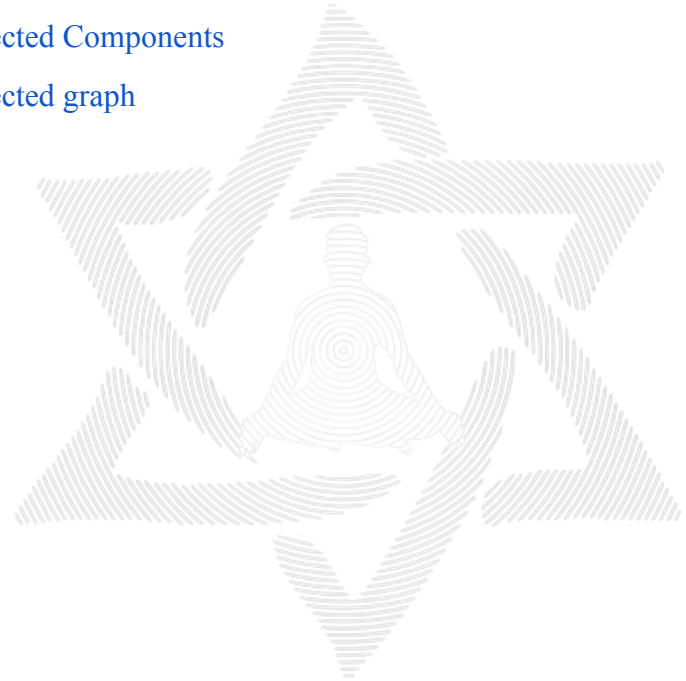
Graphs
Lesson 8

1. Find if there is a path between two vertices in a directed graph
2. Connectivity in a directed graph
3. Tarjan's Algorithm to find strongly connected Components
4. Articulation Points (or Cut Vertices) in a Graph
5. Bridges in a graph
6. Biconnected Components
7. Biconnected graph



Find if there is a path between two vertices in a directed graph

Given a Directed Graph and two vertices in it, check whether there is a path from the first given vertex to the second.

Trade-offs between BFS and DFS: Breadth-First search can be useful to find the shortest path between nodes, and depth-first search may traverse one adjacent node very deeply before ever going into immediate neighbours.

Complexity Analysis:

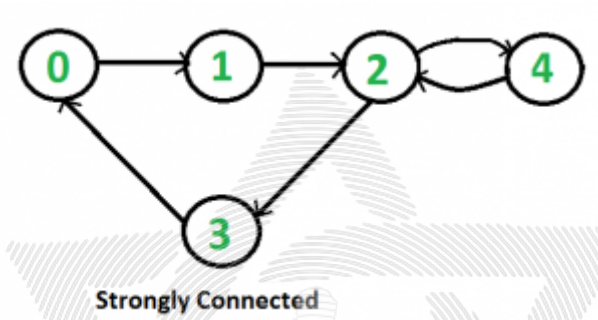
- Time Complexity: O(V+E) where V is the number of vertices in the graph and E is the number of edges in the graph.
- Space Complexity: O(V).
  There can be at most V elements in the queue. So the space needed is O(V).

```
1.    vector<int>bfsOfGraph(int V, vector<int> adj[])
2.          {
3.             vector<int>v;
4.             bool visited[V]={false};
5.             queue<int>q;
6.             q.push(0);
7.             visited[q.front()]=true;
8.             while(!q.empty())
9.             {
10.             v.push_back(q.front());
11.
12.           for(int i=0;i<adj[q.front()].size();i++)
13.           {
14.              if(visited[adj[q.front()][i]] == false)
```

```
15.                 {
16.                     q.push(adj[q.front()][i]);
17.                     visited[adj[q.front()][i]]=true;
18.                 }
19.             }
20.             q.pop();
21.               }
22.               return v;
23.               // Code here
24.           }
```

Check if a graph is strongly connected (Kosaraju using DFS)

Given a directed graph, find out whether the graph is strongly connected or not. A directed graph is strongly connected if there is a path between any two pairs of vertices. For example, the following is a strongly connected graph.



It is easy for an undirected graph, we can just do a BFS and DFS starting from any vertex. If BFS or DFS visits all vertices, then the given undirected graph is connected. This approach won't work for a directed graph. For example, consider the following graph which is not strongly connected. If we start DFS (or BFS) from vertex 0, we can reach all vertices, but if we start from any other vertex, we cannot reach all vertices.

For a directed graph :

1)A simple idea is to use a all pair shortest path algorithm like Floyd Warshall or find Transitive Closure of graph. Time complexity of this method would be O(v3).

2)We can also do DFS V times starting from every vertex. If any DFS doesn't visit all vertices, then the graph is not strongly connected. This algorithm takes O(V*(V+E)) time which can be the same as transitive closure for a dense graph.

3)Kosaraju’s

First define a Condensed Component Graph as a graph with ≤V nodes and ≤E edges, in which every node is a Strongly Connected Component and there is an edge from C to C′,where C and C′ are Strongly Connected Components, if there is an edge from any node of C to any node of C′.

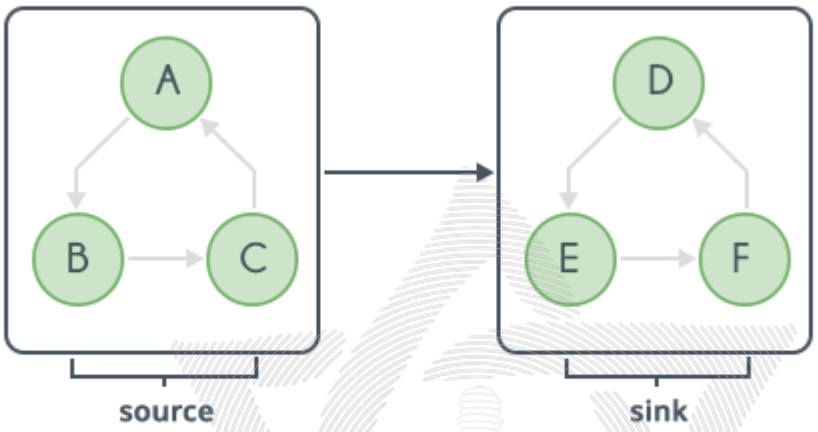


Fig 2. Condensed Component Graph

It can be proved that the Condensed Component Graph will be a Directed Acyclic Graph(DAG). To prove it, assume the contradiction that it is not a DAG, and there is a cycle. Now observe that on the cycle, every strongly connected component can reach every other strongly connected component via a directed path, which in turn means that every node on the cycle can reach every other node in the cycle, because in a strongly connected component every node can be reached from any other node of the component. So if there is a cycle, the cycle can be replaced with a single node because all the Strongly Connected Components on that cycle will form one Strongly Connected Component.Therefore, the Condensed Component Graph will be a
DAG. Now, a DAG has the property that there is at least one node with no incoming edges and at least one node with no outgoing edges. Call the above 2 nodes as Source and Sink nodes.

1) Initialize all vertices as not visited.

2) Do a DFS traversal of a graph starting from any arbitrary vertex v. If DFS traversal doesn't visit all vertices, then return false.

3) Reverse all arcs (or find transpose or reverse of graph)

4) Mark all vertices as not-visited in the reversed graph.

5) Do a DFS traversal of a reversed graph starting from the same vertex v (Same as step 2). If DFS traversal doesn't visit all vertices, then return false. Otherwise return true.

The idea is, if every node can be reached from a vertex v, and every node can reach v, then the graph is strongly connected. In step 2, we check if all vertices are reachable from v. In step 4, we check if all vertices can reach v (In the reversed graph, if all vertices are reachable from v, then all vertices can reach v in the original graph).

```
1.  vector<int>adj2[5001];
2.          void reverse(int V, vector<int> adj[])
3.          {
4.             //vector<int> adj2[V];
5.
6.             for(int i=0;i<V;i++)
7.             {
8.                for(int j=0;j<adj[i].size();j++)
9.                {
10.                  adj2[adj[i][j]].push_back(i);
11.               }
12.            }
13.         }
14.         void DFS(int src , vector<int> adj[] , stack<int>&st , int visited[])
15.         {
16.            visited[src]=1;
17.            for(int i=0;i<adj[src].size();i++)
18.            {
19.               if(!visited[adj[src][i]])
20.               {
21.                  DFS(adj[src][i] , adj , st , visited);
22.               }
```

```
            }
            st.push(src);
        }
        //Function to find number of strongly connected components in the graph.
int kosaraju(int V, vector<int> adj[])
{
    int visited[V]={0};
    stack<int>st;
    //step 1 dfs and maintain a stack
    for(int i=0;i<V;i++)
    {
        if(!visited[i])
        {
            DFS(i,adj,st , visited);
        }
    }
    //step 2 reverse the graph
    reverse(V,adj);
     stack<int>st2;
    for(int i=0;i<V;i++)
        visited[i]=0;
    int count=0;

    //step 3 traverse the stack and graph
    while(!st.empty())
    {
        int x=st.top();
        if(!visited[x])
            {
                DFS(x,adj2,st2,visited);
                count++;
            }
        st.pop();
    }
    return count;
    //code here
}
```

Tarjan's Algorithm to find Strongly Connected Components

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.

Tarjan Algorithm is based on the following facts:

1. DFS search produces a DFS tree/forest

2. Strongly Connected Components form subtrees of the DFS tree.

3. If we can find the head of such subtrees, we can print/store all the nodes in that subtree (including head) and that will be one SCC.

4. There is no back edge from one SCC to another (There can be cross edges, but cross edges will not be used while processing the graph).

To find the head of a SCC, we calculate the disc and low array . low[u] indicates the earliest visited vertex (the vertex with minimum discovery time) that can be reached from the subtree rooted with u.

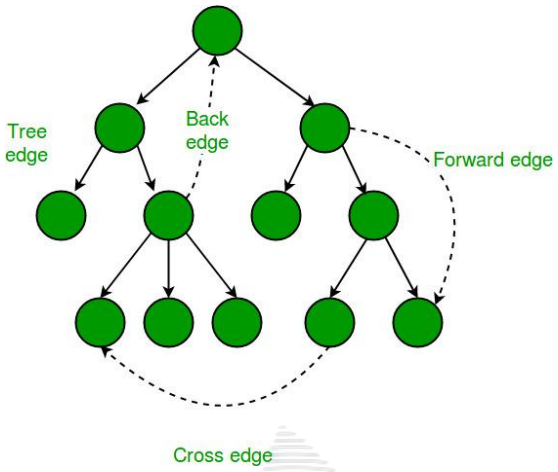
A node u is head if disc[u] = low[u].

Case1 (Tree Edge): If node v is not visited already, then after DFS of v is complete, then minimum of low[u] and low[v] will be updated to low[u].

low[u] = min(low[u], low[v]);

Case 2 (Back Edge): When child v is already visited, then the minimum of low[u] and Disc[v] will be updated to low[u].

low[u] = min(low[u], disc[v]);



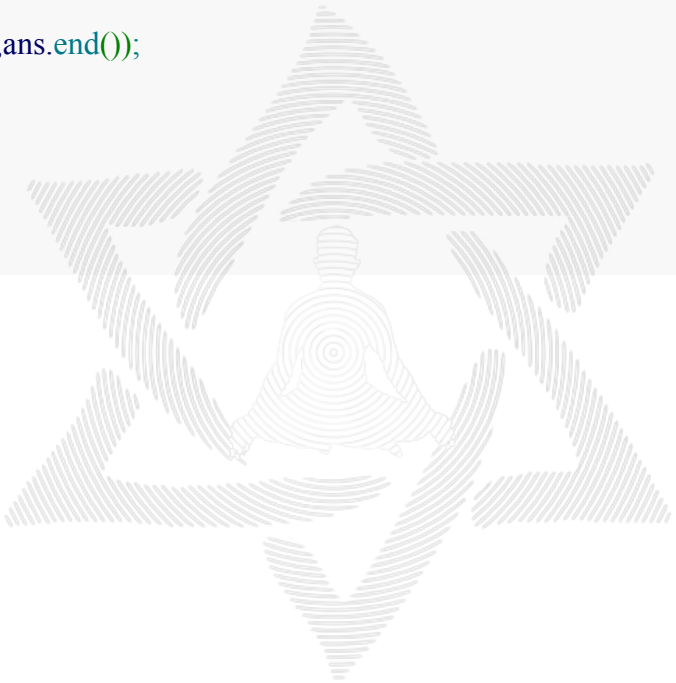
To make sure, we don't consider cross edges, when we reach a node which is already visited, we should process the visited node only if it is present in the stack, else ignore the node.

Time Complexity: The above algorithm mainly calls DFS, DFS takes O(V+E) for a graph represented using an adjacency list.

```
1.  void traverse(int i,vector<int> adj[],vector<vector<int>>&ans,vector<int>&low,vector<int>&disc,stack<int>&s,vector<bool>&stack_val , int &timer)
2.  {
3.    disc[i]=low[i]=timer;
4.    timer++;
5.    s.push(i);
6.    stack_val[i]=true;
7.
8.    for(auto v:adj[i])
9.    {
10.     if(disc[v]==-1)
11.     {
12.       traverse(v,adj,ans,low,disc,s,stack_val,timer);
13.       low[i]=min(low[i],low[v]);
14.     }
15.     else if(stack_val[v])
16.     low[i]=min(low[i] , disc[v]);
```

```
    }
    if(low[i]==disc[i])
    {
        vector<int>val;
        while(!s.empty()&&s.top()!=i)
        {
            val.push_back(s.top());
            stack_val[s.top()]=false;
            s.pop();
        }
        if(!s.empty())
        {
            val.push_back(s.top());
            stack_val[s.top()] = false;
            s.pop();
            sort(val.begin(), val.end());
            ans.push_back(val);
        }
    }

}
class Solution
{
        public:
    //Function to return a list of lists of integers denoting the members
    //of strongly connected components in the given graph.
    vector<vector<int>> tarjans(int V, vector<int> adj[])
    {
        //vector for storing the scc values
        vector<vector<int>>ans;

        //low:node with lowest discovery time accessible
        vector<int>low(V,-1);

        //disc:discovery time og all the nodes
        vector<int>disc(V,-1);

```

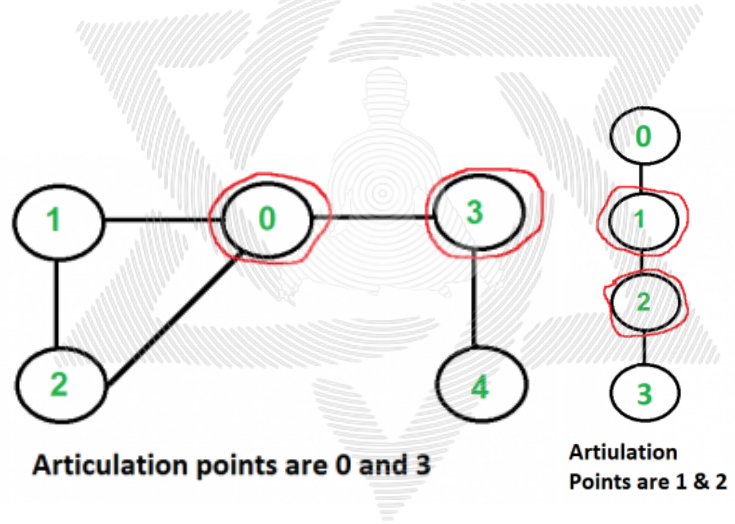
```
54.         //stack for storing dfs
55.         stack<int>s;
56.
57.         //array for accessing stack elements in O(1)
58.         vector<bool>stack_val(V,false);
59.         int timer=0;
60.         //looping through all nodes once
61.         for(int i=0;i<V;i++)
62.         {
63.            if(disc[i]==-1)
64.                traverse(i,adj,ans,low,disc,s,stack_val,timer);
65.         }
66.         sort(ans.begin(),ans.end());
67.         return ans;
68.         //code here
69.       }
70.   };
```

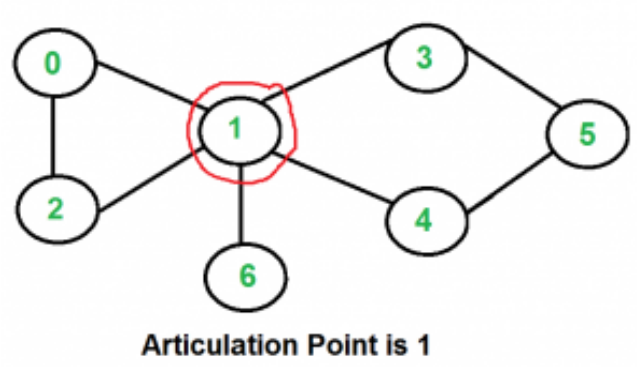


Articulation Points (or Cut Vertices) in a Graph

A vertex in an undirected connected graph is an articulation point (or cut vertex) if removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more components. They are useful for designing reliable networks.

For a disconnected undirected graph, an articulation point is a vertex removal which increases the number of connected components.





A simple approach is to one by one remove all vertices and see if removal of a vertex causes a disconnected graph. Following are the steps of a simple approach for a connected graph.

1) For every vertex v, do following

…..a) Remove v from graph

..…b) See if the graph remains connected (We can either use BFS or DFS)

…..c) Add v back to the graph

Time complexity of above method is O(V*(V+E)) for a graph represented using an adjacency list.

A O(V+E) algorithm to find all Articulation Points (APs)

The idea is to use DFS (Depth First Search). In DFS, we follow vertices in tree form called DFS tree. In the DFS tree, a vertex u is the parent of another vertex v, if v is discovered by u (obviously v is an adjacent of u in the graph). In DFS tree, a vertex u is an articulation point if one of the following two conditions is true.

1) u is the root of the DFS tree and it has at least two children.

2) u is not the root of DFS tree and it has a child v such that no vertex in subtree rooted with v has a back edge to one of the ancestors (in DFS tree) of u.

```
1.  void traverse(int i,vector<int>adj[],vector<int>&low,vector<int>&disc,vector<int>&ans,int parent,int &timer,vector<bool>&visited)
2.      {
3.          //checking for child nodes
4.          int child=0;
5.          //updating the timer
6.          timer++;
7.          low[i]=timer;
8.          disc[i]=timer;
9.
10.         //visiting the node
```

```
11.         visited[i]=true;
12.
13.         for(int j=0;j<adj[i].size();j++)
14.         {
15.             int x=adj[i][j];
16.             if(!visited[x])
17.             {
18.                 //if not visited then its a child node
19.                 child++;
20.                 //traversing the child node
21.                 traverse(x,adj,low,disc,ans,i,timer,visited);
22.
23.                 //updating the low
24.                 low[i]=min(low[i],low[x]);
25.
26.                 //if not a root node and
27.                 //if there is no back edge from this child node 'x' to ancestor of i ,
28.                 //then i is AP
29.                 if(parent!=-1 && low[x]>=disc[i]&& ans[i]==0)
30.                     ans[i]=1;
31.             }
32.             //not going back to its parent
33.             //comint to a visited node means its a back edge
34.             else if(x!=parent)
35.             {
36.                 low[i]=min(low[i],disc[x]);
37.             }
38.         }
39.         if(parent==-1 && child>1 && ans[i]==0)
40.             ans[i]=1;
41.     }
42.     vector<int> articulationPoints(int V, vector<int>adj[])
43.     {
44.         //creating a vector to store result
45.         vector<int>ans(V,0);
46.
47.         //low:node with lowest discovery time accessible
```

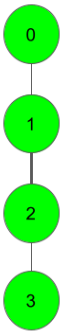
```
48.         vector<int>low(V,-1);
49.
50.         //disc:discovery time
51.         vector<int>disc(V,-1);
52.
53.         vector<bool>visited(V,false);
54.
55.         int parent=-1,timer=0;
56.
57.         for(int i=0;i<V;i++)
58.         {
59.             if(!visited[i])
60.                 traverse(i,adj,low,disc,ans,parent,timer,visited);
61.         }
62.
63.         vector<int>AP;
64.         for(int i=0;i<V;i++)
65.         {
66.             if(ans[i])
67.                 AP.push_back(i);
68.         }
69.         if(AP.empty()){
70.             AP.push_back(-1);
71.             return AP;}
72.         sort(AP.begin(),AP.end());
73.         return AP;
74.         // Code here
75.     }
```

Bridge Edge in Graph

Given an undirected graph of V vertices and E edges and another edge (c-d), the task is to find if the given edge is a bridge in the graph, i.e., removing the edge disconnects the graph.

Example 1:

Input:



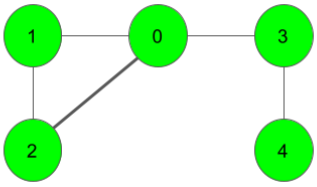
c = 1, d = 2

Output:

1

Example 2:

Input:



c = 0, d = 2

Output:

0

Expected Time Complexity: O(V + E).

Expected Auxiliary Space: O(V).

Constraints:

$1 \leq V,E \leq 10^5$

$0 \leq c, d \leq V-1$

Biconnected Graph

Given a graph with n vertices, e edges and an array arr[] denoting the edges connected to each other, check whether it is Biconnected or not.

Note: The given graph is Undirected.

Example 1:

Input:

n = 2, e = 1

arr = {0, 1}

Output:

1

Example 2:

Input:

n = 3, e = 2

arr = {0, 1, 1, 2}

Output:

0

Expected Time Complexity: O(n+e)

Expected Auxiliary Space: O(n)

Constraints:

1 <= e <= 100

2 <= n <= 100

```
1.  void traverse(int i,vector<int>adj[],vector<int>&low,vector<int>&disc,vector<int>&ans,int parent,int &timer,vector<bool>&visited)
2.  {
3.      //checking for child nodes
4.      int child=0;
5.      //updating the timer
6.      timer++;
7.      low[i]=timer;
8.      disc[i]=timer;
9.
10.     //visiting the node
11.     visited[i]=true;
12.
13.     for(int j=0;j<adj[i].size();j++)
14.     {
15.         int x=adj[i][j];
16.         if(!visited[x])
17.         {
18.             //if not visited then its a child node
19.             child++;
20.             //traversing the child node
21.             traverse(x,adj,low,disc,ans,i,timer,visited);
22.
23.             //updating the low
24.             low[i]=min(low[i],low[x]);
25.
26.             //if not a root node and
27.             //if there is no back edge from this child node 'x' to ancestor of i ,
28.             //then i is AP
29.             if(parent!=-1 && low[x]>=disc[i]&& ans[i]==0)
30.                 ans[i]=1;
31.         }
32.         //not going back to its parent
33.         //comint to a visited node means its a back edge
34.         else if(x!=parent)
35.         {
36.             low[i]=min(low[i],disc[x]);
```

```
            }
        }
        if(parent==-1 && child>1 && ans[i]==0)
            ans[i]=1;
    }
    vector<int> articulationPoints(int V, vector<int>adj[])
    {
        //creating a vector to store result
        vector<int>ans(V,0);

        //low:node with lowest discovery time accessible
        vector<int>low(V,-1);

        //disc:discovery time
        vector<int>disc(V,-1);

        vector<bool>visited(V,false);

        int parent=-1,timer=0;

        for(int i=0;i<V;i++)
        {
            if(!visited[i])
                traverse(i,adj,low,disc,ans,parent,timer,visited);
        }

        vector<int>AP;
        for(int i=0;i<V;i++)
        {
            if(ans[i])
                AP.push_back(i);
        }
        /*if(AP.empty()){
            AP.push_back(-1);
            return AP;}
         sort(AP.begin(),AP.end());*/
        return AP;
```

```
74.         // Code here
75.     }
76.     int biGraph(int arr[], int n, int e)
77.     {
78.         vector<int>adj[n];
79.         if(e==1&&n==2)
80.             return 1;
81.         for(int i=0;i<2*e-1;i+=2)
82.         {
83.             adj[arr[i]].push_back(arr[i+1]);
84.             adj[arr[i+1]].push_back(arr[i]);
85.         }
86.         for(int i=0;i<n;i++)
87.         {
88.             if(adj[i].size()==0)
89.                 return 0;
90.         }
91.         vector<int>AP=articulationPoints(n,adj);
92.         if(AP.empty())
93.             return 1;
94.         return 0;
95.         // code here
96.     }
```