of the code
which need to be executed recursively*/
// Recursive DFS traversal function
// Update attributes for the current vertex and explore adjacent nodes
    graph->distance[source]=*time;
    (*time)++;
    graph->color[source]=GRAY;
    printf("%d ",source);
    for(int i=0;i<graph->nV;i++){
        if(graph->edges[source][i]==1){
            if(graph->color[i]==WHITE){//Explore adjacent unexplored nodes
first
                graph->predecessor[i]=source;
                dfs_visit(graph,i,time);
            }
        }
    }
    graph->finish[source]=*time; // Record the time of completion and mark
the vertex as BLACK
    (*time)++;
    graph->color[source]=BLACK;
}

// displays the path from the current 'source' in graph to the provided
'destination'.
// The graph holds the value of the traversal type, so the function should
let the caller know what kind of traversal was done on the graph and from
```

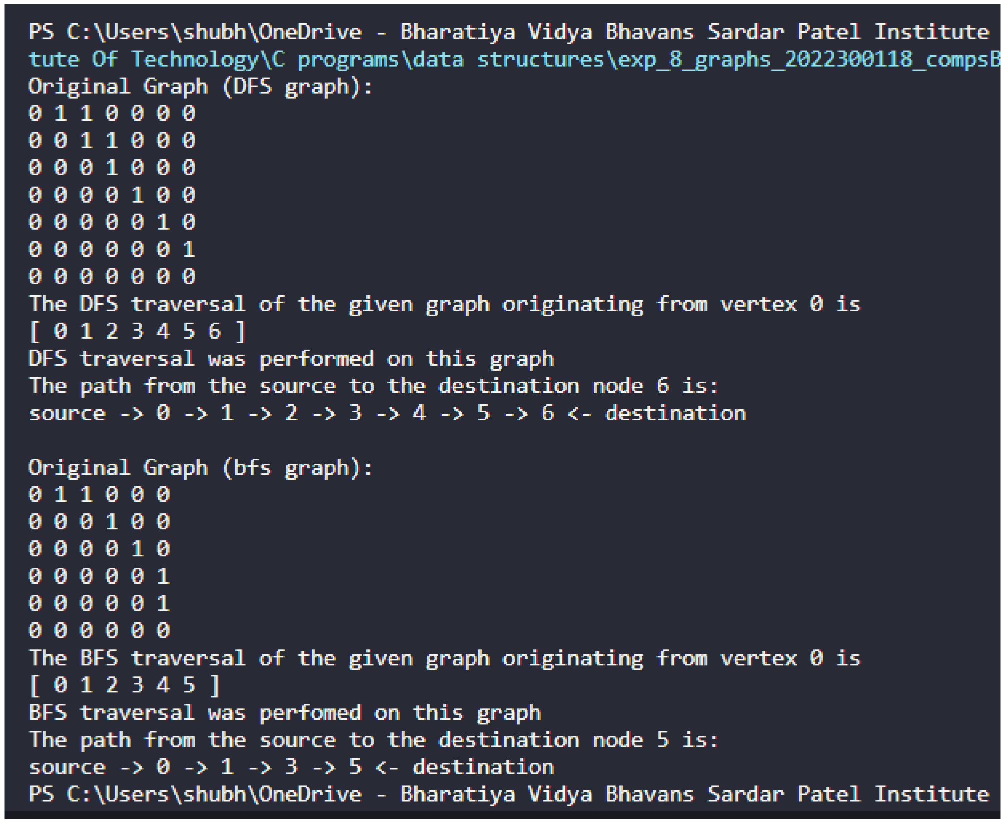
```c
which vertex, along with the path.
void display_path(GraphRep *graph, Vertex destination){
    if(graph->distance[destination]<0){
        printf("This node is unreachable from the source vortex\n");
        return;
    }
    if(graph->type==BFS){
        printf("BFS traversal was perfomed on this graph\n");
    }
    else{
        printf("DFS traversal was performed on this graph\n");
    }
    if(destination<0 || destination>graph->nV || graph-
>predecessor[destination]>graph->nV){
        if(destination==graph->source){
            printf("The destination is same as the source vortex");
        }
        else{
            printf("The destination is invalid!\n");
        }
    }//error cases
    int arr[graph->nV];
    int index=0;
    int current=destination;
    while(true){/*we just backtrack by following the predecessor array for a
```

```c
chain of nodes from the destination to the source*/
        arr[index]=current;
        current=graph->predecessor[current];
        if(current>graph->nV){
            break;
        }
        index++;
    }
    printf("The path from the source to the destination node %d
is:\n",destination);
    printf("source -> ");
    for(int i=index;i>=0;i--){
        if(i>0){
            printf("%d -> ",arr[i]);
        }
        else{
            printf("%d <- destination\n",arr[i]);
        }
    }
}


// display the graph in the matrix form
void display_graph(GraphRep *graph){
    for(int i=0;i<graph->nV;i++){
        for(int j=0;j<graph->nV;j++){//printing the 2d adjacency matrix
            printf("%d ",graph->edges[i][j]);
        }
        printf("\n");
    }
}


int main() {
    GraphRep *graph = init_graph(7, true);
    Edge edges[] = {{0, 1}, {0, 2}, {1, 2}, {1, 3}, {2,3}, {3, 4}, {4, 5},
{5, 6}};
    for (int i = 0; i < sizeof(edges) /sizeof(edges[0]); i++) {
        insert_edge(graph, edges[i]);
    }
    printf("Original Graph (DFS graph):\n");
    display_graph(graph);
    traverse_dfs(graph, 0);
    display_path(graph, 6);
    GraphRep *graph2= init_graph(6, true);
    Edge edges2[] = {{0, 2}, {1, 3}, {2, 4}, {3, 5}, {4,5}, {0, 1}};
    for (int i = 0; i < sizeof(edges) /sizeof(edges[0]); i++) {
        insert_edge(graph2, edges2[i]);
    }
    printf("\nOriginal Graph (bfs graph):\n");
    display_graph(graph2);
    traverse_bfs(graph2, 0);
    display_path(graph2, 5);
return 0;
}
```

RESULT:



```
PS C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute
tute Of Technology\C programs\data structures\exp_8_graphs_2022300118_compsB
Original Graph (DFS graph):
0 1 1 0 0 0 0
0 0 1 1 0 0 0
0 0 0 1 0 0 0
0 0 0 0 1 0 0
0 0 0 0 0 1 0
0 0 0 0 0 0 1
0 0 0 0 0 0 0
The DFS traversal of the given graph originating from vertex 0 is
[ 0 1 2 3 4 5 6 ]
DFS traversal was performed on this graph
The path from the source to the destination node 6 is:
source -> 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 <- destination

Original Graph (bfs graph):
0 1 1 0 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1
0 0 0 0 0 1
0 0 0 0 0 0
The BFS traversal of the given graph originating from vertex 0 is
[ 0 1 2 3 4 5 ]
BFS traversal was perfomed on this graph
The path from the source to the destination node 5 is:
source -> 0 -> 1 -> 3 -> 5 <- destination
PS C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute
```

THEORY:	**Graph Data Structure:** A graph is a data structure used in computer science and mathematics to represent relationships and connections between objects. It consists of a set of vertices (nodes) and a set of edges that connect these vertices. Graphs can be used to model a wide range of real-world scenarios, making them a versatile and powerful data structure. Graphs can be categorized into two main types: 1. **Directed Graph (Digraph):** In a directed graph, edges have a direction, indicating a one-way relationship between vertices. Each edge points from one vertex to another. Directed graphs are used to represent situations where there is a clear direction of influence or flow. 2. **Undirected Graph:** In an undirected graph, edges have no direction, and relationships between vertices are symmetric. If there is an edge between vertices A and B, it means there is a connection from A to B and from B to A. Undirected graphs are suitable for modeling mutual relationships. **Key Concepts in Graphs:** • **Vertex (Node):** A fundamental unit in a graph that represents an entity or object. Vertices can be connected by edges. • **Edge:** A connection between two vertices in a graph. Edges can have attributes such as weight, cost, or direction, depending on the type of graph. • **Adjacency:** Two vertices are said to be adjacent if there is an edge connecting them. • **Path:** A path is a sequence of vertices where each vertex is connected to the next by an edge. The length of a path is the number of edges it contains. • **Cycle:** A cycle is a path that starts and ends at the same vertex, without revisiting any other vertex in the path. • **Connected Graph:** A graph is connected if there is a path between every pair of vertices. In a directed graph, there are different levels of connectedness, such as weakly connected and strongly connected. **Breadth-First Search (BFS):** Breadth-First Search is a graph traversal algorithm used to explore or search through all the vertices of a graph systematically. It starts at a given source vertex and explores its neighbors before moving on to their neighbors. BFS uses a queue data structure to keep track of the vertices to visit. Key points about BFS: • It is used to find the shortest path in unweighted graphs.

	<ul><li>It ensures that all vertices at a given distance from the source are explored before moving to vertices at a greater distance.</li><li>BFS is well-suited for tasks like finding connected components, determining reachability, and solving puzzles.</li></ul>**Depth-First Search (DFS):** Depth-First Search is another graph traversal algorithm that explores as far as possible along a branch before backtracking. It starts at a source vertex and explores one branch of the graph to its deepest level before returning and exploring other branches. DFS is often implemented using recursion or a stack. Key points about DFS:<ul><li>It can be used to find paths, cycles, and connected components.</li><li>DFS is not necessarily the most efficient way to find the shortest path, but it is useful for exploring all paths in a graph.</li><li>Recursive DFS can be implemented using function calls to explore the graph.</li></ul>
CONCLUSION:	graphs are versatile data structures used to model various relationships and dependencies. BFS and DFS are essential algorithms for traversing and analyzing graphs, with different strengths and use cases. BFS is suitable for tasks involving the shortest path and exploring nearby vertices, while DFS is more focused on exploration of paths, cycles, and connected components. BFS is implemented with the help of a queue, while DFS can be implemented with the help of a stack or with recursion.