

Date-

Assignment No. :

Problem Statement:

Program in C to traverse the vertices of a given graph with depth first search algorithm.

Theory:

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected. This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

- i> Pick a starting node and push all its adjacent nodes into a stack.
- ii> Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Algorithm:

Input specification:

1. I : The incidence matrix of dimension $(n \times n)$ of the given graph
2. v_s : The source vertex to start the search from

Output specification:

1. A two dimensional array $I[1..n][1..n]$ whose starting index is 1 and ending index is n , size of the array being $(n \times n)$.
2. A stack to store the intermediate vertices, say S .

Steps:

/*1 is the adjacency matrix of a graph G , and v_s is the source vertex from

which traversal would start */

Step 1: Repeat step 2 For(all $v \in V$)

Step 2: $Status[v] = \text{unvisited}$ //initially all node is made unvisited
[End of For loop]

Step 3: Set $Status[v_s] = \text{visited}$

Step 4: Set $U = v_s$

Step 5: $Push(S, v_s)$ //Push is a function to push an element into any stack S
[Starting Do-While loop]
 $found = \text{FALSE}$

Step 6: Repeat through step 9 to step 21 For(all $y \in V$)

Step 7: If($status[v] = \text{unvisited}$ AND v is adjacent to u)

Step 8: Print u, v

Step 9: $Push(S, v)$ // insert an element into stack S

Step 10: $status[v] = \text{visited}$

Step 11: $u = v$

Step 12: $found = \text{TRUE}$

Step 13: Break

[End of If structure]

Step 14: If ($found = \text{FALSE}$)

Step 15: u=POP(S) // delete an element from stack S

[End of If structure]

[End of For loop]

Step 16: Repeat through step 7 to step 20 while(Q is not empty OR v is visited)

[End of Do-While loop]

Step 17: End

Source Code:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node //Creation of node
{
    int data;
    struct node *next;
} *h=NULL;
struct node *getnode(int data) //Dynamic allocation
{
    struct node *temp;
    temp= (struct node *) malloc (sizeof (struct node));
    temp->data=data;
    temp->next=NULL;
    return temp;
}
void push(int data) //Definition of the push function
{
    struct node *t,*x;
    x=getnode (data);
    if (h==NULL)
    {
        h=x;
    }
    else
    {
        x->next=h;
```

```

        h=x;
    }
}
int pop() //Definition of the pop function
{
    int u;
    if (h==NULL)
        printf ( "UNDERFLOW");
    else
    {
        u=h->data;
        h=h->next;
    }
    return u;
}
int status(int s[30],int n) //Definition of the status function
{
    int i;
    for (i=1; i<=n; i++)
    {
        if(s[i]==0)
            return 1;
    }
    return 0;
}
void dfs(int l[10][10],int n,int vs) //Function for the searching algorithm
{
    int i, u, s[30] , found;
    for (i=1; i<=n; i++)
        s [i]=0;
    s [vs]=1;
    u=vs;
    push(vs);
    do
    {
        do

```

```

{
    found=0;
    for (i=1; i<=n; i++)
    {
        if(s[i]==0&&l[u][i]==1)
        {
            printf("\n%d %d",u,i);
            push(i);
            s[i] =1;
            found=1;
            u=i;
            break;
        }
    }
    if (found==0)
    {
        u=pop();
    }
}while(h!=NULL) ;
for ( i=1; i<=n; i++)
{
    if(s[i]==0)
    {
        s[i]=1;
        push(i);
        u=i;
        break;
    }
}
for (i=1 ; i<=n; i++)
{
    if(l[u][i]==1&&s[u]!=2)
    {
        printf("\n%d to %d",u,i);
        s[u]=2;
        break;
    }
}

```

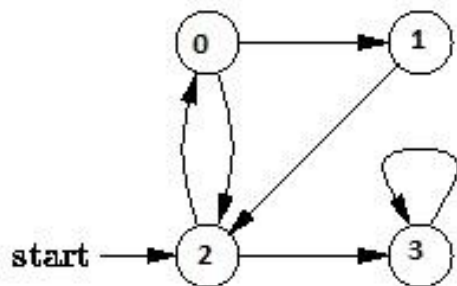
```

        }
    }while (status (s, n));
}
void show(int l[10][10],int n)
{
    int i,j;
    printf("\n") ;
    for (i=1; i<=n; i++)
    {
        for(j=1; j<=n;j++)
            printf(" %d ",l[i][j]);
        printf("\n");
    }
}
int main()
{
    int choice;
    int l[10][10],n,i,j,vs;
    printf("Enter order of the adjacency matrix : ");
    scanf ("%d", &n);
    for (i=1; i<=n; i++)
        for(j=1; j<=n; j++)
        {
            printf("Enter weight between %d & %d",i,j);
            scanf("%d",&l[i][j]);
        }
    printf("\nAdjacency matrix is . . .\n");
    show (l, n) ;
    printf("Enter source vertex :");
    scanf("%d",&vs);
    dfs(l,n,vs);
    getch();
    return 0;
}

```

Input & Output:

Input graph:



Output of program:

```
Enter order of the adjacency matrix : 4
Enter weight between 1 & 1: 0
Enter weight between 1 & 2: 1
Enter weight between 1 & 3: 1
Enter weight between 1 & 4: 0
Enter weight between 2 & 1: 0
Enter weight between 2 & 2: 0
Enter weight between 2 & 3: 1
Enter weight between 2 & 4: 0
Enter weight between 3 & 1: 1
Enter weight between 3 & 2: 0
Enter weight between 3 & 3: 0
Enter weight between 3 & 4: 1
Enter weight between 4 & 1: 0
Enter weight between 4 & 2: 0
Enter weight between 4 & 3: 0
Enter weight between 4 & 4: 1
```

Adjacency matrix is . . .

```
0 1 1 0
0 0 1 0
1 0 0 1
0 0 0 1
```

Enter source vertex :3

```
3 1
1 2
4 to 4
```

Discussion:

1. Setting a nodes (with Stack) label takes $O(1)$ time.
2. Each Nodes Is labeled twice:
 - a. Once as Unexplored.
 - b. Once as Visited.
3. Each Edge is labeled twice:
 - a. Once as Unexplored.
 - b. Once as Discovery or BACK.
4. Because the adjacency list of each nodes is scanned only when the nodes is Pop, each adjacency list is scanned at most once. Total time spent in scanning adjacency list is $O(E)$ [in worst case]. As initializations, takes $O(V)$ times, then total running time of DFS is $O(V + E)$.