Experiment No.9
Implementation of Graph traversal techniques - Depth First Search, Breadth First Search
Name: Swarup Satish Kakade
Roll No: 19
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.

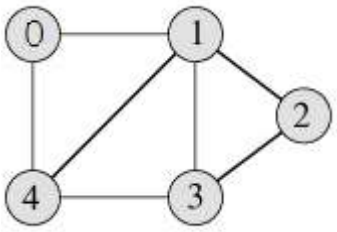
Objective:

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)

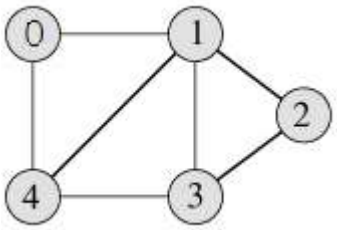
Input: V is a starting vertex

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;
}
```

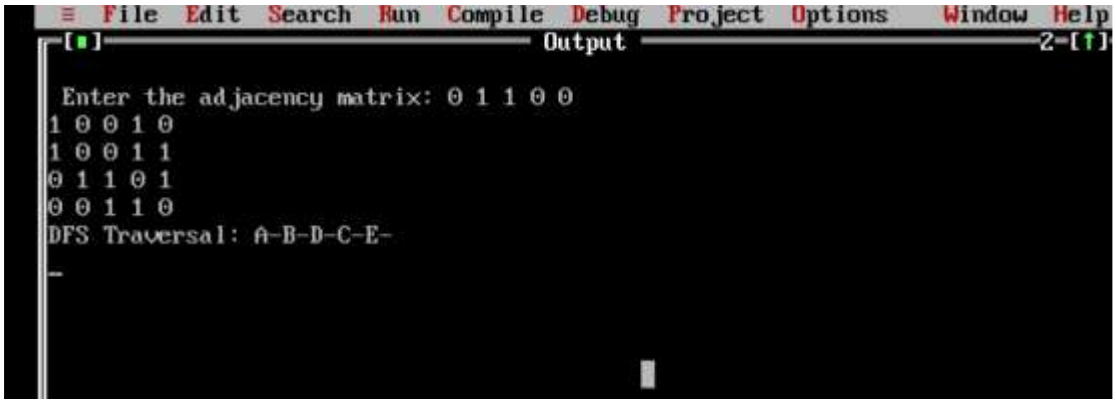
BFS

```
#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 .

The program utilizes an adjacency matrix to depict the graph. In this matrix, each cell adj[i][j] signifies a connection between vertex i and vertex j. The value in each cell typically denotes the existence of an edge, with 1 indicating its presence and 0 representing its absence. This choice of representation was made due to its simplicity and ease of implementation, particularly when working with depth-first search (DFS) and breadth-first search (BFS) algorithms. It enables straightforward traversal and facilitates the verification of connections between nodes. However, it's worth noting that for graphs with sparse structures, this representation may not be the most memory-efficient option

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

Breadth-First Search (BFS) has diverse applications in various domains:

1. **Shortest Path Discovery:** In unweighted graphs, BFS is a handy tool for determining the shortest path between two nodes. It's frequently harnessed in navigation systems to calculate the quickest route between two locations, assuming uniform travel time for each edge.
2. **Connected Components Identification:** Beyond its role in identifying connected nodes, BFS is also employed to uncover connected components within a graph. These components are essentially subsets of nodes where every node can be reached from any other node within the same subset.
3. **Bipartite Graph Recognition:** BFS is instrumental in discerning whether a graph is bipartite, meaning it can be neatly divided into two sets of nodes. In such a graph, there are no connections between nodes within the same set. This determination is made by checking for the existence of a cycle with an odd length during graph traversal.
4. **Web Crawling:** Search engines and web crawlers extensively use BFS for web page indexing by systematically following links. The BFS algorithm ensures that pages on the same level (i.e., at the same distance from the starting page) are explored before venturing deeper into the web structure.

Depth-First Search (DFS) has diverse applications in various domains:

1. **Topological Ordering:** DFS is a valuable tool for accomplishing topological sorting in directed acyclic graphs (DAGs). This is crucial in scenarios where tasks possess dependencies, such as in project management, ensuring a systematic order of execution.
2. **Cycle Detection:** DFS serves as a reliable mechanism to detect cycles within a graph. Its applications extend to critical areas like identifying deadlocks in

operating systems and pinpointing interdependencies in software development, enhancing system reliability.

3. **Pathfinding Strategies:** In diverse domains such as maze-solving and game development, DFS is deployed to explore potential paths until a specific objective is achieved. This can be pivotal for navigating mazes or uncovering alternative routes in gaming scenarios.

4. **Connected Components and Strongly Connected Components:** While BFS excels in finding connected components, DFS is the preferred choice for detecting strongly connected components (SCCs) within directed graphs. SCCs hold significance in various fields, including compiler optimization and the intricate analysis of social networks, facilitating insights into network dynamics and relationships.