Experiment No.10
Aim: Implementation of DFS and BFS traversal of graph.
Name:Nikhil Kamalaji Shingade
Roll no: 57
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.

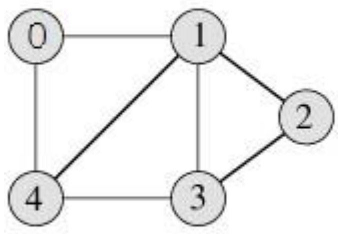
Objective:

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

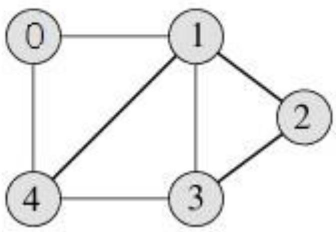
End if

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+

+

Algorit

hm:

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:

Implementation of DFS

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

// Adjacency Matrix representation
int adj[MAX][MAX];
int visited[MAX];
int n; // Number of vertices
```

```
void DFS(int vertex)
{
    printf("%d ", vertex);
    visited[vertex] = 1;

    for (int i = 0; i < n; i++)
{
        if (adj[vertex][i] == 1 && !visited[i])
 {
            DFS(i);
        }
    }
}

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

    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &adj[i][j]);
        }
    }

    for (int i = 0; i < n; i++) {
        visited[i] = 0;
    }

    printf("DFS traversal starting from vertex 0:\n");
    DFS(0);

    return 0;
}
```

Output

```
/tmp/8lKGv8xSGz.o
Enter the number of vertices: 4
Enter the adjacency matrix:
1 0 1 0
0 1 1 0
1 1 0 1
0 1 0 0
DFS traversal starting from vertex 0:
0 2 1 3

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

Implementation of BFS

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

// Adjacency Matrix representation
int adj[MAX][MAX];
int visited[MAX];
int n; // Number of vertices

void BFS(int startVertex)
{
   int queue[MAX], front = -1, rear = -1;
   int i;

   printf("%d ", startVertex);
   visited[startVertex] = 1;
   rear++;
   queue[rear] = startVertex;

   while (front != rear)
 {
      front++;
      int currentVertex = queue[front];

      for (i = 0; i < n; i++)
 {
         if (adj[currentVertex][i] == 1 && !visited[i]) {
            printf("%d ", i);
            visited[i] = 1;
            rear++;
```

```
                queue[rear] = i;
            }
        }
    }
}

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

    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < n; i++)
{
        for (int j = 0; j < n; j++)
 {
            scanf("%d", &adj[i][j]);
        }
    }

    for (int i = 0; i < n; i++)
 {
        visited[i] = 0;
    }

    printf("BFS traversal starting from vertex 0:\n");
    BFS(0);

    return 0;
}
```

Output

```
/tmp/URB2fpcunV.o
Enter the number of vertices: 4
Enter the adjacency matrix:
1 0 1 0
0 1 1 0
1 1 0 1
0 1 0 0
BFS traversal starting from vertex 0:
0 2 1 3

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

Conclusion:

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

ANS

For a graph with 4 vertices (0, 1, 2, 3), an adjacency matrix looks like this:

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

1. **Simplicity**:
    - Easy to implement and understand.
    - Each row and column represent vertices, and the value at matrix[i][j] shows i f there is an edge between vertices i and j.
2. **Constant Time Access**:
    - Checking for the presence of an edge between two vertices is O(1), making i t efficient for dense graphs where many vertices are connected.
3. **Memory Usage**:

- For dense graphs, where the number of edges is close to the number of vertices squa red, the adjacency matrix efficiently utilizes memory

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

⇒ Space Efficiency:

- Adjacency lists are more space-efficient than adjacency matrices, especially for sparse graphs. In an adjacency matrix, memory is allocated for every possible edge, leading to a potentially significant amount of unused space. In contrast, an adjacency list only stores edges that exist.

Ease of Traversal:

- Traversing neighbors is straightforward with an adjacency list. When a vertex is accessed, its list of adjacent vertices can be easily iterated over, making algorithms like DFS and BFS efficient.