Date : Page No :

Assignment Number

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 while moving forward if no more nodes along the current path, a move to backwards on the same path is taken 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:

1. Pick a starting node and push all its adjacent nodes into a stack.
2. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
3. Repeat this process until the stack is empty.

However, it must be ensured that the nodes that are visited are marked. This will prevent from visiting the same node more than once. If the nodes that are visited are not marked, same node is visited more than once, the algorithm may end up in an infinite loop.

Algorithm

Algorithm_DFS(I, v_S)

Input :

1. I : The incidence matrix of dimension (n x n) of the given graph
2. v_S : 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 :

Step 1: Repeat step 1.a for(all v ∈ V)
Begin
a) Set Status[v] = unvisited //initially all node is made unvisited
[End of for loop]

Step 2: Set Status[v_S] = visited

Step 3: Set u = v_S

Step 4: Push(S, v_S) // Push is a function that inserts an element
// to the top of the stack S

Step 5: Repeat through steps 5.a to 5.g while (S is not empty Or Status[v] = visited)
Begin
a) Set Found = False
b) Repeat through steps b.i to b.ii for(all y ∈ V)
Begin
i. If(status[v] = unvisited And v is adjacent to u)
Then
I. Print u, v
II. Push(S, v)

III. Set status[v] = visited
IV. Set u = v
V. Set Found = True
VI. Break
[End of if structure]

ii. If (Found = False)
Then
I. 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 = 1; i <= n; 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 = 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 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);
    return 0;
}
```

Input and Output

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

Adjacency matrix is . . .

 1  0  0
 0  1  0
 1  0  1
Enter source vertex :1

2 to 2
3 to 1
```

Discussion

1. Setting a node's label (with Stack) takes O(1) time.
2. Each node is labeled twice:
   a) Once as unexplored.
   b) Once as visited.
3. Each edge is labeled twice:
   a) Once as unexplored.
   b) 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).