

Depth-First Search

Mike Mirzayanov

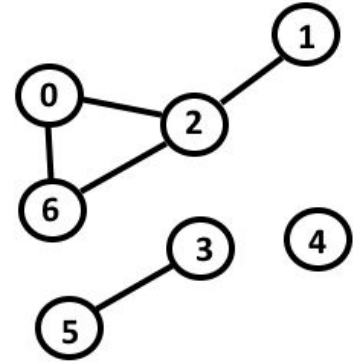
Undirected Graphs

An **undirected graph** is an ordered pair $G = (V, E)$, where V is a set of vertices (nodes) and E is a set of 2-elements subsets of V called edges.

Usually, $0 < |V| < \infty$. Edges are denoted as (u, v) . Multiple edges and loops are not allowed.

On the picture:

- $V = \{0, 1, 2, 3, 4, 5, 6\}$, 7 vertices in total
- $E = \{(0,2), (0,6), (2,1), (2,6), (3,5)\}$, 5 edges in total
- Vertex degree is number of adjacent vertices. For example, $\deg(0)=2$, $\deg(1)=1$, $\deg(2)=2$, ..., $\deg(4)=0$.
- Path is a sequence of vertices, connected consistently, $(2,6,0,2,1)$ is a path.

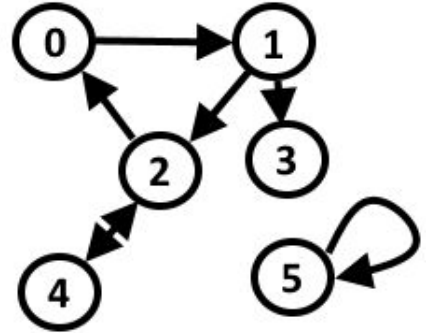


Directed Graphs

A **directed graph** is an ordered pair $G = (V, E)$, where V is a set of vertices (nodes) and E is a set of pairs over V called edges (arcs).

Usually, $0 < |V| < \infty$. Edges are denoted as (u, v) . Multiple edges are not allowed, but loops are allowed.

On the picture:



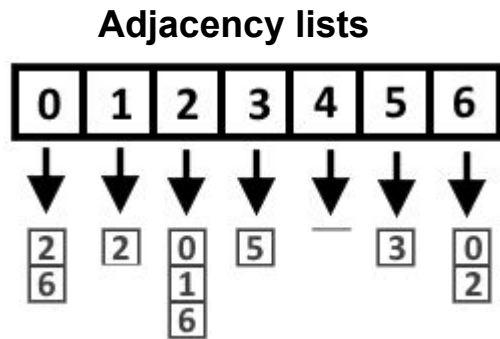
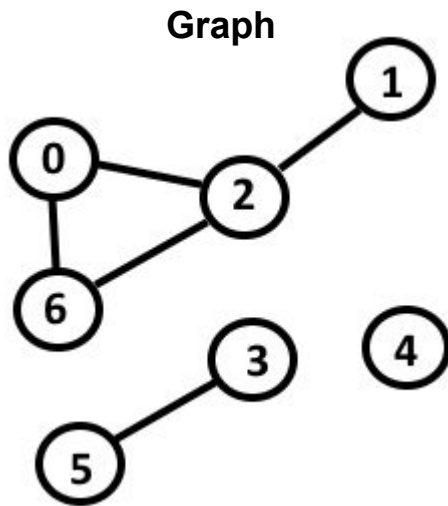
- $V = \{0, 1, 2, 3, 4, 5\}$, 6 vertices in total
- $E = \{(0,1), (1,2), (1,3), (2,0), (2,4), (4,2), (5,5)\}$, 7 edges in total
- Vertex out-degree is number of outgoing edges. For example, $out_deg(0)=1$, $out_deg(1)=2$, $out_deg(3)=0$, ..., $out_deg(5)=1$. Similarly, $in_deg(0)=in_deg(1)=1$
- Path is a sequence of vertices, connected consistently, $(4,2,0,1,2)$ is a path.

Store graphs in a program

The simplest and efficient way is **adjacency lists**. Maintain array (vector) g , where $g[u]$ is neighbours of vertex u .

C++:

```
vector<vector<int>> g;  
...  
cin >> n >> m;  
g.resize(n);  
for (int i = 0; i < m; i++) {  
    cin >> x >> y;  
    g[x].push_back(y);  
    g[y].push_back(x);  
}
```



Path Search Problem

Given a graph G (directed or undirected) and two vertices s and t . Check if a path from s to t exists? If a path exists, find and print any path from s to t .

We need some strategy to traverse the graph visiting all the vertices reachable from s . Think about it as about exploring a maze.

Two popular strategies (algorithms) DFS and BFS:

- DFS: explore maze by a single person moving each time to unvisited place or moving back if no neighbouring unvisited place.
- BFS: explore maze as fire spreads

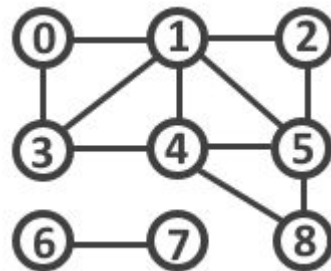
Path Tree

Path tree is an outgoing tree rooted at s . Path tree covers all vertices reachable from s .

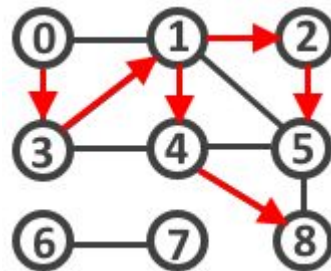
- $parent[u] = u$ for root vertex $u=s$
- $parent[u] = -1$ for unreachable vertices u
- $parent[u] = v$ if v is the previous vertex for u on the path from s to u

```
function pathTo(t):  
    path := []  
    while parent[t] != t:  
        path.push(t)  
        t = parent[t]  
    path.push(t)  
    reverse(path)  
    return path
```

Graph example, $s=0$.



Path tree example, $s=0$.



Parent array:

$parent = [0, 3, 1, 0, 1, 2, -1, -1, 4]$

DFS explanation

DFS starts from at the root vertex s . Say, u is a current vertex of DFS. If there is not visited neighbour v , go to v in recursive manner. If there is no such neighbour, go back returning from recursive call.

Simplest implementation (works in $O(|V|+|E|)$):

```
function DFS( $u$ ):  
     $visited[u] := true$   
    for each  $v$ ,  $v$  is neighbour of  $u$ :  
        if not  $visited[v]$ :  
            DFS( $v$ )
```

If graph is not connected (not all vertices are reachable from s), use serie of DFS calls, like:



```
 $visited[] \leftarrow false$   
for each  $u$ ,  $u$  is a vertex:  
    if not  $visited[u]$ :  
        DFS( $u$ )
```

DFS explanation

Colors:

- * **WHITE**: vertex is not visited yet (not started)
- * **GRAY**: vertex is visited but not finished (started but not finished)
- * **BLACK**: vertex is finished

Times:

- * $tin[u]$: time vertex u started to be processed (**WHITE**  **GRAY**)
- * $tout[u]$: time vertex u finished to be processed (**GRAY**  **BLACK**)

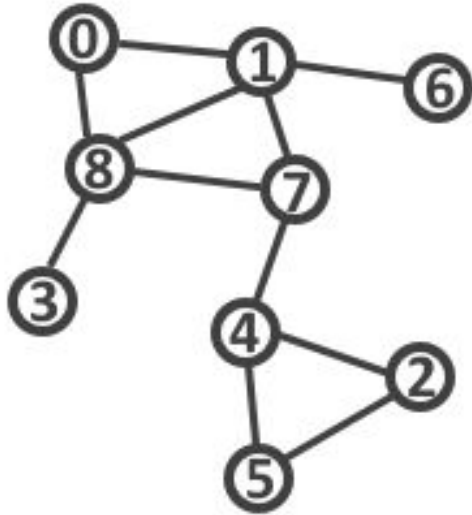
```
function DFS( $u, p$ ):  
     $parent[u] := p$   
     $color[u] := GRAY$   
     $tin[u] := T++$   
    for each  $v$ ,  $v$  is neighbour of  $u$ :  
        if  $color[v] == WHITE$ :  
            DFS( $v, u$ )  
     $color[u] := BLACK$   
     $tout[u] := T++$ 
```

main:

```
 $parent[] \leftarrow -1$   
 $color[] \leftarrow WHITE$   
DFS( $s, s$ )
```

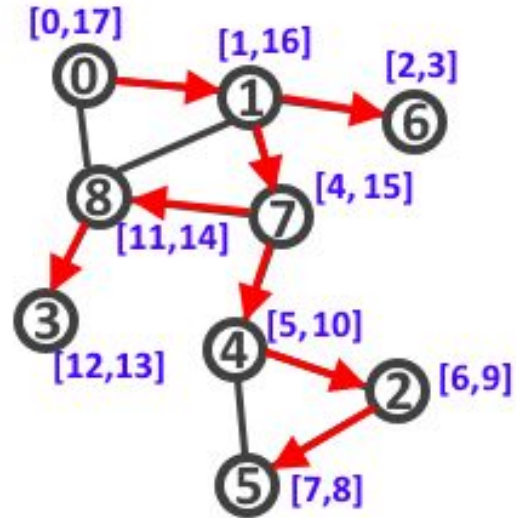
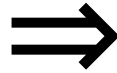
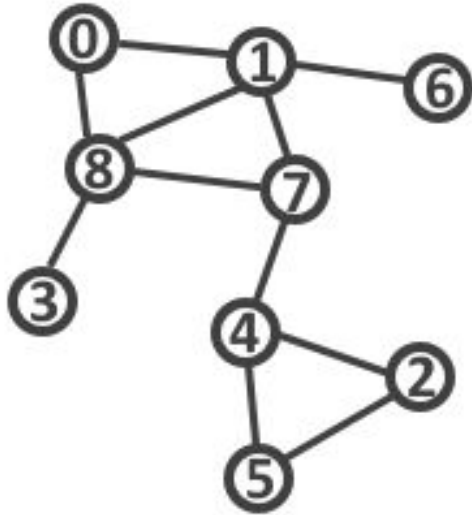

DFS explanation

Example ($s=0$):



DFS explanation

Example ($s=0$):



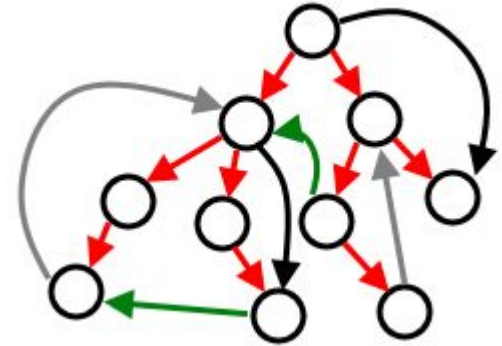
Time Segments

For any two vertices u and v time segments $[tin[u], tout[u]]$, $[tin[v], tout[v]]$:

- $[tin[u], tout[u]]$ nested in $[tin[v], tout[v]]$ - meaning u is descendant of v in DFS-tree,
- $[tin[v], tout[v]]$ nested in $[tin[u], tout[u]]$ - meaning v is descendant of u in DFS-tree,
- $[tin[v], tout[v]]$ and $[tin[u], tout[u]]$ do not intersect - meaning u and v are not comparable.

Edge Types

- \rightarrow Tree edges: from GRAY to WHITE
- \rightarrow Backward edges: from GRAY to GRAY
- \rightarrow Forward edges: from GRAY to BLACK
- \rightarrow Cross edges: from GRAY to BLACK



* Look on times t_{in}/t_{out} to choose forward or cross

For an undirected graph type of an edge is assigned in moment of the first lookup along the edge.

For undirected graphs forward and cross edges do not exist, only tree and backward edges exist.

Cycle Criterion

Cycle exists if and only if a backward edge exists, i.e. lookup from the current vertex to a GRAY vertex (careful with undirected graphs: skip an edge by which you came).

For undirected graphs:

- visited/not visited information is enough

For directed graphs:

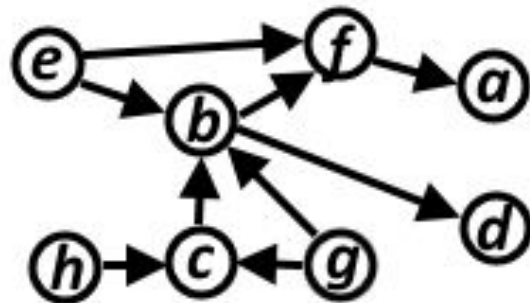
- three colors (WHITE, GRAY, BLACK) are required

Topological Sort

A **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex v , u comes before v in the ordering.

Possible topological sorts for graph from the picture:

- [e, h, g, c, b, f, a, d]
- [g, h, c, e, b, f, d, a]
- and others



Criterion: A topological sort exists if and only if the graph is an acyclic.

Topological Sort

Naive approach. Each time extract a vertex u with $out_deg(u)=0$

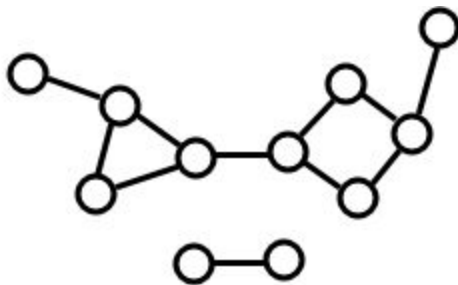
Linear algorithm. Sort vertices in order of decreasing of $tout$ values.

```
function topsort( $u$ ):  
     $visited[u] := true$   
    for each  $v$ ,  $v$  is neighbour of  $u$ :  
        if not  $visited[v]$ :  
            topsort( $v$ )  
     $order.push(u)$ 
```

```
main:  
     $order := [], visited[] \leftarrow false$   
    for each  $u$ ,  $u$  is a vertex:  
        if not  $visited[u]$ :  
            topsort( $u$ )  
    reverse( $order$ )
```

Bridges

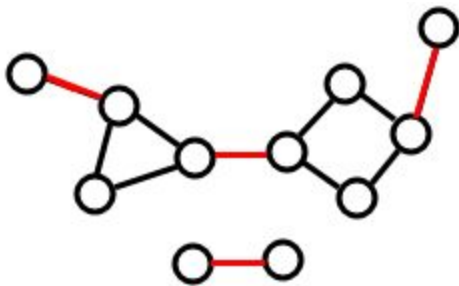
Given an undirected graph, a **bridge** is an edge of the graph whose deletion increases its number of connected components.



Criterion. An edge is a bridges if and only if it does not belong to any simple cycle.

Bridges

Given an undirected graph, a **bridge** is an edge of the graph whose deletion increases its number of connected components.



Bridges are marked with red

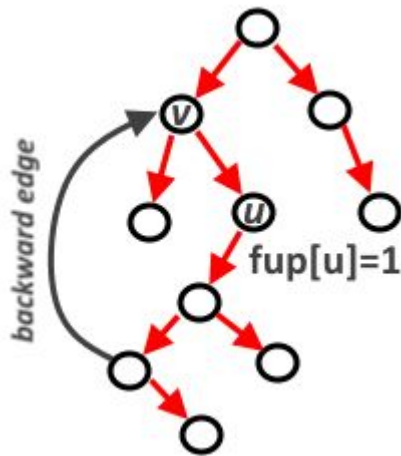
Criterion. An edge is a bridges if and only if it does not belong to any simple cycle.

Bridges

After DFS only tree edges could be bridges.

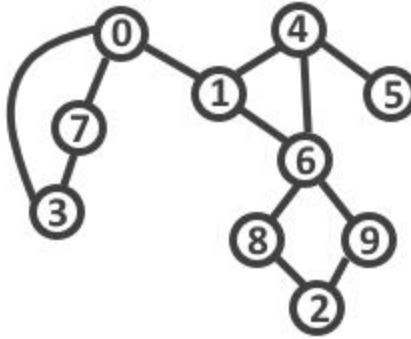
How to check if tree edge is a bridge?

Definition. Forward-up value, $fup[u]$ is minimal possible distance from the root to v in DFS-tree where v is reachable from u by zero or more tree edges and after it zero or one backward edge.



Bridges

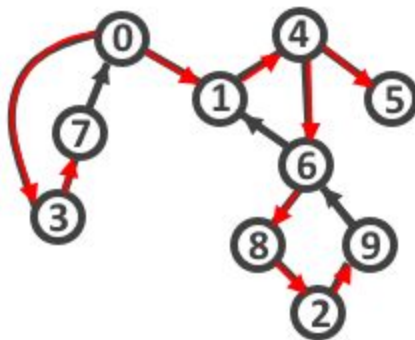
Example



Consider DFS visits neighbours in order of increasing their indices.

Bridges

Example



Calculate values fup :

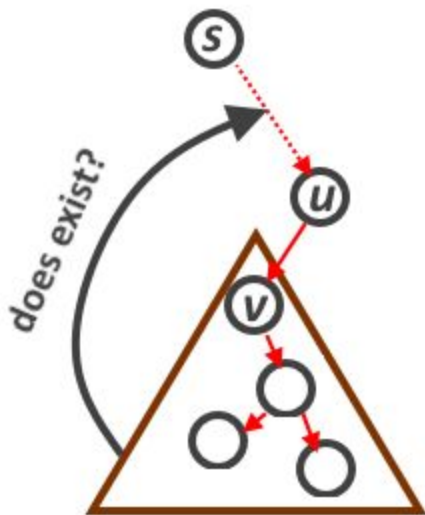
- $fup = [0, 1, 3, 0, 1, 3, 1, 0, 3, 3]$.

Bridges

Calculate *fup* values in $O(|V|+|E|)$:

```
function DFS(u, p, d):  
    visited[u] := true  
    dep[u] := fup[u] := d  
    for each v, v is neighbour of u:  
        if v == p:  
            continue  
        if not visited[v]:  
            DFS(v, u, d+1)  
            fup[u] := min(fup[u], fup[v])  
        else:  
            fup[u] := min(fup[u], dep[v])
```

Bridges



Tree edge (u, v) is a bridge if and only if:

- there is no edge from subtree of v to u or upper.

Or:

Tree edge (u, v) is a bridge if and only if:

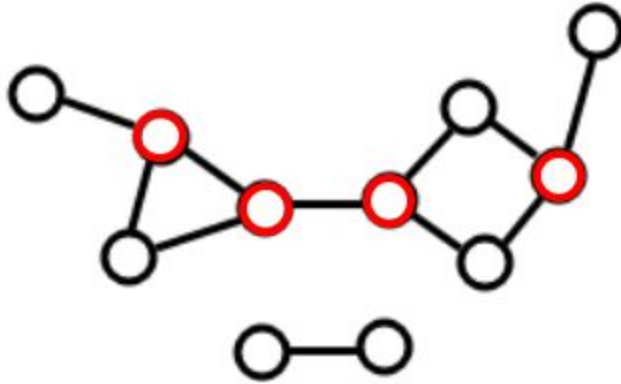
- $fup[v] > dep[u]$.

It means that to find all edges, just add one line into the code to calculate fup -values (after $DFS(v, u, d+1)$).

Conclusion. Easy linear algorithm to find all bridges exists.

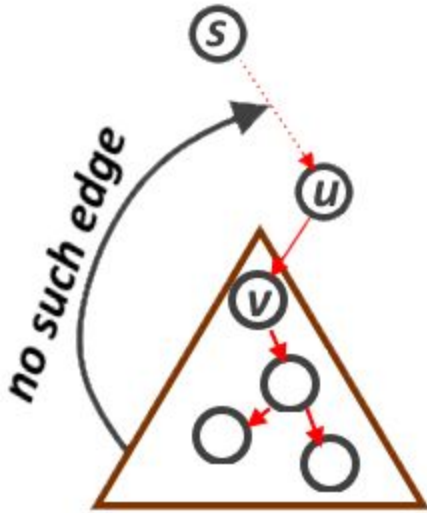
Cut Vertices (Articulation Points)

Given an undirected graph, a **cut vertex** is a vertex of the graph whose deletion increases its number of connected components.



There is no simple connection between bridges and cut points!

Cut Vertices (Articulation Points)



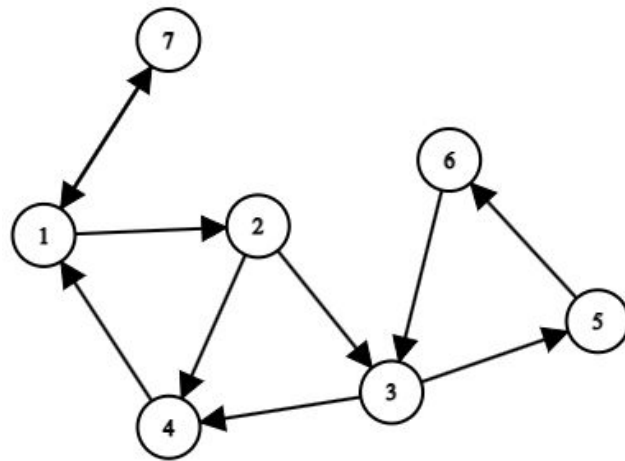
A vertex u is a cut point if and only if:

- u is not a root and such tree edge (u, v) exists that $fup[v] \geq dep[u]$,
- u is root and number of children in DFS-tree is at least 2.

Conclusion. Easy linear algorithm to find all cut vertices exists.

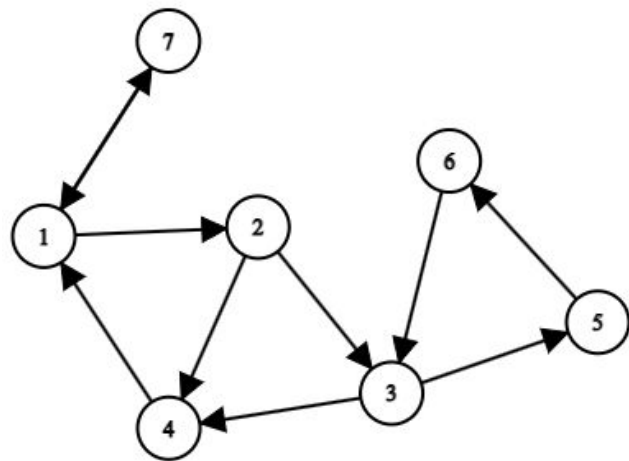
Strongly connected components

- A **directed** graph is strongly connected if every vertex is reachable from every other vertex.
- How to check if the given directed graph is strongly connected in linear time?



Strongly connected components

- A **directed** graph is strongly connected if every vertex is reachable from every other vertex.
- How to check if the given directed graph is strongly connected in linear time?
- Run DFS from any vertex u and check that all vertices are reachable from u
- Reverse the graph and again in reversed graph run DFS from any vertex u and check that all vertices are reachable from u

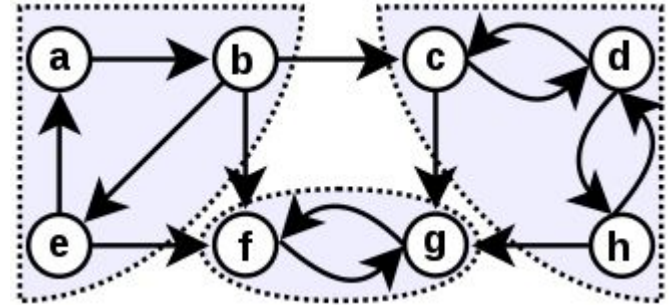


Strongly connected components

The strongly connected components of an arbitrary directed graph are maximal subgraphs that are themselves strongly connected.

Such partition on strongly connected subgraphs is unique.

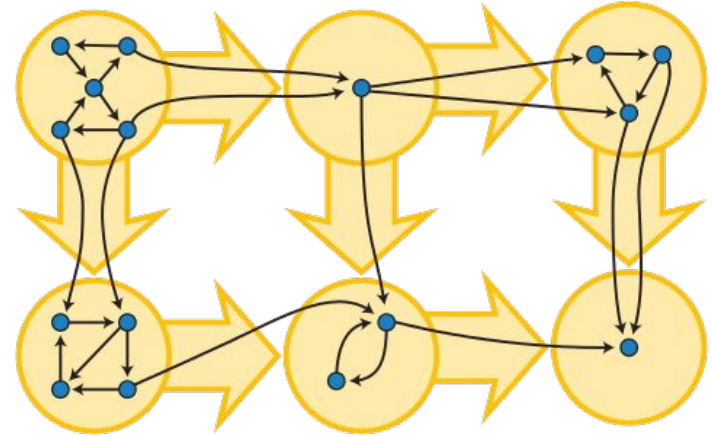
There are several linear algorithms to find strongly connected components.



Strongly connected components

If each strongly connected component is contracted to a single vertex, the resulting graph is a directed acyclic graph, the condensation of G .

A directed graph is acyclic if and only if it has no strongly connected subgraphs with more than one vertex, because a directed cycle is strongly connected and every non-trivial strongly connected component contains at least one directed cycle.



Kosaraju's algorithm

1. Run serie of DFS to visit all vertices. Order vertices in decreasing time of their tout values. Pay attention, it's exactly how topsort works!
2. Reverse the graph.
3. Do new serie of DFS in reversed graph in order of item 1.
4. Each DFS tree is a single strongly connected component.

Strongly connected component will be found in order of top sort in the correspondent condensation.

Thank you for your attention

Questions?