Experiment No.9
Implementation of Graph traversal techniques - Depth First Search, Breadth First Search
Name:Ayush Gupta
Roll No:14
Date of Performance:
Date of Submission:
Marks:
Sign:

Experiment No. 9: Depth First Search and Breath First Search

Aim : Implementation of DFS and BFS traversal of graph.

Objective:

Understand the Graph data structure and its basic operations.

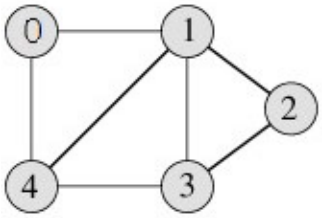
Understand the method of representing a graph.

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

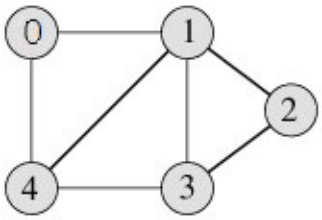
Output : A list VISIT giving order of visited vertices during traversal.

Description: linked structure of graph with gptr as pointer

```
if gptr = NULL then
        print "Graph is empty" exit
u=v
OPEN.PUSH(u)
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
Return VISIT
```

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 5

void depth_first_search(int adj[][MAX],int visited[],int start)

{
```

```
        int stack[MAX];
int top = - 1, i;
printf("%c-",start + 65);
visited[start] = 1;
stack[++top] = start;
while(top!= -1)
{
 start = stack[top];
                for(i = 0; i < MAX; i++)
 {
        if(adj[start][i] && visited[i] == 0)
 {
 stack[++top] = i;
printf("%c-", i + 65);
visited[i] = 1;
break;
 }
 }
                if(i == MAX)
 top--;
}
}
int main()
```

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

bfs

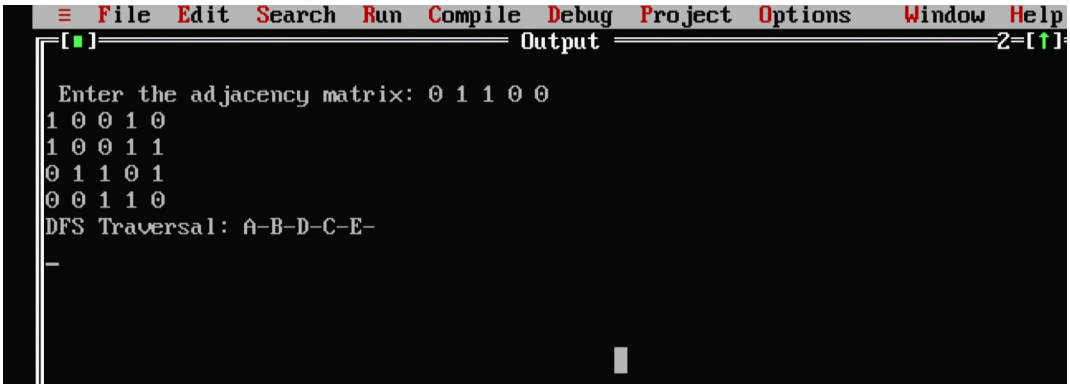
```
#include <stdio.h>
#define MAX 10
void breadth_first_search(int adj[][MAX],int visited[],int start)
{
        int queue[MAX],rear =-1,front =-1, i;
queue[++rear] = start;
visited[start] = 1;
```

```
while(rear != front)
{
 start = queue[++front];
 if(start == 4)
 printf("5\t");
 else
 printf("%c \t",start + 65);
                    for(i = 0; i < MAX; i++)
 {
                              if(adj[start][i] == 1 && visited[i] == 0)
 {
 queue[++rear] = i;
visited[i] = 1;
 }
 }
}
}
int main()
{
          int visited[MAX] = {0};
          int adj[MAX][MAX], i, j;
printf("\n Enter the adjacency matrix: ");
          for(i = 0; i < MAX; i++)
```

```
                    for(j = 0; j < MAX; j++)
 scanf("%d", &adj[i][j]);
breadth_first_search(adj,visited,0);
            return 0;
}
```

Output:

dfs



Bfs

Conclusion:

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

The provided programs implement Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms for graph traversal. The graph representation used in these programs is an adjacency matrix.

Adjacency Matrix Representation:

In an adjacency matrix, a two-dimensional array is used where rows and columns represent nodes (vertices) of the graph. The elements of the array indicate the presence or absence of edges (connections) between nodes. In these programs, adjacency matrices are represented as `int adj[MAX][MAX]`, where `MAX` is a predefined constant representing the maximum number of nodes in the graph.

Why an Adjacency Matrix is Chosen:

The choice of using an adjacency matrix for these programs has certain advantages:

1. Simplicity: Adjacency matrices are easy to understand and implement, making the code more straightforward.
2. Efficient Edge Existence Check: In an adjacency matrix, checking if an edge exists between two nodes is as simple as accessing the corresponding element in the matrix (e.g., `adj[start][i]`). This is efficient for both DFS and BFS algorithms.
3. Space Efficiency for Dense Graphs: If the graph is dense (i.e., it has many edges), an adjacency matrix is a space-efficient representation. It only requires space proportional to the

number of nodes squared.

4. Suitable for Small Graphs: For small graphs where memory usage is not a concern, adjacency matrices are a convenient choice.

However, it's important to note that adjacency matrices can be memory-inefficient for sparse graphs (graphs with few edges) because they require space proportional to the square of the number of nodes, even if most of the entries are zero.

In practice, the choice of graph representation depends on the characteristics of the specific graph and the operations to be performed. For sparse graphs, an adjacency list representation (a list of adjacent nodes for each node) is often preferred because it is more memory-efficient.

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

Breadth-First Search (BFS) and Depth-First Search (DFS) are versatile graph traversal algorithms that have applications beyond just finding connected nodes. Here are some additional applications of BFS and DFS, along with explanations of how they are attained:

Applications of BFS:

1. Shortest Path Finding: BFS can be used to find the shortest path between two nodes in an unweighted graph. Starting from the source node, BFS explores nodes layer by layer until the target node is reached. The path is found by tracing back from the target node to the source node.

2. Maze Solving: BFS is used in maze-solving algorithms to find the shortest path from the starting point to the exit. Each cell in the maze is treated as a node, and BFS is applied to explore cells in the order of their distance from the start.

3. Web CrawlingIn web crawling and web scraping, BFS is used to systematically navigate websites and follow links to gather data. It ensures that pages on the same level are explored before moving to deeper levels of the website.

Applications of DFS:

1. Topological Sorting: DFS can be applied to topologically sort nodes in a directed acyclic graph (DAG). This is useful in scheduling tasks with dependencies, such as in project management or compiler design.
2. Cycle Detection: DFS can detect cycles in a graph. If, during traversal, a previously visited node is encountered (other than the parent in a depth-first search tree), it indicates the presence of a cycle.
3. Path Finding and Connected Components:While DFS can find paths between nodes, it is particularly useful for identifying connected components in a graph. It groups nodes that are reachable from each other.
4. Solving Puzzles and GamesDFS is used in puzzle-solving algorithms (e.g., Sudoku) and board games (e.g., chess). It explores possible moves and states recursively until a solution or game outcome is reached.

How They Are Attained:

- For shortest path finding with BFS: The algorithm starts at the source node and explores neighbors level by level until it reaches the target node. The path is reconstructed by backtracking from the target node to the source node.
- For maze solving with BFS: Each cell in the maze is treated as a node. The algorithm starts from the initial cell and explores neighboring cells layer by layer, marking visited cells and keeping track of the path.
- For topological sorting with DFS: The algorithm recursively explores nodes in a depth-first manner, visiting children before their parents. The order of visiting nodes provides the topological sort.
- For cycle detection with DFS: If the algorithm encounters a previously visited node (other than the parent node in a depth-first search tree), it indicates the presence of a cycle.
- For connected components with DFS: DFS is applied to identify connected components by visiting nodes and marking them as part of the same component. This process is repeated for unvisited nodes.
- For puzzle solving and game playing: DFS explores possible moves or game states recursively

until a solution or game outcome is reached, making decisions at each step to progress toward the goal.