**Algorithm on Graph**

Q.1 <https://leetcode.com/problems/critical-connections-in-a-network/>

**Approach**

1. **Graph Representation**:
   * Use an adjacency list to represent the graph. This is an efficient way to store the graph since we only need to store each connection once and adjacency lists are easy to traverse.
2. **Depth-First Search (DFS) with Tarjan's Algorithm**:
   * The goal of Tarjan's algorithm in this context is to find all the bridges in an undirected graph.
   * We will perform a DFS traversal starting from any unvisited node. During this traversal, we'll maintain:
     + tin[u]: The discovery time of node u (the time when the node was first visited).
     + low[u]: The lowest discovery time reachable from node u.
3. **Tarjan's Conditions for Bridges**:
   * During DFS, if there exists an edge (u, v) such that low[v] > tin[u], then the edge (u, v) is a bridge.
   * This condition means there is no way to reach the ancestor of u from v or any of its descendants without using the edge (u, v).
4. **Algorithm Steps**:
   * **Initialize Data Structures**:
     + adj: An adjacency list to store the graph.
     + tin and low: Vectors initialized to store discovery and low times for each node.
     + vis: A vector to keep track of visited nodes.
     + bridges: A list to store all the critical connections (bridges).
   * **Build the Graph**:
     + For each connection (u, v), add v to the adjacency list of u and vice versa.
   * **DFS Function**:
     + Perform DFS starting from an unvisited node.
     + Set the discovery time and low time of the node.
     + For every connected node (neighbor), if it's not the parent:
       - If the neighbor hasn't been visited, recursively call DFS for it.
       - Update the low time of the current node.
       - Check if the current edge is a bridge.
     + If the neighbor is already visited and isn't the parent, update the low time using the discovery time of the neighbor.
   * **Main Function**:
     + Call DFS from every unvisited node to ensure all components are covered.

|  |
| --- |
| class Solution {  private:      int timer=1;      void dfs(int node,int parent,              vector<int> adj[],              vector<int>& vis,              vector<int> &low,              vector<int> &tin,vector<vector<int>> &bridges)      {          vis[node]=1;          low[node]=tin[node]=timer;          timer++;          for(auto it:adj[node])          {              if(it==parent) continue;              if(vis[it]==0)              {                  dfs(it,node,adj,vis,low,tin,bridges);                  low[node]=min(low[it],low[node]);                  if(low[it] > tin[node])                  {                      bridges.push\_back({it,node});                  }              }              else              {                  low[node]=min(low[it],low[node]);              }          }      }  public:      vector<vector<int>> criticalConnections(int n, vector<vector<int>>& connections) {          vector<int> vis(n,0);          vector<int> adj[n];          vector<int> low(n),tin(n);          vector<vector<int>> bridges;          for(auto it:connections)          {              adj[it[0]].push\_back(it[1]);              adj[it[1]].push\_back(it[0]);          }          dfs(0,-1,adj,vis,low,tin,bridges);          return bridges;      }  }; |

TC:O(V+2E) SC:O(3N)

Q.2 <https://www.geeksforgeeks.org/problems/articulation-point-1/1?utm_source=youtube&utm_medium=collab_striver_ytdescription&utm_campaign=articulation-point>

 **Definition**:

* An **articulation point** (or cut vertex) is a node in a graph whose removal increases the number of connected components.

 **DFS Traversal**:

* Use Depth-First Search (DFS) to explore the graph.
* Track discovery time (tin[]) and the lowest point reachable (low[]) for each node.

 **Key Conditions**:

* **Non-root Node**: A node u is an articulation point if there exists a child v such that low[v] >= tin[u].
* **Root Node**: A root node is an articulation point if it has more than one child in the DFS tree.

 **Algorithm**:

* Start DFS from an unvisited node.
* For each node, recursively explore its adjacent nodes.
* Update low[] based on the child's low[] value.
* After DFS, mark nodes as articulation points based on the conditions above.

|  |
| --- |
| class Solution {  private:  int timer=0;  void dfs(int node,int parent,vector<int> adj[],vector<int>&vis,int low[],int tin[],vector<int>&mark)  {  vis[node]=1;  tin[node]=low[node]=timer;  timer++;  int child=0;  for(auto it:adj[node])  {  if(parent==it) continue;  if(!vis[it])  {  dfs(it,node,adj,vis,low,tin,mark);  low[node]=min(low[node],low[it]);    if(low[it] >= tin[node] && parent!=-1)  {  mark[node]=1;  }  child++;  }  else  {  low[node]=min(low[node],tin[it]);  }    }    if(child > 1 && parent==-1)  {  mark[node]=1;  }  }  public:  vector<int> articulationPoints(int n, vector<int>adj[]) {  // Code here  vector<int> vis(n,0);  int tin[n];  int low[n];  vector<int> mark(n,0);    for(int i=0;i<n;i++)  {  if(!vis[i])  {  dfs(i,-1,adj,vis,low,tin,mark);  }  }    vector<int> ans;  for(int i=0;i<n;i++)  {  if(mark[i]==1) ans.push\_back(i);  }    if(ans.size()==0) return {-1};    return ans;  }  }; |

TC:O(V+2E)+O(N)+O(N) SC:O(3N)

Q.3 <https://www.naukri.com/code360/problems/count-strongly-connected-components-kosaraju-s-algorithm_1171151?leftPanelTabValue=SUBMISSION>

 **Graph Construction**:

* Build the adjacency list from the given edges.

 **First DFS Pass (Original Graph)**:

* Perform DFS on each unvisited node.
* Push nodes onto a stack in the order of their finishing times.

 **Transpose the Graph**:

* Reverse the direction of all edges in the graph.

 **Second DFS Pass (Transposed Graph)**:

* Pop nodes one by one from the stack.
* Perform DFS on the transposed graph for each unvisited node to identify SCCs.
* Count each DFS call as a separate SCC.

 **Return Result**:

* The total count of SCCs is returned as the result.

|  |
| --- |
| #include<stack>  void dfs(int node,vector<int>& vis,stack<int> &st,vector<int> adj[])      {          vis[node]=1;            for(auto it:adj[node])          {              if(!vis[it])              {                  dfs(it,vis,st,adj);              }          }            st.push(node);      }        void dfs2(int node,vector<int> adjT[],vector<int> &vis)      {          vis[node]=1;            for(auto it:adjT[node])          {              if(!vis[it])              {                  dfs2(it,adjT,vis);              }          }      }  int stronglyConnectedComponents(int V, vector<vector<int>> &edges)  {      // Write your code here.      vector<int> vis(V,0);          stack<int> st;          vector<int> adj[V];          for(auto it:edges)          {              adj[it[0]].push\_back(it[1]);          }            for(int i=0;i<V;i++)          {              if(!vis[i])              {                  dfs(i,vis,st,adj);              }          }            vector<int> adjT[V];            for(int i=0;i<V;i++)          {              vis[i]=0;              for(auto it:adj[i])              {                  //i-->it make it it--> i                  adjT[it].push\_back(i);                }          }            //step 3          int scc=0;            while(!st.empty())          {              int node=st.top();              st.pop();                if(!vis[node])              {                  dfs2(node,adjT,vis);                   scc++;              }              }            return scc;  } |

TC:O(V+E) SC:O(V)