

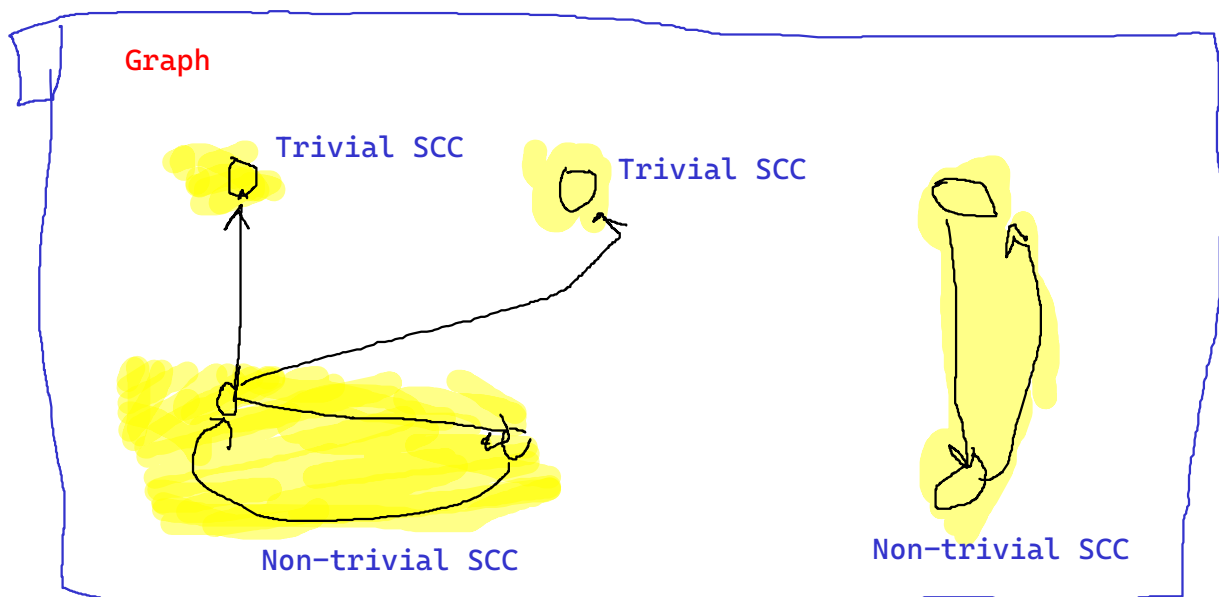
Strongly Connected Components

A directed graph is called strongly connected if there is a path in each direction between each pair of vertices of the graph. That is, a path exists from the first vertex in the pair to the second, and another path exists from the second vertex to the first.

In a directed graph G that may not itself be strongly connected, a pair of vertices u and v are said to be strongly connected to each other if there is a path in each direction between them.

A strongly connected component of a directed graph G is a subgraph that is strongly connected, and is maximal with this property: no additional edges or vertices from G can be included in the subgraph without breaking its property of being strongly connected. The collection of strongly connected components forms a partition of the set of vertices of G .

A strongly connected component C is called trivial when C consists of a single vertex which is not connected to itself with an edge, and non-trivial otherwise.



There are 2 main DFS based linear time algorithms associated with SCCs:

- Tarjan's Algorithm
- Kosaraju's Algorithm

Tarjan's Algorithm

$$O(V + E)$$

This algorithm gets all the strongly connected components from the Graph

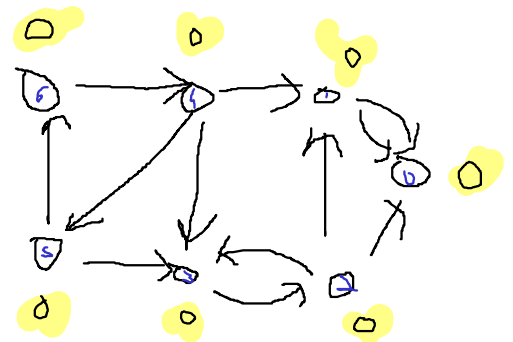
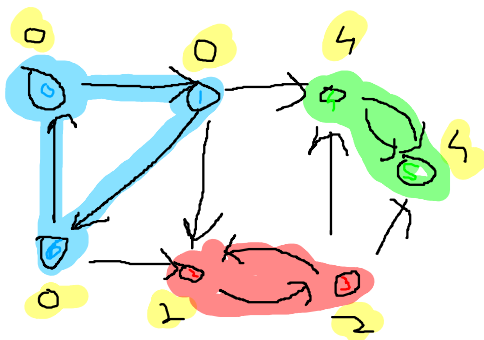
Input(s): Graph (V, E)

Output(s): List of List of vertices that are part of the same SCC

Each vertex of the graph appears in exactly one of the strongly connected components. Any vertex that is not on a directed cycle forms a strongly connected component all by itself: for example, a vertex whose in-degree or out-degree is 0, or any vertex of an acyclic graph.

One important thing to understand here is: The low-link value of a node is the smallest node ID reachable from that node when doing a depth-first search (DFS), including itself.

The problem with DFS and low link values is that if you change the way the graph is labelled it may produce a new lowlink score for each node that may not identify each SCC as DFS traversal is random. We use the Stack Invariants to fix this.



Tarjan's algorithm maintains a stack of valid nodes from which to update low-link values. Nodes are added to the stack of valid nodes as they are explored for the first time. Nodes are removed from the stack each time a complete SCC is found.

```

// Graph here uses adjacency list
void strong_connect(
    vector<vector<Edge>>& graph, // Edge is a struct with weight and to_node components
    ll v,
    stack<ll>& S,
    vector<bool>& on_stack,
    vector<ll>& indices,
    vector<ll>& lowlink,
    ll& index,
    vector<vector<ll>>& scc_list)
{
    indices[v] = index;
    lowlink[v] = index;
    index++;
    S.push(v);
    on_stack[v] = true;

    // all successors of v
    for (Edge e : graph[v]){
        if (indices[e.to_node]==-1){
            // successor has not been visited yet
            strong_connect(graph, e.to_node, S, on_stack, indices, lowlink, index, scc_list);
            lowlink[v] = min(lowlink[v], lowlink[e.to_node]);
        } else if (on_stack[e.to_node]){
            // Successor already on the stack and so is in the same SCC
            lowlink[v] = min(lowlink[v], indices[e.to_node]);
        }
    }

    // root of the scc graph - chosen to be the lowlink value
    if (lowlink[v] == indices[v]){
        vector<ll> scc;
        ll w;
        do {
            w = S.top();
            S.pop();
            on_stack[w] = false;
            scc.push_back(w);
        } while (w != v); // Stop when we reach the root of this SCC
        scc_list.push_back(scc);

        return; // terminate
    }
}

vector<vector<ll>> tarjans_algorithm(vector<vector<Edge>>& graph){
    vector<ll> indices(graph.size(), -1);
    vector<ll> lowlink(graph.size(), -1);
    vector<bool> onStack(graph.size(), false);

    ll index {0};
    stack<ll> S;
    vector<vector<ll>> scc_list;


    for (size_t v =0; v<graph.size(); v++){
        if (indices[v]==-1){
            strong_connect(graph, v, S, onStack, indices, lowlink, index, scc_list);
        }
    }
    return scc_list;
}

```

Kosaraju's Algorithm

The idea behind Kosaraju's algorithm is you traverse the graph twice.

```
void Visit(vector<vector<Edge>>& graph, ll v, stack<ll>& L, vector<bool>& visited){
    if (!visited[v]){
        visited[v] = true;
        for (Edge& e : graph[v]){
            Visit(graph, e.to_node, L, visited);
        }
        L.push(v);
    }
}
```



```
void Assign(
    vector<vector<Edge>>& graph,
    ll u,
    ll root,
    vector<ll>& assigned_component_root,
    vector<vector<ll>>& components)
{
    if (assigned_component_root[u] == -1){
        assigned_component_root[u] = root;
        components[root].push_back(u);
        // get in- neighbours using a edge traversal  $O(E)$  here
        vector<ll> in_neighbours;
        for (size_t v = 0; v < graph.size(); v++){
            for (Edge& e : graph[v]){
                if (e.to_node == u){
                    in_neighbours.push_back(v);
                }
            }
        }
        for (ll& in_neighbour : in_neighbours){
            Assign(graph, in_neighbour, root, assigned_component_root, components);
        }
    }
}
```

```
vector<vector<ll>> kosaraju(vector<vector<Edge>>& graph ){
    vector<bool> visited (graph.size(), false);
    vector<ll> assigned_component (graph.size(), -1);
    stack<ll> L;
    for (size_t v = 0; v < graph.size(); v++){
        Visit(graph, v, L, visited);
    }

    vector<vector<ll>> components(graph.size());
    vector<vector<ll>> actual;

    while (!L.empty()){
        ll u = L.top(); L.pop();
        Assign(graph, u, u, assigned_component, components);
    }

    for (vector<ll>& vec : components ){
        if (!vec.empty()){
            actual.push_back(vec);
        }
    }
    return actual;
}
```