Algo. DeleteEdge-LL-DG

Input: DGptr, the pointer to the graph. $\langle V_i, V_j \rangle$, the edge to be deleted from the vertex V_i to V_j.

Output: The graph w/o the edge from the vertex V_i to V_j.

Steps:

Malay

1. Let N = no. of vertices in the graph.
2. If (V_i > N) or (V_j > N) then
3. Print "Vertex does not exist : Error in edge removal"
4. Else
5. DeleteAny_SL (DGptr[V_i], [V_j]) // Delete V_j from the adjacency list of V_i
6. EndIf
7. Stop.

Breadth-first Search Algorithm

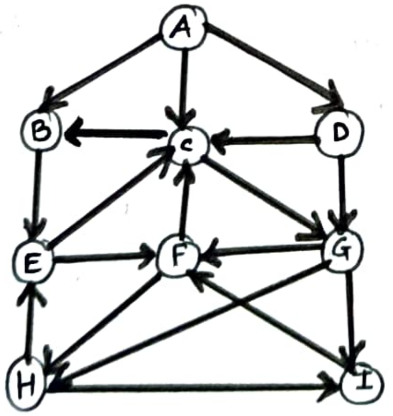
Breadth - First Search (BFS) → is a graph search algorithm that begins at the Root Node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explore their unexplored neighbouring nodes and so on, until it finds the goal.

That is, we start examing the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth.

Malay

This means that we need to track the neighbour of the node and guarantee that every node in the graph is processed and no node is processed more than once.

This is accomplished by using a queue that will hold the nodes that are waiting for further processing & a variable STATUS to represent the current state of the node.



GRAPH G, & its Adjacency lists.

Adjacency list
A: B,C,D
B: E
C: B,G
D: C,G
E: C,F
F: C,H
G: F,H,I
H: E,I
I: F

Consider the graph G given, The adjacency list of G is also given. Assume that G represent the daily flights b/w different cities and we want to fly from city A to I with minimum stops. That is, find the minimum path P from A to I given that every edge has a length of 1.

Soln: The minimum path P can be found by applying the BFS algorithm that begins at city A and ends when I is encountered. During the execution of the algorithm, we use two arrays:→

QUEUE and ORIG. While QUEUE is used to hold the nodes that have to be processed, ORIG is used to keep Track of origin of each edge.

Malay

Initially, FRONT = REAR = -1. The algorithm for this is as follows:—

(a) Add A to Queue and NULL to ORIG

FRONT = 0	QUEUE = A
REAR = 0	ORIG = \0

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and Enqueue the neighbours of A. Also, add A as the ORIG of its neighbour.

FRONT = 1	QUEUE = A	B	C	D
REAR = 3	ORIG = \0	A	A	A

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbour of B. Also add B as the ORIG of its neighbour

FRONT = 2	QUEUE = A	B	C	D	E
REAR = 4	ORIG = 10	A	A	A	B

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbour of C. Also add C as the ORIG of its neighbour. Note that C has two neighbours B and G. Since B has already been added to the queue & it is not in the Ready State, we will not add B and only add G.

Malay

FRONT = 3	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = 10	A	A	A.	B	C

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbour. Note that D has two neighbours C and G. Since both of them have already been added to the queue & they are not in the Ready State, we will not add them again.

FRONT = 4	QUEUE	A	B	C	D	E	G
REAR = 5	ORIG	10	A	A	A	B	C

(f) Dequeue a node by setting FRONT = FRONT+1 and enqueue the neighbours of E. Also add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the Queue and it is not in the Ready state, we will not add C and add only F.

FRONT = 5	QUEUE = A	B	C	D	E	G	F
REAR = 6	ORIG = 10	A	A	A	B	C	E

(g) Dequeue a node by setting FRONT = FRONT+1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H and I.

FRONT = 6	QUEUE = A	B	C	D	E	G	F	H	I
REAR = 9	ORIG = 10	A	A	A	B	C	E	G	G

Malay

Since F is already added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered & added to the QUEUE.

Now, Backtrack from I using ORIG to find the minimum path P. Thus we have P as

$$A \longrightarrow C \longrightarrow G \longrightarrow I.$$

ALGORITHM FOR BFS.

Step 1 : SET STATUS = 1 (READY STATE).
FOR EACH NODE IN G.

Step 2: Enqueue the starting node A and set its STATUS = 2
(waiting state)

step 3: Repeat steps 4 and 5 until QUEUE is empty

step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Maky

step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

step 6: EXIT.

SPACE COMPLEXITY ⇒ $O(b^d)$ → Branching factor b (no. of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level $O(b^d)$.

TIME COMPL. = $O(|E| + |V|)$ — all edges and vertices are traversed.

B.F.S.

```c
# 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 ("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;
```

MALAY
TRIPATHI

output

0 1 0 1 0
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
0 0 1 1 0
A B D C E

DEPTH - FIRST SEARCH ALGORITHM.

The depth first search algorithm progress by expanding the starting node of G and then going deeper and deeper until goal node is found, or until a node that has no children is encountered.

When the dead-end reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.

In otherwords, depth-first search begins at a starting node A which becomes the current node. Then it examines each node N along a path P which begins at A. That is, we process a neighbours of A, then a neighbours of neighbours of A, and so on.
During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node.
Otherwise, the unvisited (unprocessed) nodes become the current node.

The algorithm proceeds like this until we reach a dead-end. On reaching the dead-end, we backtrack to find another path P'.
The algorithm terminates when backtracking leads back to the starting node A. In this algorithm, edges that lead to a new vertex are called DISCOVERY EDGES and EDGES that lead to an already visited vertex are called BACK EDGES.

Malay

This algo. is similar to the IN-ORDER TRAVERSAL OF A BINARY TREE.

ALGORITHM.

Step1. SET STATUS=1 (ready state) for each node in G

Step2. Push the starting node A on the stack and set its status =2 (waiting state)

Step3. Repeat Steps 4 and 5 until Stack is empty.

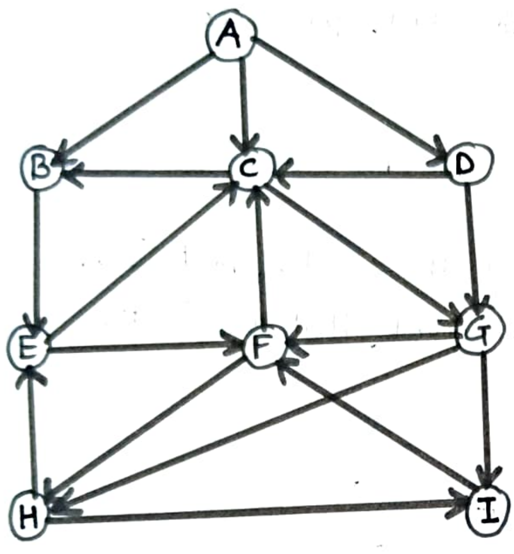
Step4. Pop the top Node N. Process it and set its STATUS=3 (processed state).

Step5. PUSH on the stack all the neighbours of N that are in the ready state (whose STATUS=1) and set their STATUS= 2 (waiting state)

[END OF LOOP]

Step6. EXIT

Consider the graph G given in the Fig B. The adjacency list of G is also a given. Suppose we want to print all the nodes that can be reached from the node H (including H itself). One alternative is to use the a Depth-First Search of G starting at node H. The procedure can be explained here.



Adjacency lists

A : B, C, D

B : E

C : B, G

D : C, G

E : C, F

F : C, H

G : F, H, I

H : E, I

I : F

Solution:-

(a) Push H onto the stack

STACK : H

(b) Pop and print the top element of the STACK, that is H. Push all the neighbours of H onto the stack that are in the Ready state. The stack now becomes

PRINT : H

STACK : E, I

(c) Pop and print the top element of the STACK, that is I. Push all the neighbours of I onto the stack that are in the ready state. The stack now becomes

PRINT : I $\boxed{\text{STACK: E, F}}$

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the Ready state. (Note F has 2 neighbours, C and H. But only C will be added, as H is not in the ready state). The stack now becomes

PRINT : F $\boxed{\text{STACK : E, C}}$

(e) Pop and print the top element of the stack, that is C. Push all the neighbours of C onto the stack that are in the ready state. The stack now becomes

PRINT : C $\boxed{\text{STACK : E, B, G}}$

(f) Pop and print the Top element of the STACK, that is G. Push all neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no Push operation is performed. The stack now becomes

PRINT : G $\boxed{\text{STACK : E, B}}$

(g) Pop and print the top element of the stack, that is B. Push all the neighbours of B onto the stack that are in the ready state. Since there are No neighbours of B that are in the ready state, no push operation is performed. The stack now becomes

PRINT : B $\boxed{\text{STACK : E}}$

(h) Pop and print the top element of the STACK; that is E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The stack now become empty.

PRINT : E | STACK: |

Since stack is now empty, the DFS of G starting at node H is complete and the nodes which were printed are :

| H, I, F, C, G, B, E |

TIME COMPLEXITY:

The time complexity of the D-F-S is proportional to the number of vertices plus the number of edges in the graph that are traversed. The time complexity can be given as (O|V| +|E|).

Program of 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]);
print("DFS TRAVERSAL");
depth-first-search(adj,visited,0);
printf("\n");
return 0;
}
```

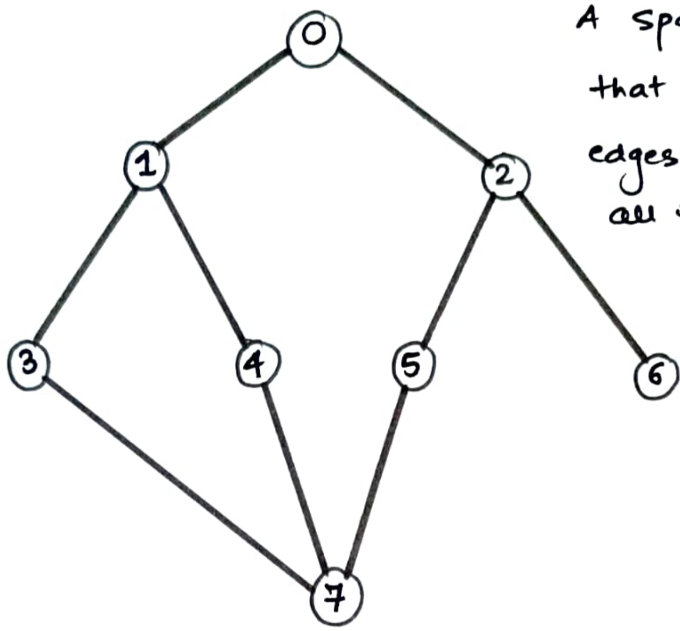
OUTPUT

ENTER THE ADJACENCY MATRIX

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

DFS TRAVERSAL : A→C→E

Ex.



Graph.

A SpanningTree is any tree that consists solely of edges in G & that includes all the vertices in G.

BFS (0) Spanning Tree	DFS (0) Spanning Tree