Experiment No.9
Implementation of Graph traversal techniques - Depth First Search, Breadth First Search
Name:Tejashree Anand Karekar
Roll No:20
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.

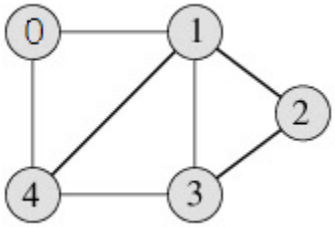
Objective:

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.



	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)

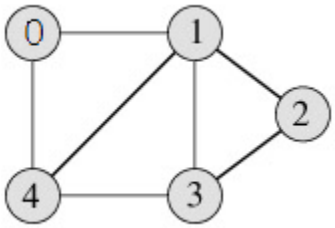
Input: V is a starting vertex

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 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

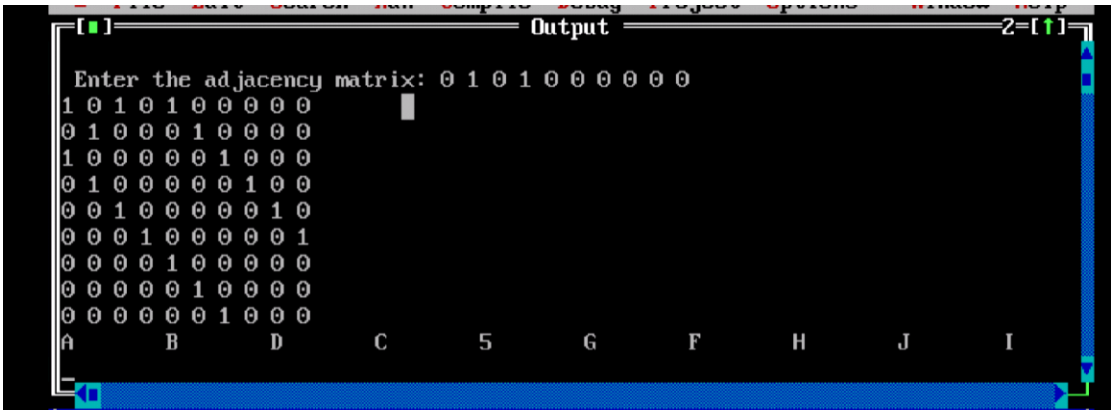
```
#include <stdio.h>
#define MAX 10
void breadth_first_search(int adj[][MAX],int visited[],int start)
{
        int queue[MAX],rear =-1,front =-1, i;
queue[++rear] = start;
visited[start] = 1;
while(rear != front)
{
 start = queue[++front];
 if(start == 4)
 printf("5\t");
 else
 printf("%c \t",start + 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:

1)Write the graph representation used by your program and explain why you choose that.
I don't have a specific program or implementation, but I can describe a common graph representation used in computer science. One common representation is the **adjacency list**, which is often chosen for its efficiency and flexibility.

In an adjacency list representation, each vertex in the graph is associated with a list of its neighboring vertices. This representation is chosen for several reasons:

1. **Space Efficiency**: It is memory-efficient for sparse graphs since it only stores information about connected vertices, saving space compared to an adjacency matrix.

2. **Efficient for Sparse Graphs**: For graphs with relatively few edges, adjacency lists are more efficient in terms of space and time complexity for traversal and manipulation.

3. **Flexibility**: It's versatile and can be used for both directed and undirected graphs by storing edges once (for undirected) or twice (for directed).

4. **Easy to Traverse Neighbors**: When you want to find the neighbors of a vertex, you can directly access the associated list, making traversal efficient.

However, the choice of graph representation can depend on the specific application and the types of operations you need to perform. In dense graphs or when you need constant-time edge lookups, an adjacency matrix might be a better choice. The decision often involves trade-offs between space efficiency and time complexity for various graph operations.

2)Write the applications of BFS and DFS other than finding connected nodes and explain how it is attained? **Breadth-First Search (BFS) Applications**:

1. **Shortest Path**: BFS can be used to find the shortest path between two nodes in an unweighted graph. By exploring nodes level by level, the first time the destination node is encountered, it guarantees the shortest path.

2. **Web Crawling**: Search engines use BFS to index and crawl web pages. The search starts from a seed webpage and expands to linked pages in a breadth-first manner.

3. **Social Network Analysis**: BFS can help analyze the structure of social networks, such as finding the degrees of separation between individuals.

4. **Puzzle Solving**: BFS is used to solve puzzles like the 8-puzzle or the Tower of Hanoi by systematically exploring possible states or moves.

Depth-First Search (DFS) Applications:

1. **Topological Sorting**: DFS can be used to perform topological sorting of directed acyclic graphs, which is useful in scheduling tasks with dependencies.

2. **Strongly Connected Components**: DFS helps find strongly connected components in directed graphs, which has applications in compiler optimization, graph analysis, and network analysis.

3. **Maze Solving**: DFS can be used to find a path through a maze. It explores as far as possible along each branch before backtracking.

4. **Detecting Cycles**: DFS can be used to detect cycles in a graph. If a back edge is encountered during the traversal, it indicates the presence of a cycle.

In both cases, these algorithms are attained by systematically exploring the graph from a starting node. BFS explores neighbors before moving to their neighbors (level by level), while DFS explores as deeply as possible along each branch before backtracking. The specific application determines how the traversal results are used, such as finding the shortest path, detecting cycles, or discovering connected components. The choice between BFS and DFS depends on the problem's requirements and the nature of the graph.