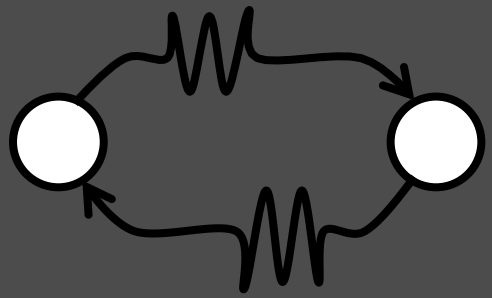
Advanced Computer Contest Preparation

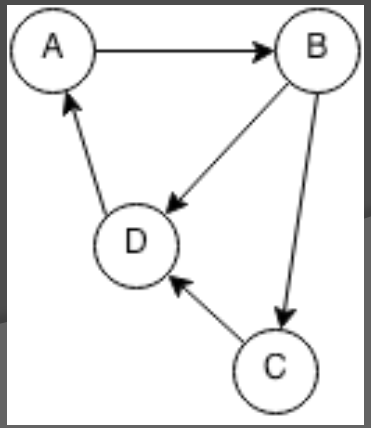
Lecture 27

TARJAN'S ALGORITHMS

Strongly Connected Components

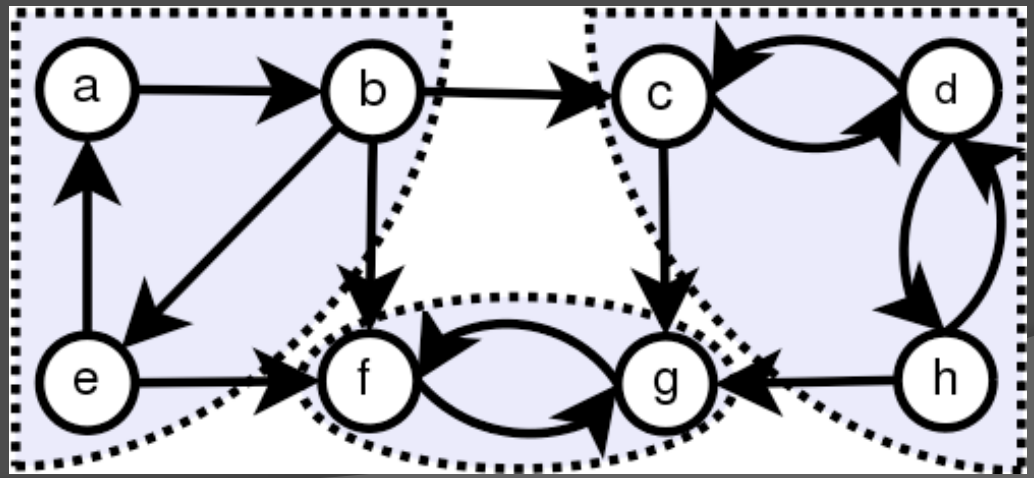
- In a directed graph, two nodes are *strongly connected* if there is a path from one node to the other and back
- A set of nodes is called a *strongly connected component* (SCC) if each pair of nodes in this subset is strongly connected





SCC Algorithm

- Given a directed graph, how can we find all of the SCCs?
- Note that we want to know the largest possible SCCs of the graph

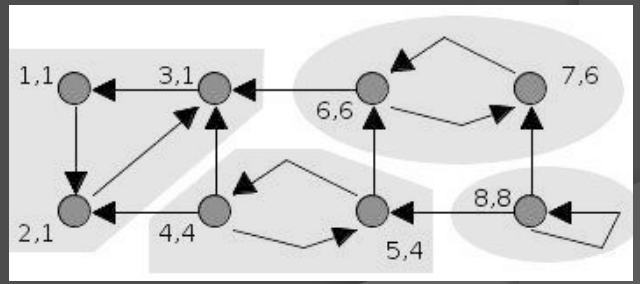


Tarjan's SCC Algorithm

- Tarjan's SCC algorithm finds all of the SCCs of a graph in a single DFS
- A stack is also used that keeps track of visited nodes that are not currently a part of an SCC
  - Any nodes that are not in the stack are either unvisited or part of an SCC

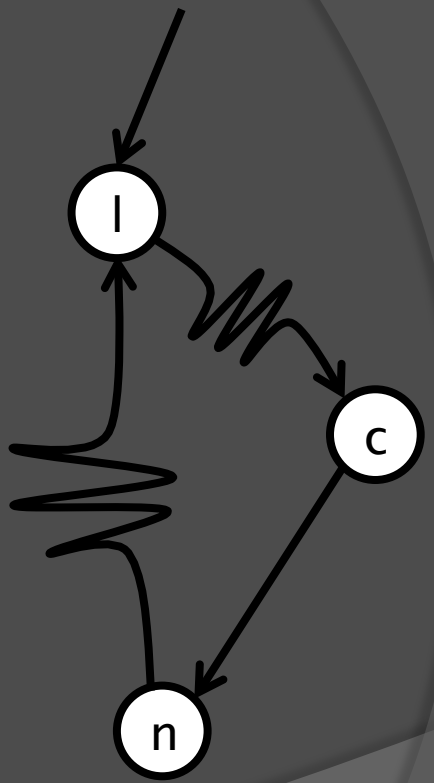
Tarjan's SCC Algorithm

- Each node has a number assigned to it representing the order of visiting (Bruce calls this **`DFN`**)
- Each node also needs to know an earlier visited node that it is strongly connected to (Bruce calls this **`low`** node)
  - Initially, the **`low`** node is equal to the node itself



Tarjan's SCC Algorithm

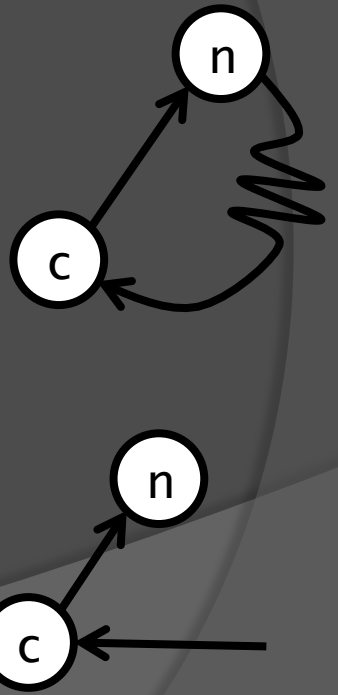
- Push the current node into the stack
- For each unvisted neighboring node:
  - Run Tarjan's on that node (remember, Tarjan's is a DFS!)
  - Check if that node's `low` node is smaller than the current node's `low` node
  - If it is, it means that the current node has a connection to a node that was visited even earlier



Current node (c)
Neighbor node (n)
Neighbor low node (l)

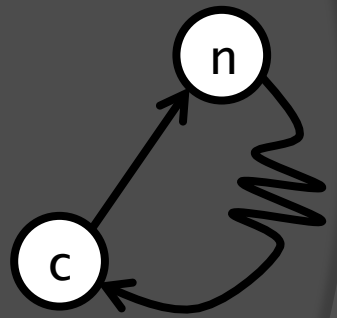
Tarjan's SCC Algorithm

- For each neighboring visited node:
  - The only possible way to have an already visited neighboring node is if:
    - The neighboring node has a path to the current node OR
    - The neighboring node does not have a path to the current node, meaning that we had to start the DFS again from some other node



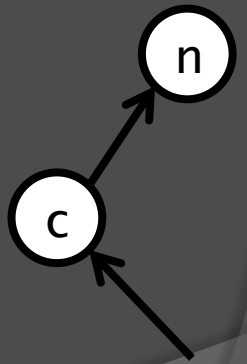
Tarjan's SCC Algorithm

- Case 1: Neighboring visited node has connection to the current node
  - Obviously, these two nodes are strongly connected
  - Therefore, node's `low` *could* be that neighboring node (use that node's `DFN`)
  - Note that the neighboring node is still in the stack



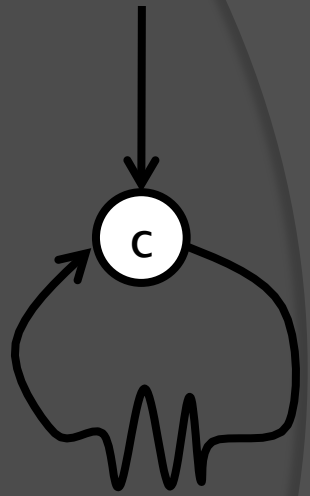
Tarjan's SCC Algorithm

- Case 2: The DFS had to be started again on another node
  - These nodes cannot be strongly connected
  - We should not try to change our node's `low`
  - Because the DFS on the neighboring node's subgraph is completed, it is already part of an SCC and thus no longer in the stack



Tarjan's SCC Algorithm

- After iterating through all neighbors, check if `low = DFN`
- If `low = DFN`
  - This means that this node does not have a connection to a node above
  - It is part of its own SCC in relation to nodes visited before
  - Pop nodes in the stack until after we pop the current node – all popped nodes are part of the same SCC

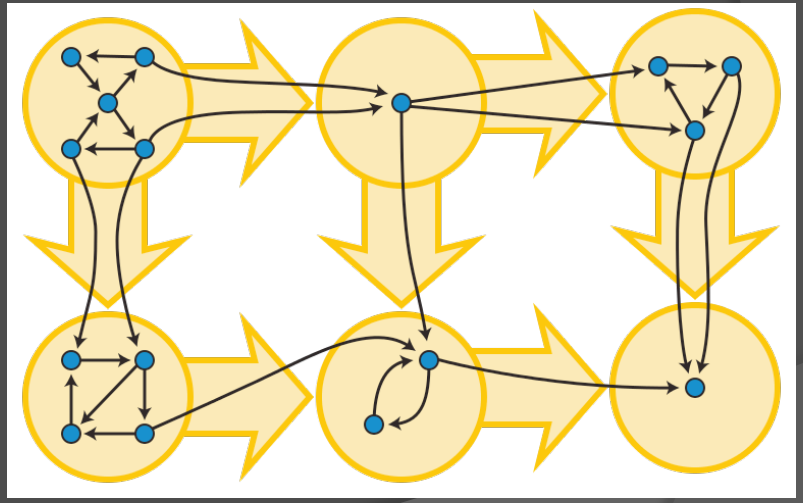


Tarjan's SCC Algorithm - Pseudocode

```
DFN[], low[], vis[]
stack S
tarjan(nd)
        DFN[nd] ← low[nd] ← ++idx
        vis[nd] ← true
        S.push(u)
        for each neighboring node u
                if not vis[u]
                        tarjan(u)
                        low[nd] ← min(low[nd],low[u])
                else if u in S
                        low[nd] ← min(low[nd],DFN[u])
        if DFN[nd] = low[nd]
                while true
                        v ← S.top
                        S.pop
                        if v = nd break
```

Generating the New Graph

- After finding all SCCs, we can form a new DAG where each SCC is a node
  - Why is the new graph a DAG?
  - Any cycles would be part of an SCC, and therefore, would already be part of a node

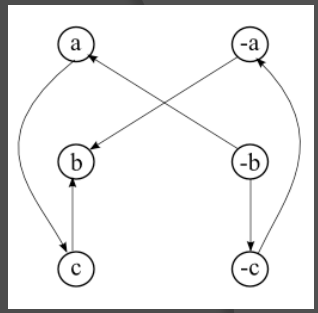


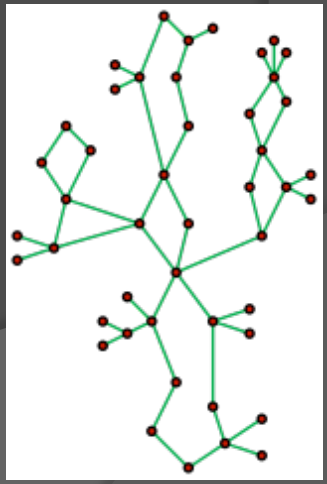
Generating the New Graph

- Simple DFS
- For each unvisited neighboring node:
  - Run the DFS on the neighboring node
  - If the neighboring node is not in the same SCC
    - Form an edge from the current node's SCC to the neighboring node's SCC
    - This edge might already exist in the new graph

Applications

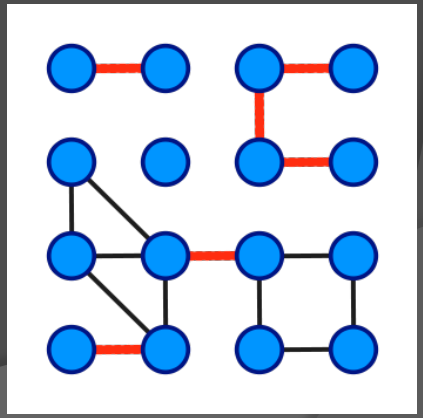
- Transforming/simplifying a directed graph into a DAG so that DP can be performed
- 2-satisfiability problem
- Finding *bridges* and *articulation points*
- Determining if a graph is a cactus graph
  - Every edge is in at most one simple cycle





Bridges

- A *bridge* is an edge where when removed, one or more nodes lose connection to a previously connected node
  - The number of components of the graph increases
- Applies mainly to undirected graphs



Red edges are bridges

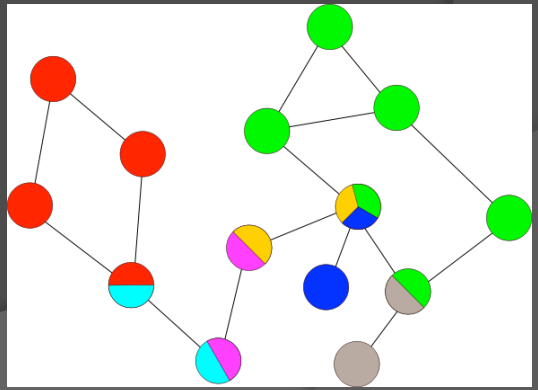
Finding Bridges

- Method is similar to Tarjan's SCC algorithm algorithm
  - We use a DFS, `low`, and `DFN`, but not a stack
- An edge `(u,v)`, where `u` is the parent, is a bridge if `low[v] > DFN[u]`
  - There is no other path from `v` to `u`
  - There is no path from `v` to a node visited earlier than `u`

Articulation Points

- An articulation point is a node where when removed, one or more nodes lose connection to a previously connected node (excluding the articulation point)
  - The number of components in the graph increases
- Typically applies to undirected graphs

Nodes with > 1 color are articulation points



Finding Articulation Points

- Method is similar to bridge finding algorithm
- For a non-starting node **u**, it is an articulation point if for one of its children **v**, **low[v]** ≥ **DFN[u]**
  - **v** does not have a path to a node visited earlier than the **u**
  - **v** can have more than one path to **u**
- For a starting node, it is an articulation point if it calls DFS on more than one child

Interesting Fact

- The versions of SCC, articulation point finding, and bridge finding were all initially described by Tarjan
- i.e. each of these algorithms can be called "Tarjan's Algorithm"

THANK YOU!