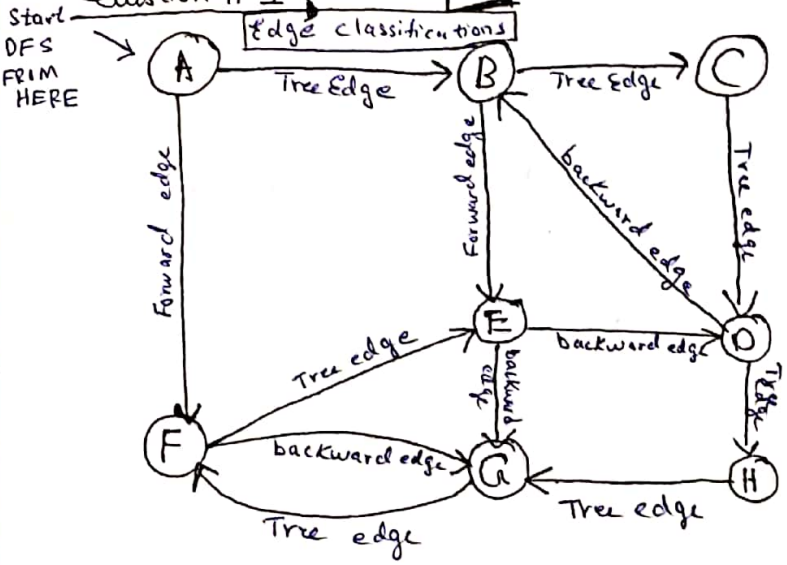
Aisha Muhammad Nawaz
20L-0921
BSCS -4A DAA
(Dr. Maryam Bashir)

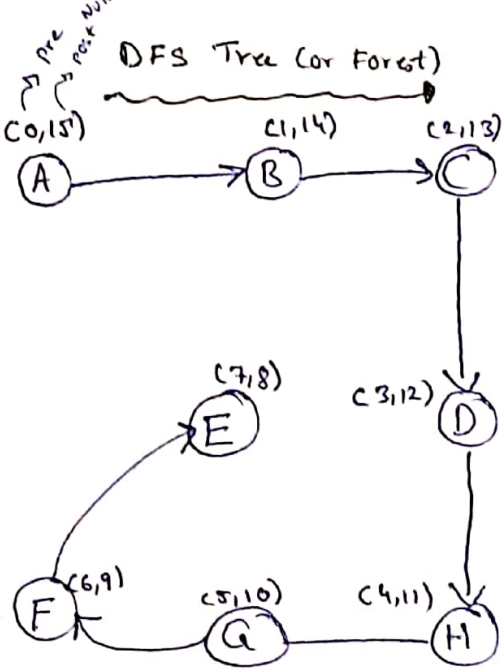
HomeWork # 8 Graphs

Question # 1

A

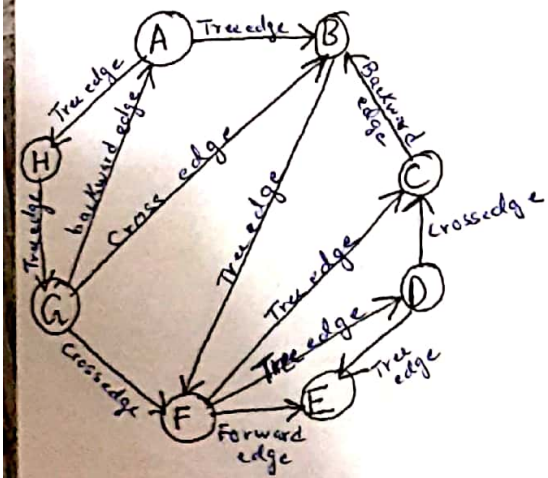
Edge classifications

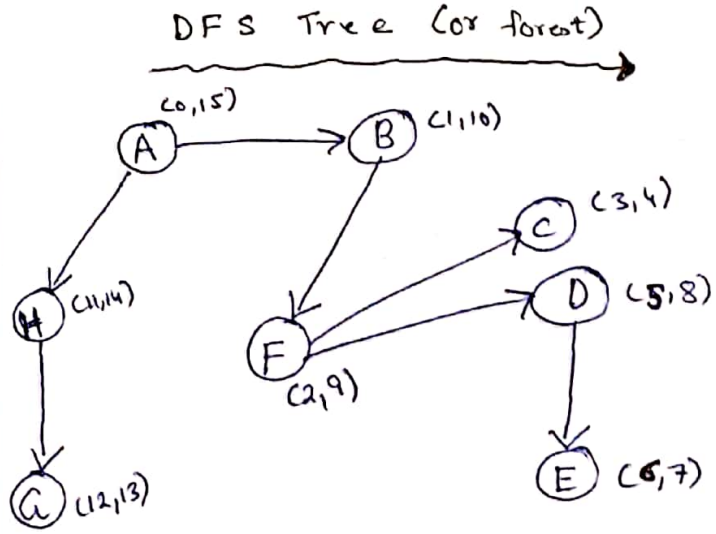




B

Edge Classifications





Question #2

```
DFS (G)     int min; // will save minimum semesters required
{   for each u ∈ V do
        color [u] ← white
        π [u] ← NIL
        time ← 0                    LinkList < char, int, int > obj   // will save node char name eg A, B, C finish time and start time
    for each u ∈ V do
{       if color [u] = white then
            DFS-VISIT (G, u, obj)}
                                    min = obj.gettail.finishtime()
                                          - obj.getheadstarttime();
                                    return min;}
```

```
DFS-VISIT (G, u, obj)
{   color [u] ← gray
    d [u] ← time ← time+1
    for each v ∈ Adj [u] do
        if color [v] = white then
            π [v] ← u
            DFS-VISIT (G, v, obj)
    color [u] ← black
    f [u] ← time ← time+1
    obj.insert at head (u, f [u], d [u])
}
```

Explanation using DFS ALGORITHM

This code uses the logic of topological sort with a few modifications. The LinkList stored not only stores node names in order of decreasing finish times but also stores corresponding finish time and start time of that particular node. To get the minimum number of semesters required to graduate, simply add the finish time of node at tail to the start time of node at head.

Question #3

Explanation

This algorithm makes use of the data structure hashmap to traverse the graph only once and reverse the adjacency list of a directed Graph.

Main Idea:

→ A hashmap shall be maintained of Adjacency lists The Key will be the vertex number and the value will be of an array type that will store the corresponding vertexes it has an edge with.

→ A new hashmap will be made of the same type after going through the original hashmap and for every key we will traverse the corresponding list.

→ For each vertex in the correspond list, new key will be added in the new hashmap, ~~nothing~~ and the original hashmap's key will added as an entry in the list for the new key in new hashmap.

Algorithm:

```
Hashmap < char, ~~int[]~~ char[] > Reversed ;
for (int i=0; i<n; i++)
{   for each ~~v ∈ adj i~~ edge of i
    ~~Reversed.insert (v, i)~~ for each i ∈ V
        for each edge of (i, w)
        {
            for each (i, w) ∈ E
                add the edge into the
                list
                add edge (w, i) to ER
        }
return reversed; }
```

Question # 4

Explanation: The idea is to add another array of paths[] and several if conditions to BFS Algo

We will use an array distance[] which will indicate the shortest distance from source vertex to vertex i.

The array of paths[i] basically stores the number of shortest paths from s(source) to vertex i.

path[] will be initialized to zero for all vertices except source vertex, It will be equal to 1. (Same vertex has path to itself which is also shortest).

When we are performing BFS traversal :-

1) if i and j are two vertices and j is a neighbour of i then if distance[j] > distance[i]+1 we shall change the distance[j] to distance[i]+1 and the number of paths will be changed to # of j i.e paths[j] = paths[i]

2) else if distance[j] = distance[i]+1 then we shall sum the number of paths of vertex i and j and mark paths[j] to that sum value because another path has been found

Algo →

```
void BFSTraversal( vector<int>a[], int distance[], int paths[],
                   int s, int n)
{
bool visited[n]; // Keep track of visited vertices
  for(int i=0; i<n; i++) // No vertex visited in the start
      visited[i]= false;
paths[s]=1                    // Base case like
distance[source]= 0
  queue<int> q; // Queue Created for BFS traversal
    q.push(s)
      visited[s] = true // mark it as visited
```

```
while (! q.empty())
{
    int curr = q.front();
    q.pop();

    for (auto z : a[curr])
    {
        if (visited[z] == false)
        {
            visited[z] = true;
            q.push(z);
        }
        if (distance[z] > distance[curr]+1)
        {
            distance[z] = distance[curr]+1;
            paths[z] = paths[curr];
        }
        else if (distance[z] == distance[curr]+1)
        {
            paths[z] = paths[z] + paths[curr];
        }
    }
}
```

// curr is current vertex

// Basically checks all neighbours

// These if/else conditions will work even if z already visited!

// Because z is coming through curr vertex it paths will be equal to curr path vertex

// Basically indicates additional path found

// Because to go to z there was already a path found and this new one is no different in terms of cost the total ways to get to z is the sum of already found + ways to get to curr vertex because we are coming through it in this path

Note

distance[] array will be initialized to ∞

{ paths[] will be initialized to 0

Except distance[s] = 0 and paths[s] = 1

Question #5

(a). Yes the statement is true because Dijkstra's algorithm originally takes $O(m \log V)$ time where V is number of vertices and m is number of edges. As in Kirchoff graph it is very visible that every vertix will have at max 2 edges going from it so total number of edge becomes $m = 2V$

Hence, Time complexity becomes $O(2|V| \log |V|)$

but we chop constants in big-oh so it simply becomes $O(|V| \log |V|)$.

(b).

* Insert all vertices and edges in binary tree. The vertices will be nodes and the edges will represent parent-child relationship
* Traverse through graph, at every node either go left or right according to which one is less weight edge.
* If going to left side doesn't not cause us to reach our destination then go back and choose right side.

~~* Traverse using DFS~~

* Traverse graph and find shortest path ~~using Dijkstra's algorithm~~

Algo

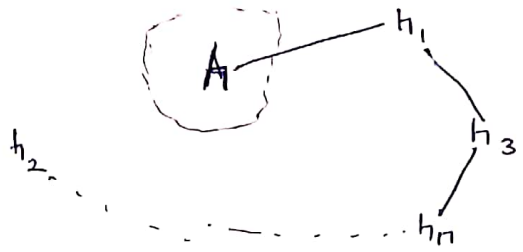
```
int[] sp Korchoff (G=(V,E), s r) {
    BinaryTree<int> tree;
    for (int i=0; i< size of (G); i++) // Takes O(n)
        tree.insert (G(V,E))
Node* root = tree.Root();
Find shortest (root) dis, color)
for (int i=0; i<V; i++) { Find i=0; for each v ∈ V do
{ Color[i] = white;
  dis[i] = ∞ }
}
    if color[v] = white do
```

* dis array is intialized to ∞ and color to white

```
for (int i=0 ; i<|V| ; i++)
{   Dist[i] = Find nextTo ( i, s) dist[i);
}
return Dist;
}

int Find nextTo ( i, s) &dist[i)
    int dist=0;
        Find Shortest (root) dis , color )
{
    if (color[v] == black)
        return d[v];
    else
    {   dis[v]= dis[v-1]+1;
        color[v] = black;  if(root->lf);
        Find Shortest (root -> left)
        if (root ->right),
        Find Shortest( root->right);
    }
}
```

Question# 6 $G=(V,E)$



Multiple Source one destination
Shortest / less risky path have to find in
$O(|V|+E \lg V)$

Solution :-

The idea is to reverse (→ in $O(V)$) all the edges in the original graph and treat the destination vertex as the starting point or source vertex and then using Dijakstra's algorithm compute shortest path to all vertices that were previously starting or source vertices. $O(E \log V)$
[So, Total $O(|V|+E\log V)$]

Algorithm :-

compute Less Risky paths $(G=(V,E), s, w)$
{
// Reverse the graph using algo in Q3 (the
that should take linear time. $O(V)$

// from here onwards find shortest paths using Dijakstra's algorithm $(V \log V)$

for (each $v \in V$)
{
$d[v] = \infty$;
color $[v]$ = white;
}
$d[s] = 0$
pred $[s]$ = NIL;
Q = (queue with all vertices)

while (Non-Empty (Q))
{
v = Extract-min (Q);
for (each $v \in$ adj $[v]$)
if $(d[v] + w(v,v) < d[v])$ {
$d[v] = d[v] + w(v,v)$;

```
        Decrease - Key (a, v, d[v]);
            pred[v] = u;
        }
    }
    color[u] = black;
}
}
```

Total time Complexity = $O(|V| + |E| \log |V|)$