Experiment No.9
Implementation of Graph traversal techniques - Depth First Search, Breadth First Search
Name:Nitish Jha
Roll No:18
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.

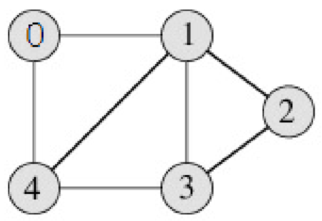
Objective:

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.



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

DFS Traversal –0 1 2 3 4

Algorithm

Algorithm: DFS_LL(V)

Input: V is a starting vertex

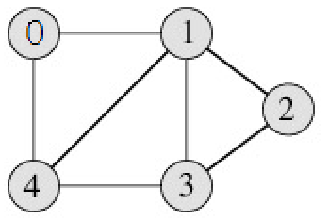
Output : A list VISIT giving order of visited vertices during traversal.

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

```c
#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;
```

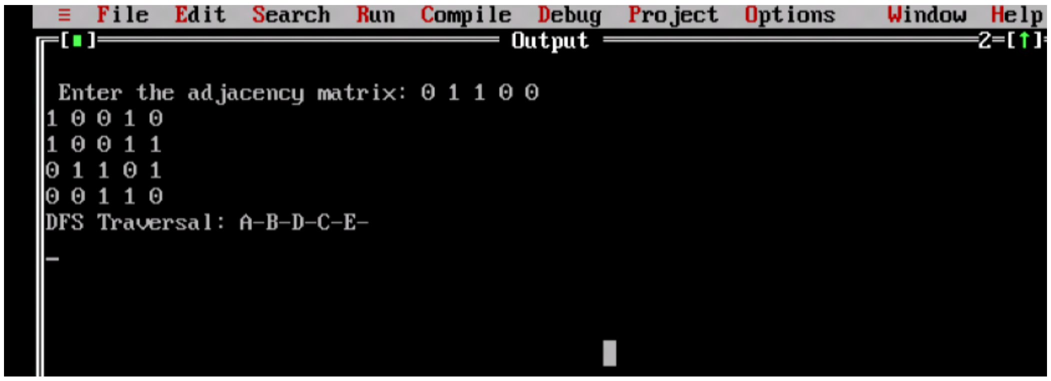
```
}
```

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;

}
```

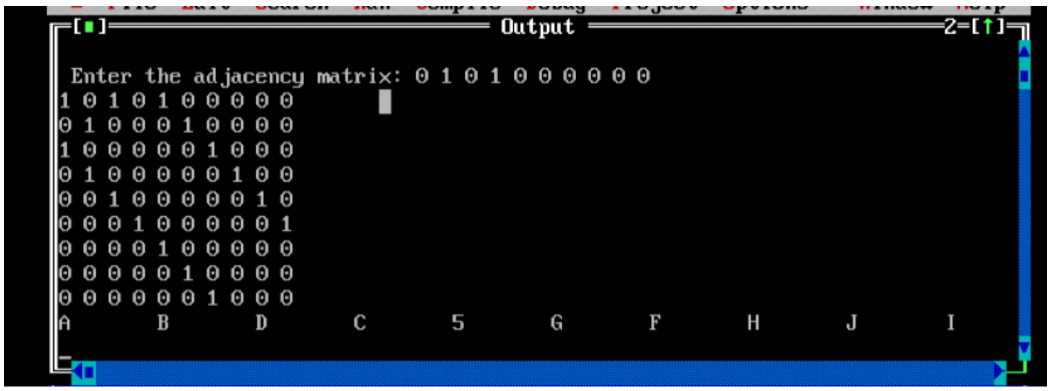
Output:

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 employs an adjacency matrix to depict the graph's structure. This matrix is essentially a 2D array where each element adj[i][j] signifies whether there is an edge connecting vertex i to vertex j. Typically, a value of 1 indicates the presence of an edge, while 0 implies its absence. The choice of using an adjacency matrix was based on its simplicity and suitability for implementing depth-first search (DFS) and breadth-first search (BFS) algorithms. This representation offers a straightforward means of navigating the graph and verifying relationships between nodes. However, it may not be the most memory-efficient choice when dealing with graphs that have relatively few connections or edges.

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) offer versatile solutions for multiple real-world scenarios:

- **Shortest Path Discovery:** Both BFS and DFS are invaluable for determining the shortest path between two nodes in a graph. While BFS excels in unweighted graphs, DFS can navigate weighted graphs using backtracking techniques.

- **Efficient Web Crawling:** Search engines utilize BFS to systematically crawl and index web pages. By exploring links level by level from an initial webpage, they efficiently index the vast web.

- **Puzzle Resolution:** DFS is a key tool for solving puzzles, ranging from intricate mazes to Sudoku and N-Queens puzzles. It systematically explores various pathways to uncover solutions.

- **Network Communication:** In computer networks, BFS ensures efficient broadcast of messages or information, preventing unnecessary duplication while reaching all connected nodes.

- **Topological Arrangement:** DFS aids in establishing the topological order of vertices in directed acyclic graphs, a critical component in scheduling and task management.

- **Minimum Spanning Trees:** Both algorithms play a pivotal role in discovering minimum spanning trees within graphs. This has far-reaching implications, from network design to data clustering.

- **Social Network Analysis:** BFS assists in calculating the degrees of separation between individuals in social networks, such as the popular "Six Degrees of Kevin Bacon" game.

- **Memory Cleanup:** DFS is a fundamental component in garbage collection algorithms, effectively reclaiming memory from objects that are no longer accessible, commonly employed in languages like Java.

These algorithms exhibit remarkable adaptability and can be applied creatively across diverse domains to address a wide array of challenges that extend beyond simple graph connectivity analysis.