Experiment No.9
Implementation of Graph traversal techniques - Depth First Search, Breadth First Search
Name: Aryan Gaikwad
Roll No: 09
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.

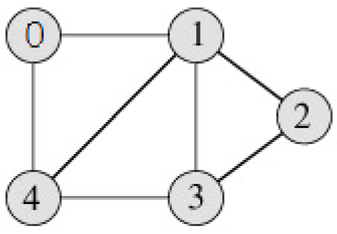
Objective:

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.



$$
\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 \\
2 & 0 & 1 & 0 & 1 & 0 \\
3 & 0 & 1 & 1 & 0 & 1 \\
4 & 1 & 1 & 0 & 1 & 0
\end{array}
$$

DFS Traversal –0 1 2 3 4

Algorithm

Algorithm: DFS_LL(V)

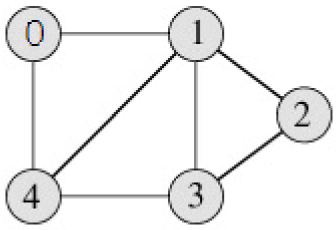
Input: V is a starting vertex

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



$$
\begin{array}{c|ccccc|}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 \\
2 & 0 & 1 & 0 & 1 & 0 \\
3 & 0 & 1 & 1 & 0 & 1 \\
4 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}
$$

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;
}
```

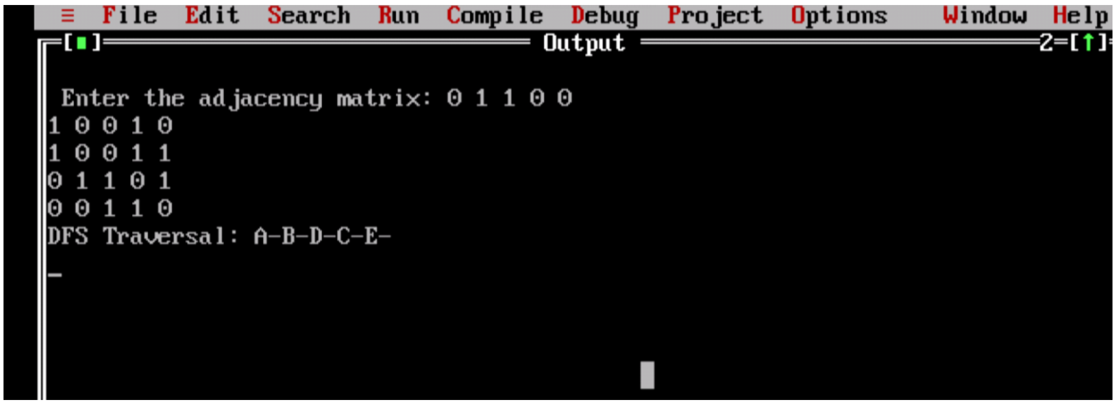
Bfs

```
#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;
}
```

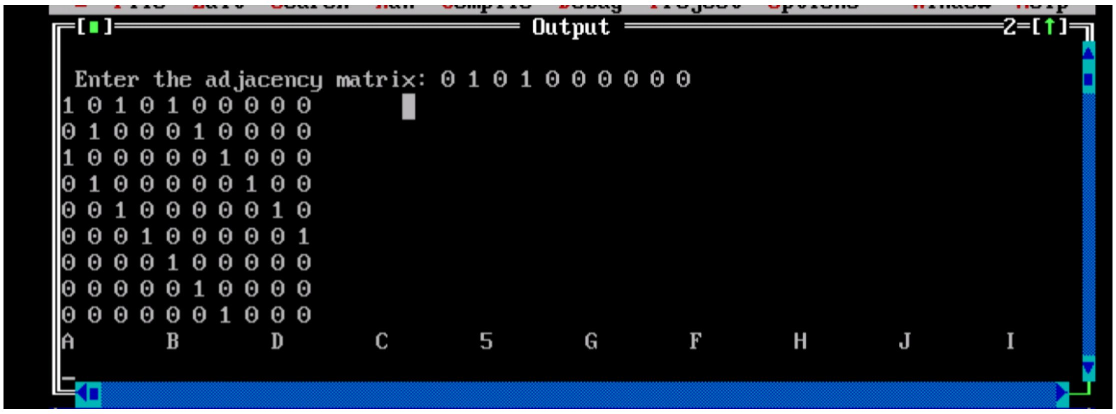
Output:

dfs


≡ File Edit Search Run Compile Debug Project Options Window Help
Output
2=[↑]

```
 Enter the adjacency matrix: 0 1 1 0 0
1 0 0 1 0
1 0 0 1 1
0 1 1 0 1
0 0 1 1 0
DFS Traversal: A-B-D-C-E-
_
```

Bfs



```
 Enter the adjacency matrix: 0 1 0 1 0 0 0 0 0 0
1 0 1 0 1 0 0 0 0 0
0 1 0 0 0 1 0 0 0 0
1 0 0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0 0 1
0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0
A       B       D       C       5       G       F       H       J       I
_
```

Conclusion:

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

The program uses an adjacency matrix to represent the graph. An adjacency matrix is a 2D array where each cell adj[i][j] represents an edge between vertex i and vertex j. The value in the cell indicates the presence (usually 1) or absence (usually 0) of an edge. This representation was chosen because it provides a simple and straightforward way to implement both depth-first search (DFS) and breadth-first search (BFS) algorithms. It allows for easy traversal and checking of connections between nodes. However, it may not be the most memory-efficient representation for sparse graphs.

2) 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 can be applied to various problems beyond finding connected nodes. Here are some applications of BFS and DFS, along with explanations of how they are achieved:

Applications of BFS:

1. Shortest Path Finding (Unweighted Graphs): BFS can be used to find the shortest path between two nodes in an unweighted graph. To achieve this, start the BFS from the source node and keep track of the level (distance) of each visited node from the source. When the destination node is reached, you have found the shortest path.

2. Minimum Spanning Tree (Prim's Algorithm): BFS can be used to find the Minimum Spanning Tree (MST) of a connected, undirected graph. Starting from an arbitrary node, perform BFS, but select edges with the lowest weights as you explore the graph. This will build a tree with the minimum total edge weights that spans all vertices.

3. Bipartite Graph Detection: BFS can determine whether a graph is bipartite (can be split into two disjoint sets of nodes, with no edges between nodes in the same set). You can color nodes alternatively during BFS, and if you encounter an edge connecting two nodes with the same color, it's not bipartite**.**

4. Crawling and Web Search: In web crawling and searching, BFS is used to explore and index web pages. The BFS algorithm starts from the initial web page (e.g., the search engine's homepage) and follows links to discover new pages in a systematic manner.

Applications of DFS:

1. Topological Sorting: DFS can be used to find a topological ordering of nodes in a directed acyclic graph (DAG). A topological order is a linear ordering of nodes such that for every directed edge (u, v), node u comes before node v in the ordering.

2. Cycle Detection: DFS can detect cycles in a graph. When traversing the graph, if you encounter a node that is already in the current DFS path, you've found a cycle. This is useful in applications like deadlock detection and course scheduling.

3. Strongly Connected Components: DFS can identify strongly connected components in a directed graph. Strongly connected components are subsets of the graph where every node can be reached from every other node within that subset.

4. Solving Puzzles and Games: DFS is used to explore various state spaces in puzzles and games, like solving mazes, Sudoku, and chess. The algorithm explores possible moves and choices recursively.

5. Generating Permutations and Combinations: DFS can be employed to generate permutations and combinations of elements in a set. You explore all possible orders or combinations by branching at each step.

6. Symbolic Logic and Propositional Logic: In artificial intelligence and symbolic logic, DFS is used to determine the satisfiability of logical expressions, known as the Boolean satisfiability problem (SAT).

In both BFS and DFS, the key is to explore nodes systematically, visiting each node once and keeping track of the nodes that have been visited or processed. The choice of algorithm depends on the specific problem requirements, such as finding the shortest path, detecting cycles, or exploring all possibilities in a systematic manner. These algorithms are fundamental tools for graph-related and combinatorial problems in computer science and mathematics.