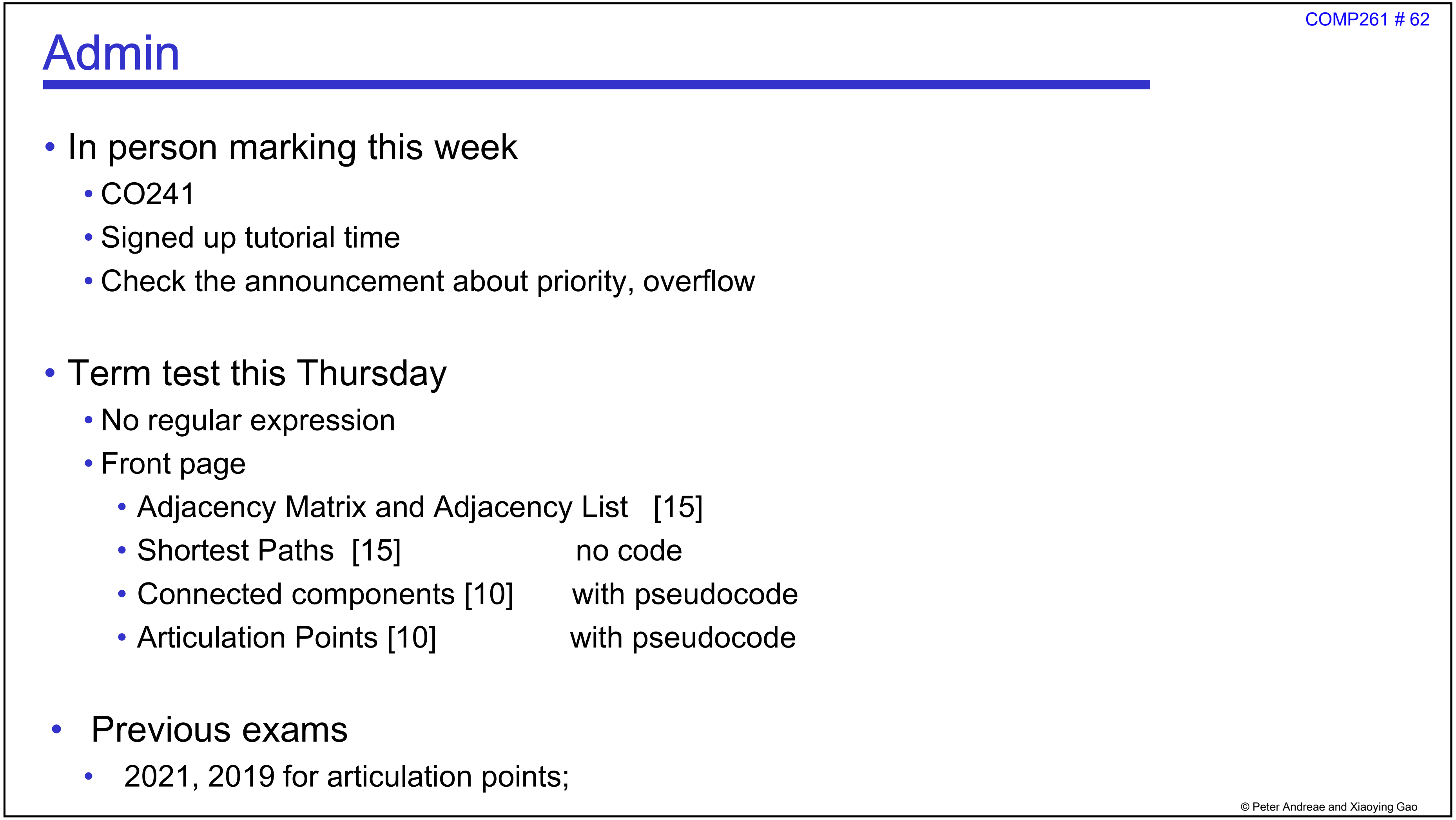
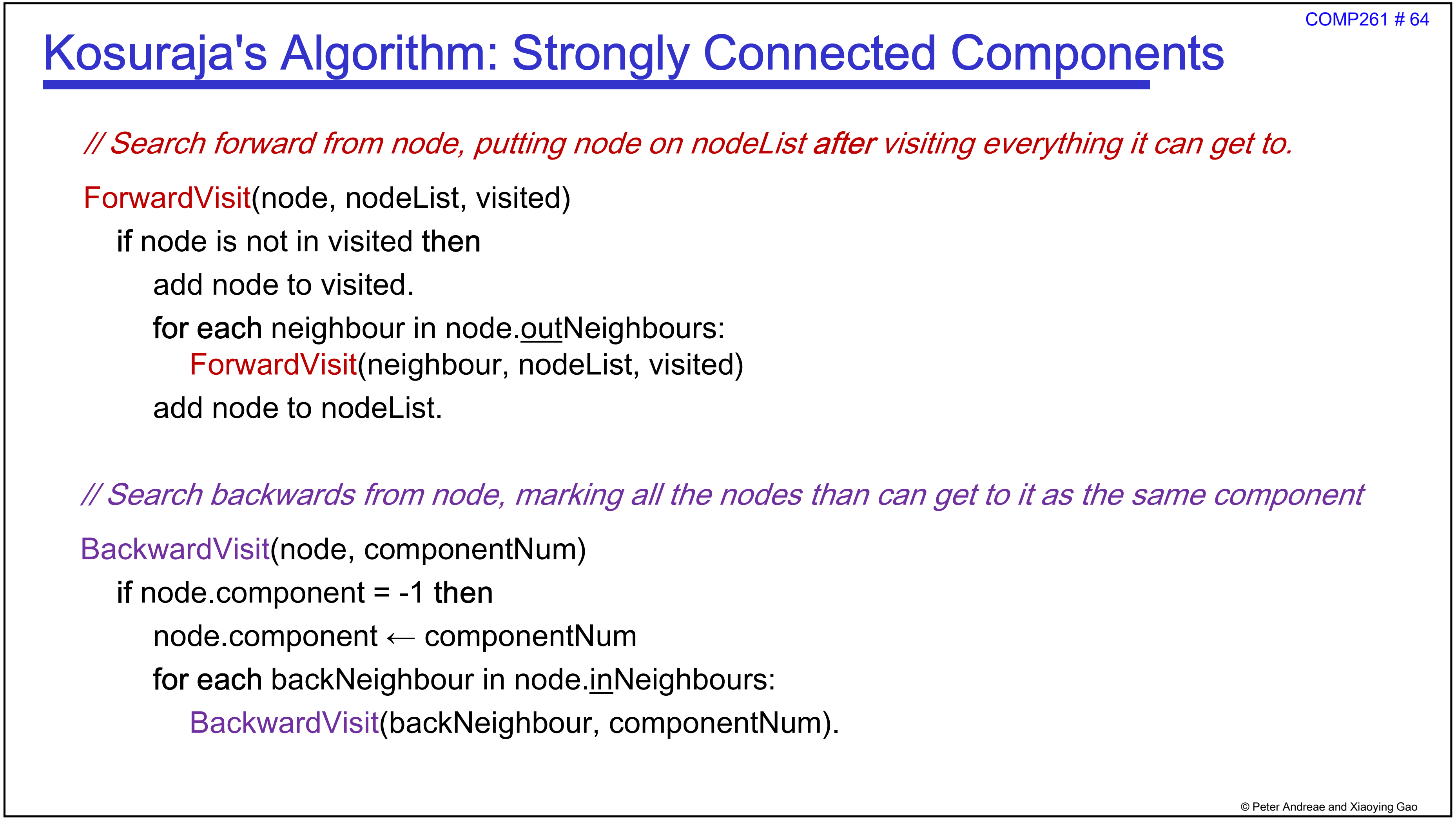


Admin

- In person marking this week
  - CO241
  - Signed up tutorial time
  - Check the announcement about priority, overflow
- Term test this Thursday
  - No regular expression
  - Front page
    - Adjacency Matrix and Adjacency List [15]
    - Shortest Paths [15] no code
    - Connected components [10] with pseudocode
    - Articulation Points [10] with pseudocode
- Previous exams
  - 2021, 2019 for articulation points;

Kosuraja's Algorithm: Strongly Connected Components

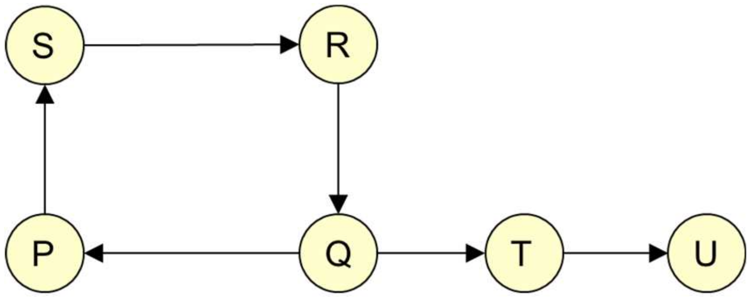
```
Kosuraja(graph):
    for each node in graph:
         node.component ← -1                            // initialize nodes to not be in a component
    componentNum ← 0
    nodeList ← empty list;
    visited ← empty set

    for each node in graph:
        if node is not visited then
            ForwardVisit(node, nodeList, visited)       // traverse graph from node forward along edges,
                                                        // adding nodes to nodeList in post-order

    for each node in nodeList in reverse order:
        if node.component = -1 then
            BackwardVisit(node, componentNum)           // traverse graph from node backward along edges
            componentNum++                              // marking nodes with the component number
```

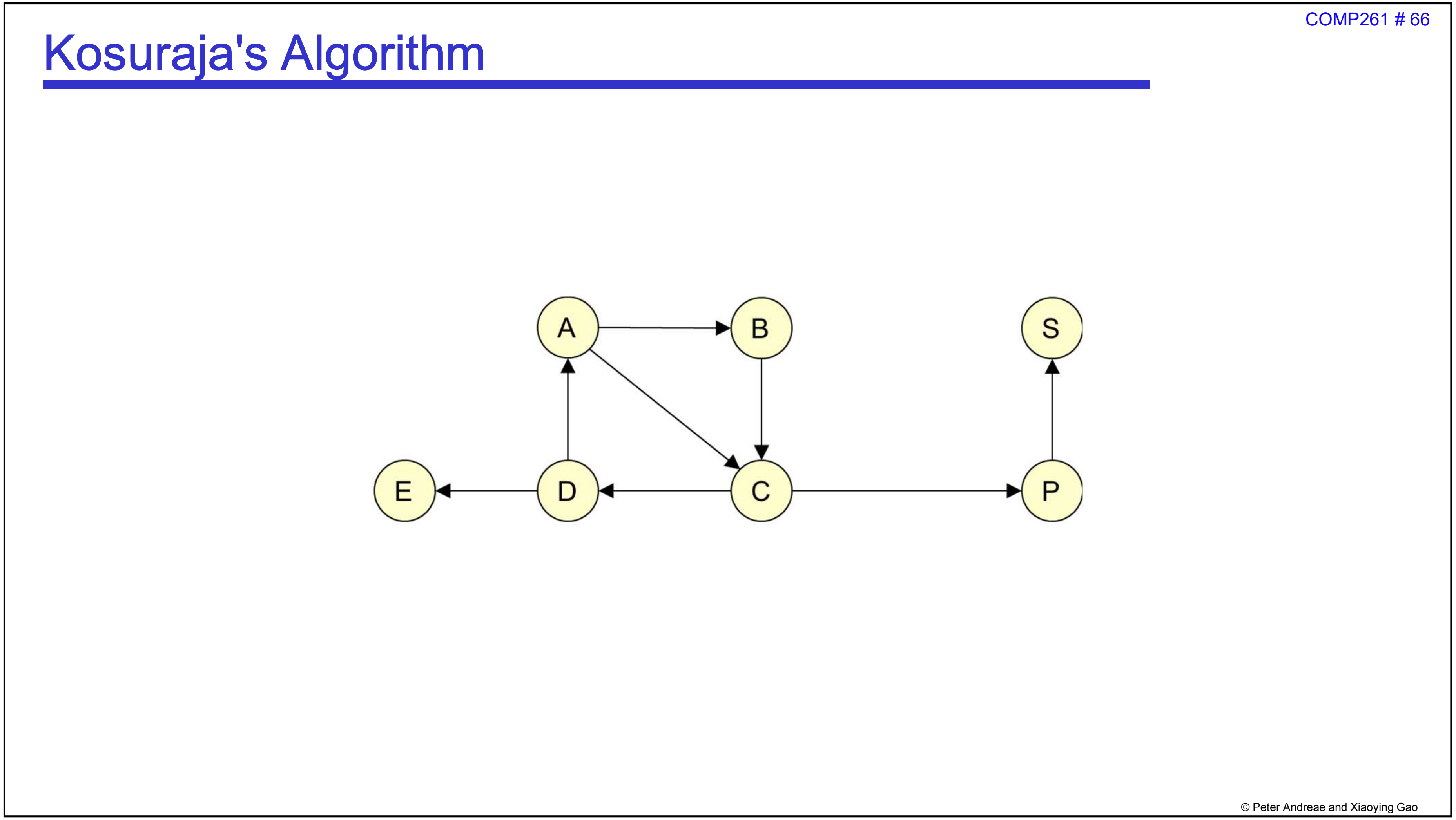


Kosuraja's Algorithm: Strongly Connected Components

```
// Search forward from node, putting node on nodeList after visiting everything it can get to.

ForwardVisit(node, nodeList, visited)
   if node is not in visited then
      add node to visited.
      for each neighbour in node.outNeighbours:
         ForwardVisit(neighbour, nodeList, visited)
      add node to nodeList.

// Search backwards from node, marking all the nodes than can get to it as the same component

BackwardVisit(node, componentNum)
   if node.component = -1 then
      node.component ← componentNum
      for each backNeighbour in node.inNeighbours:
         BackwardVisit(backNeighbour, componentNum).
```

Kosuraja's Algorithm

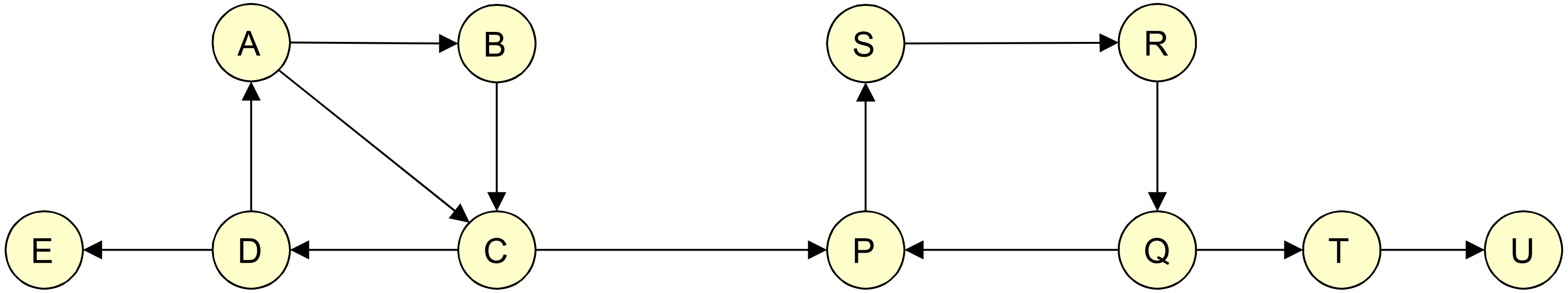




COMP261 # 66

Kosuraja's Algorithm

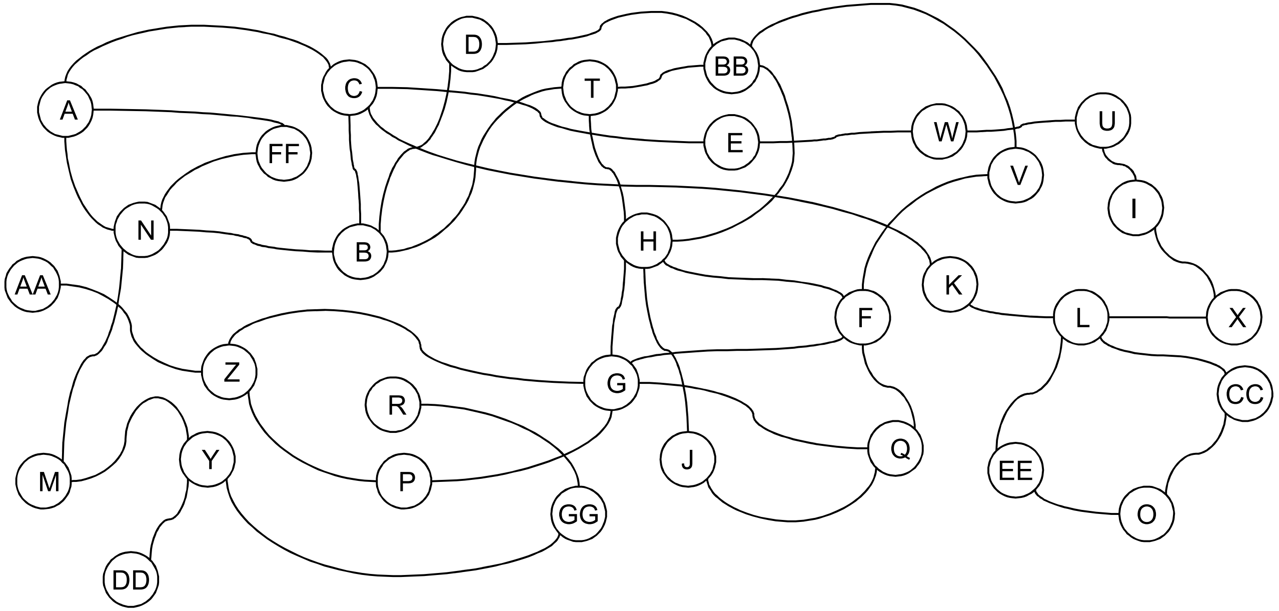
Kosuraja's Algorithm



NodeList:

Articulation points

- This graph is connected, but is it “fragile”?
  Would deleting one node disconnect it?



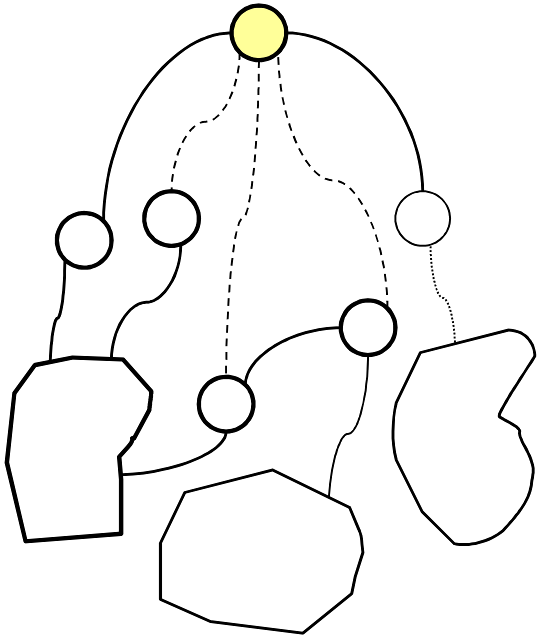
- Articulation point: node whose removal would disconnect part of the graph.

  (for undirected graphs - articulation points in directed graphs are a bit more complex)

Articulation Points: a bad algorithm

```
ArticulationPoints(graph):
    aPoints ← empty set,
    for each node  in graph
        visited  ← empty set
        add node  to visited
        Traverse(first neighbour of node, visited)
        for each  neighbour  of node
            if  neighbour is not visited  then
                add node to aPoints
    return aPoints

Traverse (node, visited ):
    if  node not in visited  then
        add node to visited
        for each neighbour of node
            Traverse(neighbour, visited )
```

Take each node out in turn,
and test for connectedness



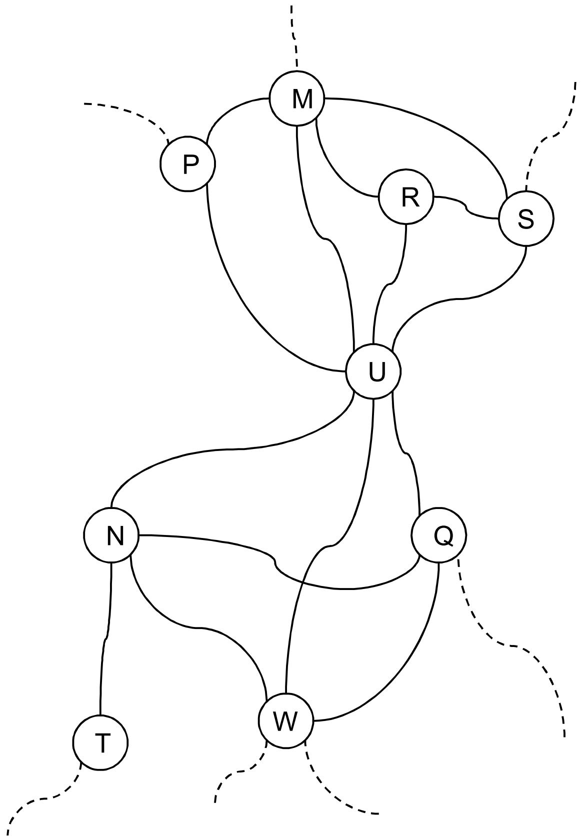
Why is it bad?

- Cost of Traverse: O(E) = $O(N^2)$ for very dense graphs
- Cost of Algorithm: O(NE) = $O(N^3)$ for very dense graphs
- Why do we have to traverse the whole graph n times, once for each node?
- Why not do a single traversal, identifying all articulation points as we go?

Articulation Points.

- What are we looking for?

Nodes in a graph that separate the graph into two groups, so that all paths from nodes in one group to nodes in the other group go through the node.



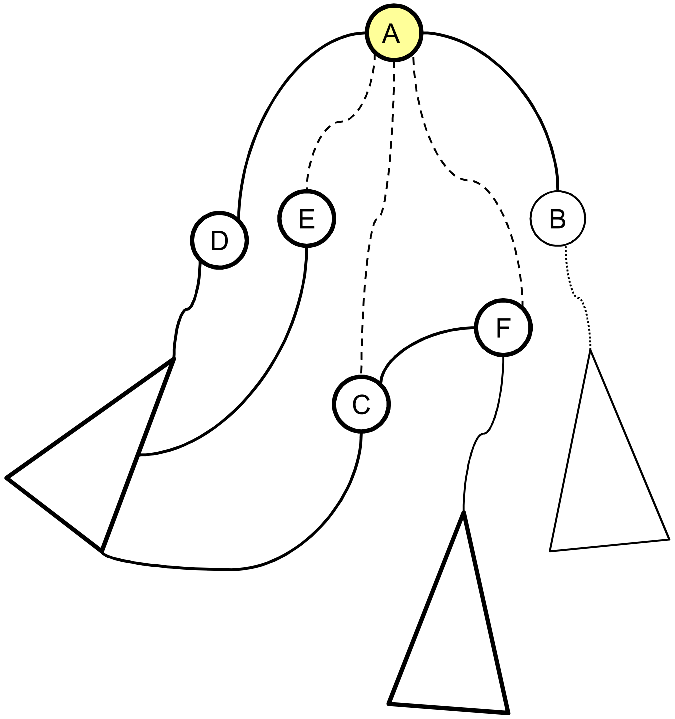
Articulation points: DFS

- Use depth first search, keeping track of the depth of each node in the search tree

- At root:

  if there is >1 edge to an unvisited node,
  then root is an articulation point.

At Root node



Articulation points: DFS

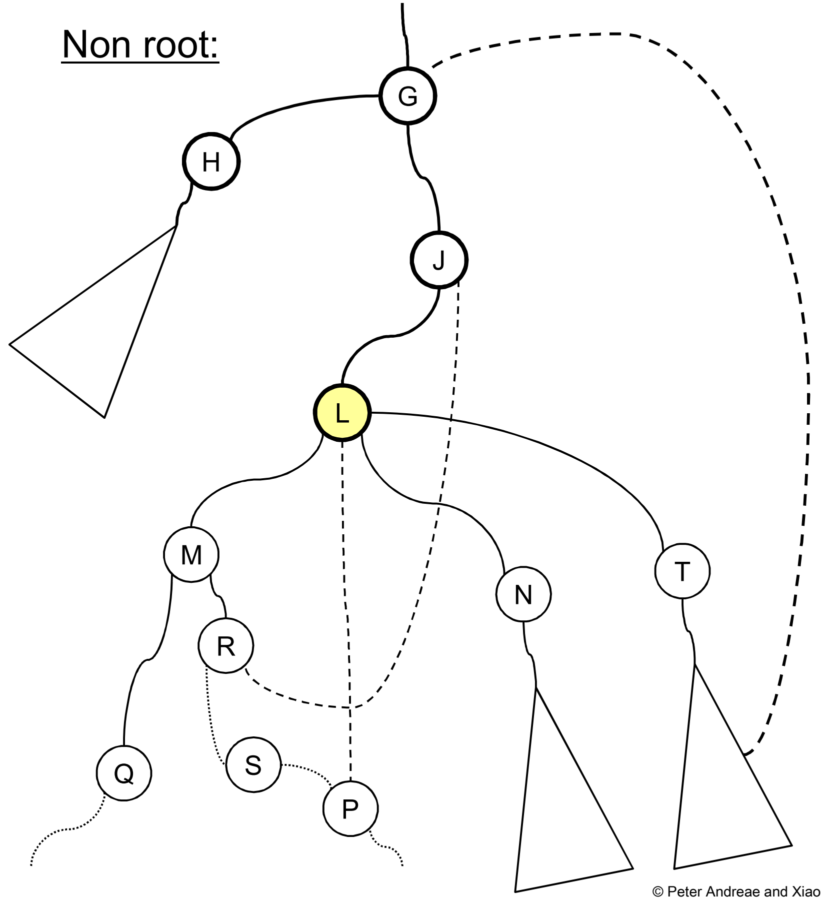
- Use depth first search, keeping track of the depth of each node in the search tree

- At root:

  if there is >1 edge to an unvisited node,
  then root is an articulation point.

- At lower nodes:

  If there is a subtree that has no edge up to an ancestor node
  then node is an articulation point.



Articulation Points

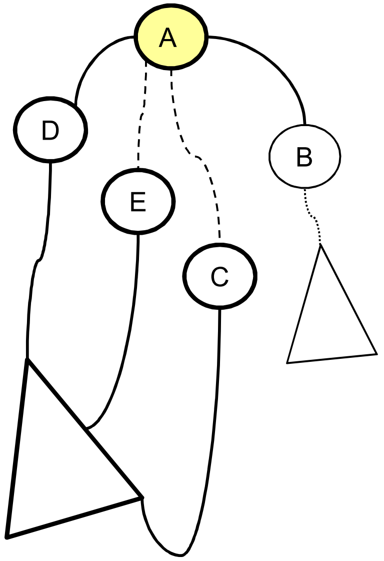
- Key ideas of algorithm:
    - Record depth of nodes as you search
    - From each recursive search of a subtree, return the highest point (ie, minimum depth) that the subtree can "reach back" to.
    - Compare the "reach back" of each subtree to depth of this node

      = depth of node node is an articulation point
    - Can use depth to record whether visited

Articulation points: Pseudo-code

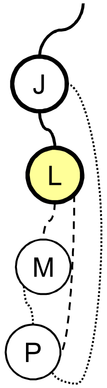
Can store depth of nodes in the nodes or in a Map<Node, int>

```
ArticulationPoints(graph):
   foreach node in graph:
        node.depth ← -1,
   aPoints ← { }          // the set of articulation points to return
   numSubtrees ← 0
   start ← first node in graph.
   start.depth ← 0,                         // visit start
   foreach neighbour of start
        if neighbour.depth = -1 then        // not visited yet
            recArtPts( neighbour, 1, start, aPoints)
            numSubtrees ++
   if numSubtrees > 1  then
        add start to aPoints
   return aPoints
```

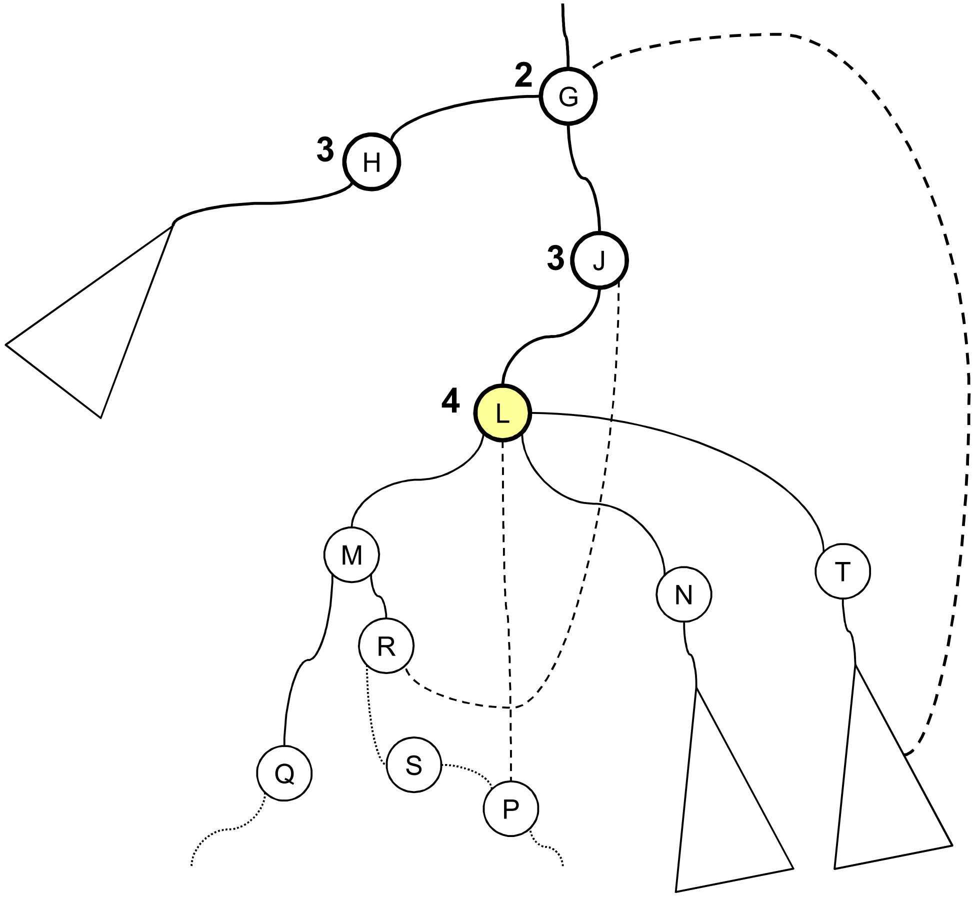


Articulation points: DFS

```
recArtPts(node, depth, fromNode, aPoints):
   node.depth ← depth,                  // visit node
   reachBack ← depth,                   // how far up this node can reach
   foreach  neighbour  of node:
      if  neighbour = fromNode  then  continue
      else if  neighbour.depth ≠ -1 then         // already visited
         reachBack ← min(neighbour.depth, reachBack)
      else
         childReach ← recArtPts(neighbour, depth +1, node, aPoints)
         if childReach >= depth  then            // subtree doesn't reach past this node.
            add node to aPoints
         reachBack ← min(childReach, reachBack )
   return  reachBack
```



Articulation points: DFS



Exercise

- 2021, graphs, (c)

- 2019 exam
- Q2: calculate depth and reachback for each node
- Identify the articulation points