Experiment No.10
Implementation of Graph traversal techniques - Depth First Search, Breadth First Search
Name: Shravani Sandeep Raut
Roll No: 48
Date of Performance:
Date of Submission:

Experiment No. 10: Depth First Search and Breath First Search

Aim : Implementation of DFS and BFS traversal of graph.

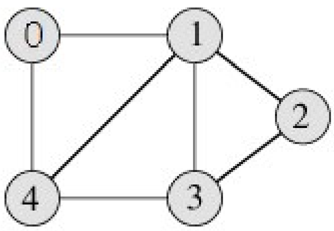
Objective:

1. Understand the Graph data structure and its basic operations.
2. Understand the method of representing a graph.
3. Understand the method of constructing the Graph ADT and defining its operations

Theory:

A graph is a collection of nodes or vertices, connected in pairs by lines referred to as edges. A graph can be directed or undirected.

One method of traversing through nodes is depth first search. Here we traverse from the starting node and proceed from top to bottom. At a moment we reach a dead end from where the further movement is not possible and we backtrack and then proceed according to left right order. A stack is used to keep track of a visited node which helps in backtracking.



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

DFS Traversal –0 1 2 3 4

Algorithm

Algorithm: DFS_LL(V)

Input: V is a starting vertex

Output : A list VISIT giving order of visited vertices during traversal.

Description: linked structure of graph with gptr as pointer

1. if gptr = NULL then
   print "Graph is empty" exit
2. u=v
3. OPEN.PUSH(u)
4. while OPEN.TOP !=NULL do
   u=OPEN.POP()
   if search(VISIT,u) = FALSE then
      INSERT_END(VISIT,u)
      Ptr = gptr(u)

While ptr.LINK != NULL do

Vptr = ptr.LINK

OPEN.PUSH(vptr.LABEL)

End while

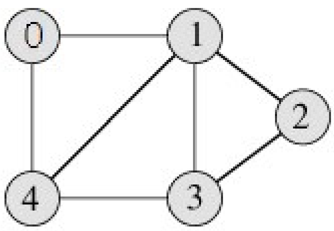
End if

End while

5. Return VISIT

6. Stop

BFS Traversal



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

BFS Traversal – 0 1 4 2 3

Algorithm

Algorithm: DFS()

i=0

count=1

visited[i]=1

print("Visited vertex i")

repeat this till queue is empty or all nodes visited

repeat this for all nodes from first till last

if(g[i][j]!=0&&visited[j]!=1)

{

push(j)

}

i=pop()

print("Visited vertex i")

visited[i]=1

count++

Algorithm: BFS()

i=0

count=1

visited[i]=1

print("Visited vertex i")

repeat this till queue is empty or all nodes visited

repeat this for all nodes from first till last

if(g[i][j]!=0&&visited[j]!=1)

{

enqueue(j)

}

i=dequeue()

print("Visited vertex i")

visited[i]=1

count++

Code:

```
//DFS

#include <stdio.h>
#define MAX_VERTICES 100

void DFS(int graph[MAX_VERTICES][MAX_VERTICES], int
visited[MAX_VERTICES], int vertices, int start)
{
   printf("%d ", start);
   visited[start] = 1;

   for (int i = 0; i < vertices; i++)
   {
      if (graph[start][i] == 1 && !visited[i])
      {
         DFS(graph, visited, vertices, i);
      }
   }
```

```
}

int main()
{
   int vertices, edges;
   printf("Enter the number of vertices: ");
   scanf("%d", &vertices);

   if (vertices <= 0 || vertices > MAX_VERTICES) {
      printf("Invalid number of vertices. Exiting..\n");
      return 1;
   }

   int graph[MAX_VERTICES][MAX_VERTICES] = {0};
   int visited[MAX_VERTICES] = {0};

   printf("Enter the number of edges: ");
   scanf("%d", &edges);

   if (edges < 0 || edges > vertices * (vertices - 1))
   {
      printf("Invalid number of edges. Exiting...\n");
      return 1;
   }

   for (int i = 0; i < edges; i++)
   {
      int start, end;
      printf("Enter edge %d (start end): ", i + 1);
      scanf("%d %d", &start, &end);

      if (start < 0 || start >= vertices || end < 0 || end >= vertices)
      {
         printf("Invalid vertices. Try again.\n");
         i--;
         continue;
      }

      graph[start][end] = 1;
      }

   int startVertex;
   printf("Enter the starting vertex for DFS traversal:");
   scanf("%d", &startVertex);

   if (startVertex < 0 || startVertex >= vertices) {
      printf("Invalid starting vertex. Exiting...\n");
      return 1;
   }
```

```
    printf("DFS Traversal Order: ");
    DFS(graph, visited, vertices, startVertex);
    return 0;
}



//BFS
#include <stdio.h>
#include <stdlib.h>
#define MAX_VERTICES 100

void addEdge(int graph[MAX_VERTICES][MAX_VERTICES], int start, int end)
{
    graph[start][end] = 1;
    graph[end][start] = 1; // For undirected graph
}


void BFS(int graph[MAX_VERTICES][MAX_VERTICES], int vertices, int startVertex)
{
    int visited[MAX_VERTICES] = {0};
    int queue[MAX_VERTICES];
    int front = -1, rear = -1;

    visited[startVertex] = 1;
    queue[++rear] = startVertex;

    printf("BFS Traversal Order: ");
    while (front != rear)
    {
        int currentVertex = queue[++front];
        printf("%d ", currentVertex);
        for (int i = 0; i < vertices; i++)
        {
            if (graph[currentVertex][i] == 1 && !visited[i])
            {
                visited[i] = 1;
                queue[++rear] = i;
            }
        }
    }
    printf("\n");
}

int main() {
    int vertices, edges;
```

```
    // Input the number of vertices
    printf("Input the number of vertices: ");
    scanf("%d", &vertices);

    if (vertices <= 0 || vertices > MAX_VERTICES) {
        printf("Invalid number of vertices. Exiting...\n");
        return 1;
    }

    int graph[MAX_VERTICES][MAX_VERTICES] = {0}; // Initialize the adjacency
matrix with zeros

    // Input the number of edges
    printf("Input the number of edges: ");
    scanf("%d", &edges);

    if (edges < 0 || edges > vertices * (vertices - 1) / 2) {
        printf("Invalid number of edges. Exiting...\n");
        return 1;
    }

    // Input edges and construct the adjacency matrix
    for (int i = 0; i < edges; i++) {
        int start, end;
        printf("Input edge %d (start end): ", i + 1);
        scanf("%d %d", &start, &end);

        // Validate input vertices
        if (start < 0 || start >= vertices || end < 0 || end >= vertices) {
            printf("Invalid vertices. Try again.\n");
            i--;
            continue;
        }

        addEdge(graph, start, end);
    }

    int startVertex;
    printf("Input the starting vertex for BFS traversal");
    scanf("%d", &startVertex);
    BFS(graph, vertices, startVertex);
    return 0;
}
```

Output:

```
Enter the number of vertices: 5
Enter the number of edges: 6
Enter edge 1 (start end): 0 1
Enter edge 2 (start end): 1 2
Enter edge 3 (start end): 2 3
Enter edge 4 (start end): 3 4
Enter edge 5 (start end): 4 0
Enter edge 6 (start end): 2 4
Enter the starting vertex for DFS traversal: 2
DFS Traversal Order: 2 3 4 0 1

=== Code Execution Successful ===
```

```
Input the number of vertices: 5
Input the number of edges: 6
Input edge 1 (start end): 0 1
Input edge 2 (start end): 1 2
Input edge 3 (start end): 2 3
Input edge 4 (start end): 3 4
Input edge 5 (start end): 4 0
Input edge 6 (start end): 2 4
Input the starting vertex for BFS traversal: 0
BFS Traversal Order: 0 1 4 2 3


=== Code Execution Successful ===
```

Conclusion:

1) Write the graph representation used by your program and explain why you choose that.

In this question we have used Adjacency Matrix. A graph can be represented in multiple ways, and one common method is using an **adjacency matrix**.

An adjacency matrix is a 2D array (matrix) used to represent a graph. The rows and columns represent the vertices of the graph, and the cells of the matrix represent the edges

- If there is an edge between vertex iii and vertex jjj, the cell matrix[i][j]matrix[i][j]matrix[i][j] will contain a value (usually 1 for unweighted graphs or the weight of the edge for weighted graphs).
- If there is no edge between vertex iii and vertex jjj, matrix[i][j]matrix[i][j]matrix[i][j] will contain 0 (or infinity for weighted graphs without an edge).

2) Write the applications of BFS and DFS other than finding connected nodes and explain how it is attained?

Applications of BFS (Breadth-First Search):

1. **Shortest Path in Unweighted Graphs**: BFS finds the shortest path from a source node to any other node in an unweighted graph by visiting all neighbors level by level, ensuring that the first time a node is reached, it is through the shortest path.
2. **Level-wise Traversal**: BFS is used in tree structures to traverse each level one by one, such as in breadth-first search of a binary tree, ensuring nodes at each depth are processed together.
3. **Web Crawlers**: BFS can be used to traverse web pages starting from a given URL, crawling all directly linked pages, then their links, and so on, level by level.

Applications of DFS (Depth-First Search):

1. **Topological Sorting**: In Directed Acyclic Graphs (DAGs), DFS is used to order vertices linearly, such that for every directed edge u→v, vertex u comes before v. This is achieved by exploring each node deeply and pushing it onto a stack after its descendants are fully explored.
2. **Cycle Detection**: DFS helps detect cycles in directed and undirected graphs by keeping track of visited nodes and back edges. If a back edge is found during DFS traversal, a cycle exists.
3. **Path Finding in Mazes**: DFS can explore paths in mazes, going deep along each possible route until a solution is found, backtracking if a dead-end is reached.

Both BFS and DFS are applied by recursively or iteratively exploring nodes and keeping track of visited states to ensure efficient traversal without revisiting nodes unnecessarily.