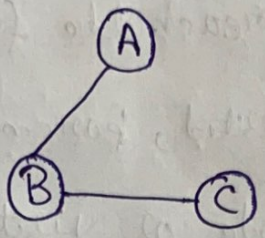
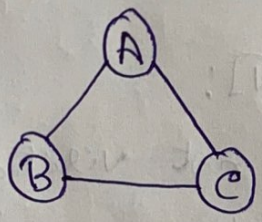
Amitdeb Prosad Bhattacharya A20402789

Problem 1

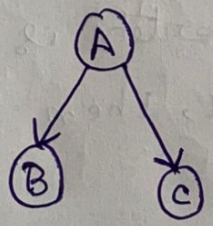
If a DFS yields no back edges then an undirected graph can be called as acyclic graph.



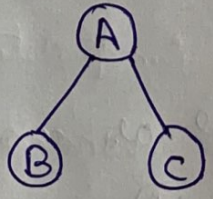
If a DFS yields back edges then an undirected graph can be called as cyclic graph.



If there is a direction between the nodes of a graph then it is called as directed graph.



If the graph is directionless then the graph is called as undirected graph.



We need to run DFS to find a back edge which will determine whether a graph is cyclic or not.

We are going to write an algorithm which will determine an undirected graph as cyclic or acyclic.

```
# A recursive function that uses visited[] and parent to
# detect cycle in subgraph reachable from vertex v.
def isCyclicUtil(self, v, visited, parent):
    # Mark the current node as visited
    visited[v] = True
    # Recur for all the vertices adjacent to this
    # vertex
    for i in self.graph[v]:
        # If the node is not visited then recurse on it
        if visited[i] == False:
            if(self.isCyclicUtil(i, visited, v)):
                return True
        # If an adjacent vertex is visited and not parent
        # of current vertex, then there is a cycle
        elif parent != i:
            return True
    return False

# Returns true if the graph contains a cycle, else false.
def isCyclic(self):
    # Mark all the vertices as not visited
    visited = [False]*[self.V]
    # Call the recursive helper function to detect cycle
    in different DFS trees
```

```
for i in range(self.V):
    if visited[i] == False:
    # If already visited ~~don't~~ don't recur
        if (self.isCyclicUtil(i, visited, -1) == True:
            return True
return False
```

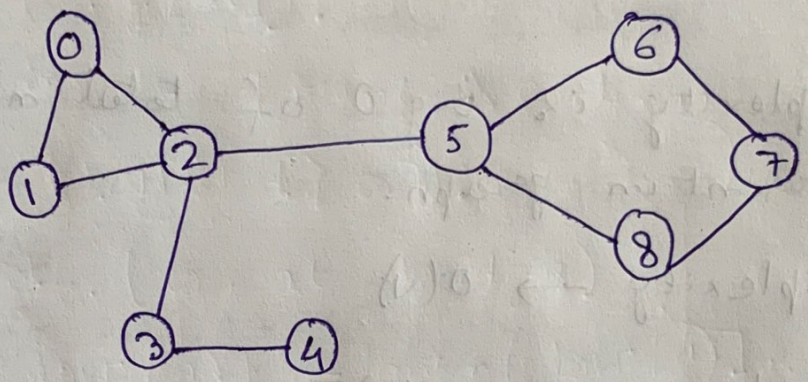
Space complexity is Big O of total number of Vertex present in graph.

Space complexity $\longrightarrow O(V)$

Time complexity $\longrightarrow O(V)$

Problem 2.

Let us consider a $G = (V, E)$ be an undirected multigraph. A bridge is an edge whose removal disconnects G. A bridge is any edge in a graph whose removal increases the number of connected components.



Here there are three bridges i.e. edge joining nodes 2 and 5, edge joining nodes 2 and 3, and edges joining nodes 3 and 4. If we remove these edges then the graph will be seperated into two parts.

Here there are three articulation point 2, 5, 3.

Bridges and articulation points are important in graph theory because they often hint at weak points, bottlenecks or vulnerabilities in a graph. Therefore, it's important to be able to quickly find/detect when and where these occur.

Both problems are related so we will develop

an algorithm to find bridges and then modify it slightly to find articulation points.

Algorithm ⟹

```
id = 0
g = adjacency list with undirected edges
n = size of the graph
# In these arrays index i represents node i
ids = [0, 0, ..., 0, 0]          # length n
low = [0, 0, ..., 0, 0]          # length n
visited = [false, ..., false]    # length n

function findBridges():
    bridges = []
    # Finds all bridges in the graph across
    # various connected components.
    for (i=0; i<n; i=i+1):
        if (!visited[i]):
            dfs(i, -1, bridges)
    return bridges
# Perform DFS to find bridges.
# at = current node, parent = previous node
# The bridges list is always of even length
# and indexes (2*i, 2*i+1) from a bridge.
# For example, nodes at indexes (0,1) are a
# bridge, (2, 3) is another etc.
```

```
function dfs(at, parent, bridges):
    visited[at] = true
    id = id + 1
    low[at] = ids[at] = id

    # For each edge from node 'at' to node 'to'
    for (to : g[at]):
        if to == parent : continue
        if (!visited[to]):
            dfs(to, at, bridges)
            low[at] = min(low[at], low[to])
            if (ids[at] < low[to]):
                bridges.add(at)
                bridges.add(to)
        else:
            low[at] = min(low[at], ids[to])
```

- Find out the cycles
- Now for each edge in every cycle, remove one edge at a time.
- After removing edge, apply DFS.
- If we traverse all vertices after applying DFS, then the edge removed is not a bridge.
- Else the edge removed is a bridge.

We are doing one DFS to level all the nodes plus V more DFSs to find all the low-link values, giving roughly as $O(V(V+E))$ but we are able to do better by updating the low-link values in one pass for $O(V+E)$.

Proof by Contradiction $\Rightarrow$

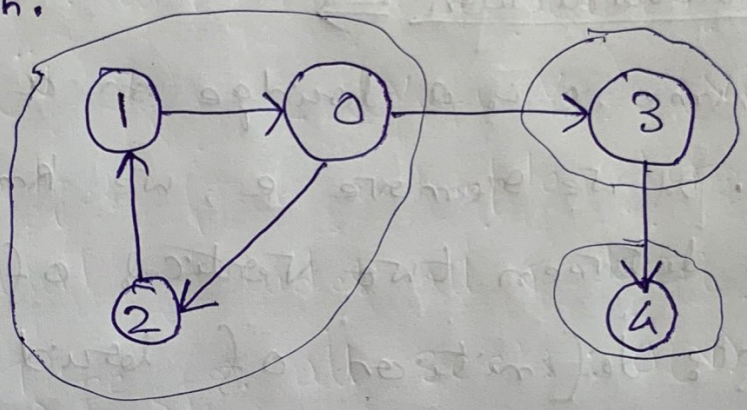
Let us assume e is a bridge on a cycle of graph G_2. If we remove e, we know there is a path between two vertices of e, which contradicts definition of bridge.

Now, let us say e lies on no cycle in G_2 graph. If we remove e from G_2, and we suppose there is a path between vertices of e after removing e, as e is not a bridge. Adding e again, as there is a path between two vertices of e which is not through e, there is a cycle in G_2 which is passed through e, which is contradicted by e is not a bridge. Therefore e is a bridge of G_2.

Problem 3

Strongly connected components ⇒

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component of a directed graph is a maximal strongly connected subgraph.



The major idea is to replace the edges within each SCC by one directed cycle and then removing redundant edges between SCCs. Since there must be at least K edges within an SCC that has K vertices, a single directed cycle of K edges gives the K-vertex SCC with the fewest possible edges.

Algorithm ⇒

Step 1: All SCC of graph G_r should be identified

Time: $\Theta(V+E)$ by using SCC(G_r)

Step 2: Component graph G_r^{SCC} should be formed.

~~Step 2: Start~~ Time: $O(V+E)$

Step 3: Start with $E' = \phi$.

Time: $O(1)$

Step 4: For each SCC of G_r, let the vertices in the SCC be $v_1, v_2, \ldots, v_k$, and add to E' the directed edges (v_1, v_2), $(v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, v_1)$. These edges form a simple, directed cycle that includes all vertices of the SCC.

Time for all SCC's: $O(V)$.

Step 5: For each edge (u, v) in component graph G_r^{SCC}, select any vertex x in u's SCC and any vertex y in v's SCC and add the directed edge (x, y) to E'.

Time: $O(E)$.

SCC(G_r):

1. Call DFS(G_r) to compute finishing times $u.f$ for each vertex u.
2. Compute G_r^T
3. Call DFS(G_r^T), but in the main loop of DFS, consider the vertices in order

of decreasing u.f

4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component.

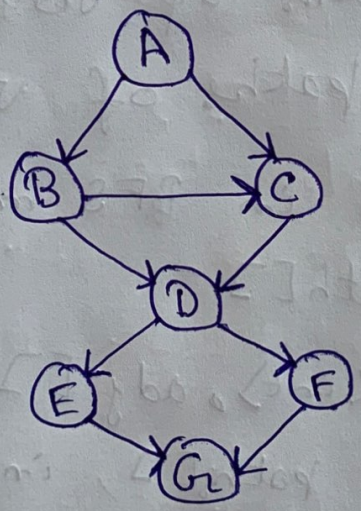
From algorithm we can say total time is $\theta(V+E)$.

Problem 4

The given directed graph, can be cyclic or acyclic. The number of shortest paths from a given vertex to each of the vertices should be printed. Let us take an example of below graph there is one shortest path vertex A to vertex A (from each vertex there is a single shortest path to itself), one shortest path between vertex A to vertex C, and four different shortest paths from vertex A to vertex G.

1. A → B → E → G
2. A → B → D → G
3. A → C → D →
4. A

1) A → B → D → E → G
2) A → B → D → F → G
3) A → C → D → E → G
4) A → C → D → F → G



We are going to use BFS, we are going to use two array dist[] and paths[].

dist[] → shortest path

path[] → number of different shortest path from source to vertex.

At first all the element in dist[] are infinity except source vertex which is equal to 0, since the distance to source vertex from itself is 0, and all the elements in paths[] are 0 except source vertex which is equal to 1.

1) if dist[Y] > dist[X]+1 decrease the dist[Y] to dist[X]+1 and assign the number of paths of vertex X to number of paths of vertex Y.

2) else if dist[Y] = dist[X]+1, then add number of paths of vertex X to the number of paths of vertex Y.

```
// Traverses graphs in BFS manner.
// dist[] and path[]
void BFS (vector<int>, adj[], int src, int dist[],
          int paths[], int n)
{
    bool visited[n];
    for (int i=0; i<n; i++)
        visited[i] = false;
```

```
dist[src] = 0;
paths[src] = 1;
queue<int> q;
q.push(src);
visited[src] = true;
while (!q.empty())
{
    int curr = q.front();
    q.pop();
    // for all neighbours of current vertex :
    for (auto x : adj[curr])
    {
        if (visited[x] == false)
        {
            q.push(x);
            visited[x] = true;
        }
    // check if there is a better path.
    if (dist[x] > dist[curr]+1)
    {
        dist[x] = dist[curr] + 1;
        paths[x] = paths[curr];
    }
    else if (dist[x] == dist[curr]+1)
        paths[x] += paths[curr];
    }
}
```

```
// function to find number of different
// shortest paths from given vertex S.
// n -> no. of vertices.
void findShortestPaths(vector<int> adj[],
                       int s, int n)
{
    int dist[n], paths[n];
    for (int i = 0; i < n; i++)
        dist[i] = INT_MAX;
    for (int i=0; i<n; i++)
        paths[i] = 0;
    BFS(adj, s, dist, paths, n);
    print("no. of sht. path");
    for (int i=0; i<n; i++)
        print("paths");
```

The time complexity is O(V+E).