**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:**

**Algorithm\_DFS(I, vs)**

**Input :**

1. I : The incidence matrix of dimension (n x n) of the given graph
2. vs : The source vertex to start the search from

**Output :** All vertices of the graph visited along the depth of the graph.

**Data Structure Used :**

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 x n)
2. A stack to store the intermediate vertices, say S

**STEPS:**

1. Repeat step 1.a for(all v Є V)

Begin

* 1. Set Status[v] = unvisited //initially all node is made unvisited

[End of for loop]

1. Set Status[vs] = visited
2. Set u = vs
3. Push(S, vs) // Push is a function that inserts an element

// to the top of the stack S

1. Repeat through steps 5.a to 5.g while (S is not empty Or Status[v] = visited)

Begin

* 1. Set Found = False
  2. Repeat through steps b.i to b.ii for(all y Є V)

Begin

* + 1. If(status[v] = unvisited And v is adjacent to u)

Then

* + - 1. Print u, v
      2. Push(S, v)
      3. Set status[v] = visited
      4. Set u = v
      5. Set Found = True
      6. Break

[End of if structure]

* + 1. If (Found = False)

Then

* + - 1. Set u = Pop(S) // Pop is a function that retrieves and removes // an element from the top of the stack

[End of if structure]

[End of for loop]

[End of while loop]

**SOURCE CODE:**

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*next;

} \*h = NULL;

struct node \*getnode(int data) {

struct node \*temp;

temp = (struct node \*)malloc(sizeof(struct node));

temp->data = data;

temp->next = NULL;

return temp;

}

void push(int data) {

struct node \*x;

x = getnode(data);

if (h == NULL) {

h = x;

} else {

x->next = h;

h = x;

}

}

int pop() {

int u = 0;

if (h == NULL)

printf("UNDERFLOW");

else {

u = h->data;

h = h->next;

}

return u;

}

int status(int s[30], int n) {

int i;

for (i = 0; i <= n-1; i++) {

if (s[i] == 0)

return 1;

}

return 0;

}

void dfs(int l[10][10], int n, int vs) {

int i, u, s[30], found;

for (i = 0; i <= n-1; i++)

s[i] = 0;

s[vs] = 1;

u = vs;

push(vs);

do {

do {

found = 0;

for (i = 0; i <= n-1; 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 = 0; i <= n-1; 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 = 0; i <= n-1; i++) {

for (j = 0; j <= n-1; j++)

printf(" %d ", l[i][j]);

printf("\n");

}

}

int main() {

int l[10][10], n, i, j, vs;

printf(" Enter order of the adjacency matrix : ");

scanf("%d", &n);

for (i = 0; i <= n-1; i++)

for (j = 0; j <= n-1; j++) {

printf("Enter the existence of path between vertices %d & %d: ", i, j);

scanf("%d", &l[i][j]);

}

printf("\nAdjacency matrix is . . .\n");

show(l, n);

printf("Enter source vertex :");

scanf("%d", &vs);

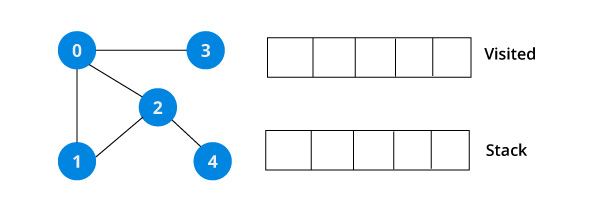
dfs(l, n, vs);

return 0;

}

**INPUT AND OUTPUT:**

**The given graph is:**



The existence of path between every pair of vertices:

1->There is a path between vertices

0->There is no path between vertices

Enter order of the adjacency matrix : 5

Enter the existence of path between vertices 0&0: 0

Enter the existence of path between vertices 0&1: 1

Enter the existence of path between vertices 0&2: 1

Enter the existence of path between vertices 0&3: 1

Enter the existence of path between vertices 0&4: 0

Enter the existence of path between vertices 1&0: 1

Enter the existence of path between vertices 1&1: 0

Enter the existence of path between vertices 1&2: 1

Enter the existence of path between vertices 1&3: 0

Enter the existence of path between vertices 1&4: 0

Enter the existence of path between vertices 2&0: 1

Enter the existence of path between vertices 2&1: 1

Enter the existence of path between vertices 2&2: 0

Enter the existence of path between vertices 2&3: 0

Enter the existence of path between vertices 2&4: 1

Enter the existence of path between vertices 3&0: 1

Enter the existence of path between vertices 3&1: 0

Enter the existence of path between vertices 3&2: 0

Enter the existence of path between vertices 3&3: 0

Enter the existence of path between vertices 3&4: 0

Enter the existence of path between vertices 4&0: 0

Enter the existence of path between vertices 4&1: 0

Enter the existence of path between vertices 4&2: 1

Enter the existence of path between vertices 4&3: 0

Enter the existence of path between vertices 4&4: 0

Adjacency matrix is . . .

01110

10100

11001

10000

00100

Enter source vertex :0

0 to 1

1 to 2

2 to 4

3 to 0

**DISCUSION:**

1. Setting a node’s label (with Stack) takes O(1) time.
2. Each node is labeled twice:
   1. Once as unexplored.
   2. Once as visited.
3. Each edge is labeled twice:
   1. Once as unexplored.
   2. Once as discovered.
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 adjaceny list is O ( E ) [ in worst case ]. As initializations, takes O( V ) times, then total running time of DFS is

O(V + E).