Ritika Kumari
A20414073

Home Work - 4

Problem 1:

Ans: If we perform DFS on an undirected graph and if there is no back edges then we can conclude that the graph does not have any cycle. But if there will be any back edges, then the graph contains a cycle. If there will be no back edges in the graph, then all the edges will became the tree edges and hence we can say that there is no cycle formed in the graph.

Algorithm:

⇒ CycleFound(G) {

(1) let's conside variable cyclefound = false;
& set all nodes to "non-visited".

(2) S = new Stack();

(3) S.push({ initial node,
parent = null })

(4) while (S ≠ empty) do
{

(5) x = S.pop();

(6) if (x is not visited)

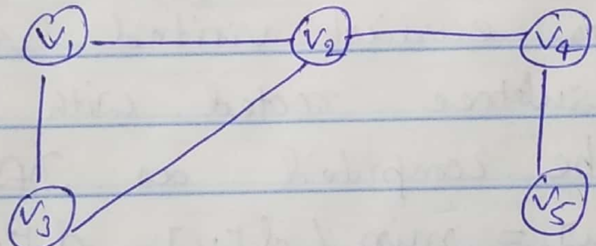
```
(7)   {
        visited [x] = true;
(8)     for each vertex V in G(V, E)
        {
(9)        if (V = x.parent)
           {
(10)          continue // If parent do
                               nothing
           }
(11)       if (V is visited)
           {
(12)          cycleFound = true;
(13)          break;
           }
(14)       S.push( { V, parent = x } )
        }
      }
    }
  }
(15)  return cycleFound;
```

Time - complexity $= O(|V|) + O(|V|)$
↳ step 4 ↳ step 8
$= O(|V|)$

<u>Proof of correctness:</u> We can proof this by using proof of contradiction.
Let's consider there exists a cycle and this above algorithm still return false.
We know that when we apply DFS on the graph, it visits every node of the graph and then we iterate through every neighbour node ~~of~~ from the source node S and recursively call 'cycleFound' method for each of node which is very similar to DFS.
Here, we are keeping track of adjacent parent node which calling traversing every other node.
Therefore ~~when~~ the particular node must be the part of 'visited' set when the cycle completes and 'cycleFound' is set to 'true' in that case, which is contradicting our assumption made initially.
Hence we can say that the algorithm is correct.

Problem 2:

Soln: A bridge in a graph(G) is an edge whose removal disconnects the graph G.
An edge e is a bridge if and only if there is no simple cycle C of G which contains e, because removing edge e wouldn't disconnect the vertices of e in the graph G.
let's take an example:



In the above example, we can see that $V_1 - V_2 - V_3 - V_1$ is a cycle and removal of any edges from this cycle will not disconnect the graph G. For example, if we remove the edge $V_2 - V_3$, then after running DFS on this graph we will get all the vertices $V_3 - V_1 - V_2 - V_4 - V_5$ of this graph G, but if we remove the edge $V_2 - V_4$ or edge $V_4 - V_5$, then the graph will become disconnected, so we can say that the edges $V_2 - V_4$ and $V_4 - V_5$ ~~is a~~

are bridges of this graph.
Hence it is proved that an edge e will be a bridge edge if and only if there is no simple cycle C of G which contains e.

__2nd part:__

To determine all the bridges of G, we assume that G is connected, which means there is a single DFS tree.
let's introduce Low[u] whose value indicates earliest visited vertex reachable from subtree rooted with v. low[u] will be computed as DFS runs.

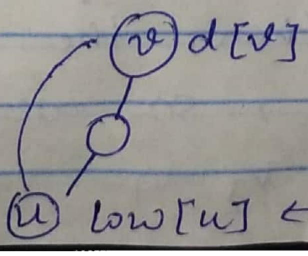
low[u] = min(d[u], d[w]) where w is an ancestor of u and there is a back edge from some descendant of u to w.

The condition for an edge (u,v) ~~is a~~ to be a bridge is low[v] > d[u]

To compute Low[u] now, we perform DFS on the vertex u.

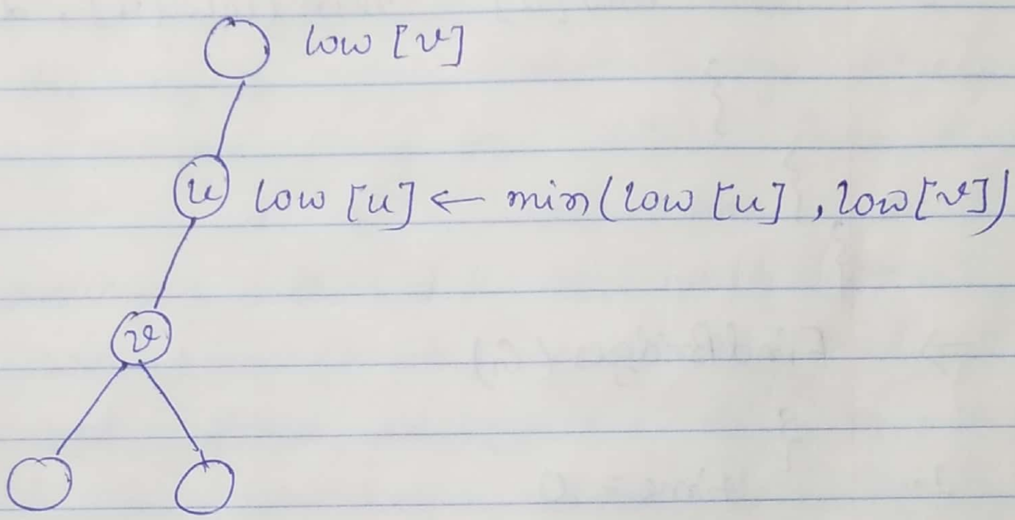
Initialization: low[u] = d[u]

Back edge (u, v): Low[u] = min(low[u], d[v])



low[u] ← min(low[u], d[v])

Tree edge (u, v) : $low[u] = \min(low[u], low[v])$



<u>Pseudocode</u> :

⇒ DFSvisit (u) {

1. mark[u] = visited
2. low [u] = d [u] = ++ time // set visited time & initialize low
3. for each (v in adj (u))

{

4. if (mark [v] = = unvisited) // ie, tree edge

{

5. pred[v] = u // u is parent of v
6. DFSvisit (v)
7. low [u] = min (low[u] , low[v])

~~low[u] ← min(low[u], d~~

}

```
            else if (v != pred[u])          // ie, back edge
            {
                low[u] = min(low[u], d[v])
            }
        }
    }

=> FindBridges(G)
    {
        time = 0
        for each (u in V)
            mark[u] = unvisited          // Initialize

        for each (u in V)
            if (mark[u] == unvisited)
                DFSvisit(u)

        for each (v in V)
        {
            u = pred[v]
            if (u != null and d[v] == low[v])
                return (u,v) as a bridge
        }
    }
```

Proof of correctness : (Proof by Contradiction)

Let's assume that there is a graph G with having cycle C. First assume that the cycle edges are bridge edges. So, now remove any one edge from cycle C and by contradiction, we are assuming that it is a bride edge. Since, we know that when an edge is removed from a cycle, it does not disconnect the vertices which is directly contradicting the bridge property. This is a reason why there can be never an edge from a cycle which will be a bridge edge.

Time Complexity :

$$= O(|E|+|V|) + O|V| + O|V| + O|V|$$

for DFs visit   Step 2   Step 4   Step 7 in FindBridges

$$= O(|E| + |V|)$$ Ans.

Problem 3:

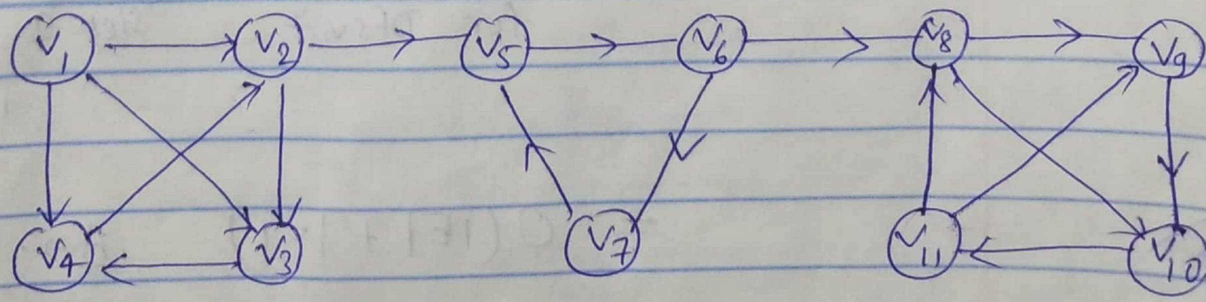
Solⁿ. Algorithm:

1. Let's consider a graph $G(V, E)$
2. Identify all SCC's (Strongly Connected Component) of G.
3. Apply DFS to each of the SCC's indivisually
4. For making component graph from all the SCC's, follow the steps:

   (a) Take any two SCC's and for each edge (u, v) in the component graph G^{SCC}, select any vertex l in u's SCC and any vertex m in v's SCC and add the directed edge (l, m) to E' which will also be an edge in G.

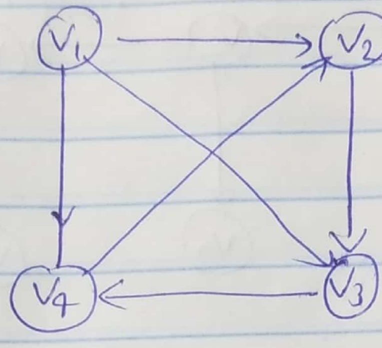
   (b) Repeat this until all the SCC's are connected.

Example: ① Graph $G(V, E)$

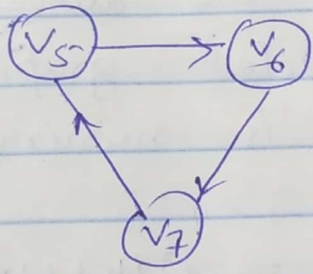


② Identifying all the SCC's :

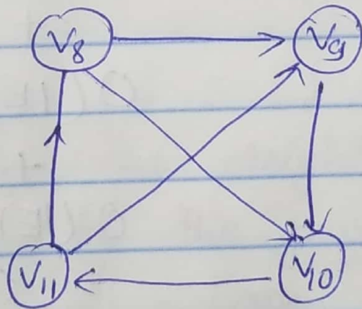
SCC1 =



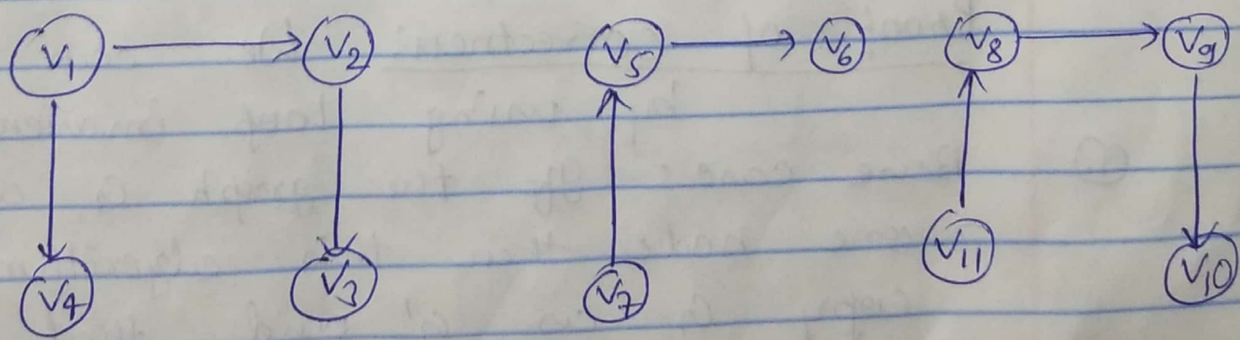
SCC2 =



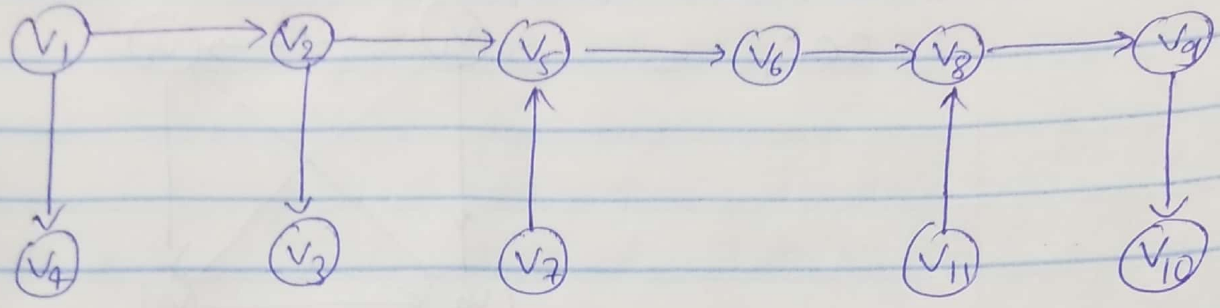
SCC3 =



③ Applying DFS on each SCC's :



(4) Make component graph $G'(V, E')$



After following the above steps, we found another directed graph $G' = (V, E')$ such that $|E'|$ is minimized.

Time Complexity: $= O(|E|+|V|)$ for idefying all SCC's
$+$
$O(|E|+|V|)$ for DFS at step 3
$+$
$O(|E|+|V|)$ for step 4

$= O(|E|+|V|)$ Ans

Proof of correctness: We can prove this by using loop invarient method:

(1) Base case: If the graph G will have one node then this algorithm just copy G as G' and that will be the minimum edge graph.

(2) After identifying SCC's at step 2, we

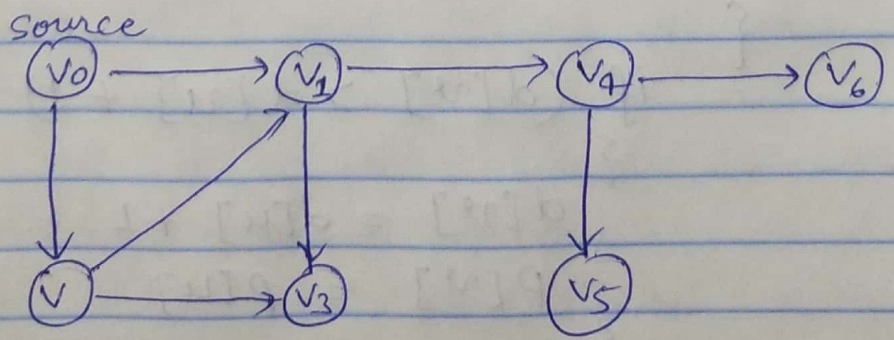
are running DFS on the SCC's to compute spanning tree for each SCC's. In this way, all SCC's will have minimum no. of edges.

(3) Now in step 4, we are iterating over the edges in G and selecting any node from SCC to connect to next SCC to form G' where connection edges will be common for both G and G'.

(4) Since we are iterating over finite no. of edges, the program will ~~definitely~~ terminate after some time.

Problem 4:

Sol^n: Let's consider a weighted directed graph G(V, E). In the graph G, for source vertex, there is one shortest path which is from source to source itself (ie, vertex 0 to vertex 0) and we will consider distance 0 for source vertex.



For finding number of shortest paths in $G(V, E)$, we can use BFS algorithm :

1. Let's create two arrays : d[ ] & p[ ]
where d[ ] represents the ~~s~~ shortest distance from source vertex
and p[ ] represents the no. of shortest paths from source vertex to each vertices.

2. Initialize $d[\,] = \infty$ for each vertex except the source vertex which is equal to 0

3. Initialize $p[\,] = 0$ for all vertex except the source vertex which is equal to 1.

```
(4) for each vertex u ∈ G(V,E)
    {
(5)     for every adjacent v of vertex u
        {
(6)         if (d[v] > d[u] + 1)
            {
(7)             d[v] = d[u] + 1
(8)             p[v] = p[u]
            }
```

2

```
(9)     else if (d[v] = d[u]+1)
            {
(10)            p[v] = p[v] + p[u]
            }
        }
    }
```

<u>Time complexity</u>: $\underset{\text{for step 4}}{O(|V|)} + \underset{\text{for step 5}}{O(|E|)}$

$$= O(|V| + |E|) \quad \underline{\text{Ans}}$$

<u>Proof of Correctness</u>: Proof by Contradiction

At the completion of BFS, if BFS visited v, then $d[v]$ is the distance from s to v where s = source vertex.

Let's assume that there is some vertex with $d()$ not equal to the distance from s.

Let v be a vertex which has the smallest distance from s (source) and u be its predecessor on shortest path from v to u.

According to discussed algorithm,

if $d[v] >$ dist from s to v then

$d[v] =$ (distance from s to u) $+1$

$\Rightarrow d[v] = d[u] + 1$

Now let's assume when u is removed from the queue then it will contradict the above chain of inequalities in any of three cases:

i) If v is not visited yet, then v could have been removed from the queue with $d[v] = d[u] + 1$ which is contradicting to above inequalities.

ii) If v is in the queue, then parent of v has lower distance than v, then $d[v] = d(\text{parent}(v)) + 1 \leq d[u] + 1$ which again contradicts the inequalities.

iii) If v has already been removed from the queue, then $d[v] \leq d[u]$ which contradicts again.

Proved

Hence, we can say that our algorithm is correct.

Scanned by CamScanner