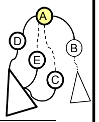
COMP261 Lecture 9

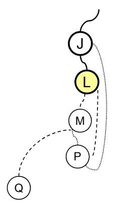
Articulation Points 2 of 2 (Algorithm)

Articulation points: DFS

```
Initialise:   for each node: node.depth ← ∞, articulationPoints ← { }
start.depth ← 0,  numSubtrees  ← 0
for each neighbour  of start
      if neighbour.depth = ∞ then
            recArtPts( neighbour, 1, start)
            numSubtrees ++
if numSubtrees > 1   then add start to articulationPoints
```



```
recArtPts(node, depth, fromNode):
      node.depth ← depth,    reachBack ← depth,
      for each  neighbour  of node other than fromNode
            if  neighbour.depth <∞ then
                  reachBack = min(neighbour.depth, reachBack)
            else
                  childReach = recArtPts(neighbour, depth +1, node)
                  if childReach  >= depth  then
                              add node to articulationPoints
                  reachBack = min(childReach, reachBack )
      return  reachBack
```

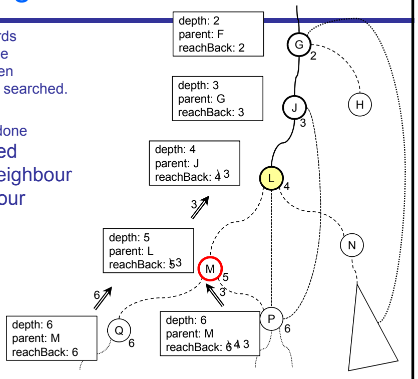


Recursive algorithm

The activation stack records the local variables, and the iterator through the children while the subtree is being searched.

Processing on a node is done
- when first reached
- between each neighbour
- after last neighbour

The simple iterative DFS only processes a node when first reached.



Removing the recursion

- How do we do the articulation points algorithm iteratively?

```
iterArtPts(firstNode, depth, fromNode):
    push firstNode on fringe
    while (fringe not empty)
        node ← pop from fringe
        node.depth ← depth,    reachBack ← depth,
        for each  neighbour  of node other than fromNode
            if  neighbour.depth < ∞ then
                reachBack = min(neighbour.depth, reachBack)
            else
              childReach = recDFS(neighbour, depth +1, node)
              reachBack = min(childReach, reachBack )
              if childReach  >= depth  then
                  add node to articulationPoints

        return  reachBack
```

We need more information on the fringe

Removing the recursion

- How do we do the articulation points algorithm iteratively?

```
iterArtPts(firstNode , depth, fromNode):
    push  ⟨firstNode, depth, fromNode⟩ onto fringe
    while (fringe not empty)
        ⟨node, depth, fromNode⟩ ← pop from fringe
        node.depth ← depth,    reachBack ← depth,
        for each  neighbour  of node other than fromNode
            if  neighbour.depth <∞ then
                reachBack = min(neighbour.depth, reachBack)
            else
                push  ⟨neighbour, depth+1, node⟩ onto fringe
                childReach = recDFS(neighbour, depth +1, node)
                reachBack = min(childReach, reachBack )
                if childReach  >= depth  then
                    add node to articulationPoint s
        return  reachBack
```

Put tuple of node, depth, parent on fringe

Recursive: complete DFS first neighbour
Iterative: just puts neighbour on fringe.
How do we get and process the result?

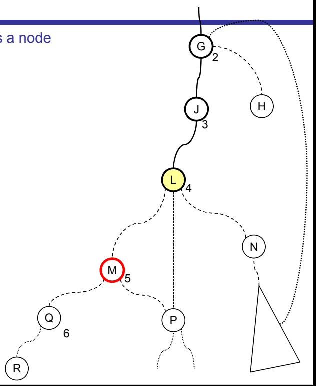
Iterative algorithm

The simple iterative DFS only processes a node when first reached.

⟨G, 2, F⟩
⟨H, 3, G⟩
⟨J, 3, G⟩
⟨L, 4, J⟩
⟨N, 5, L⟩
⟨P, 5, L⟩
⟨M, 5, L⟩
⟨P, 6, M ⟩
⟨Q, 6, M⟩
⟨R, 7, Q⟩

How do we now update the reach of M?
How do we mark M as an artic. point?

Need a more sophisticated stack!



Iterative

- In general, need to make an explicit stack corresponding to the activation stack:
  - Each stack element must have fields corresponding to parameters and local variables of the recursive method
  - Stack element must have equivalent of "program counter" to keep track of progress
  - Do not remove stack element from stack until all children have been processed
- Pattern:

```
while  stack is not empty
    peek at element on top of stack
    perform next action of element (possibly adding new element)
        storing values in fields of element.
    if element processing is complete   then  remove element from stack.
```

Articulation Points with stack

Stack elements contain:

node:	graph node to be processed
reach:	local variable to store current reach back level

as well as:

parent:	stack element we came from: (a) to not revisit its graph node, (b) to update its reach
depth:	that the node will have, if visited via this stack element
children:	queue of unvisited neighbours to be processed in turn

When peek at a stack element:

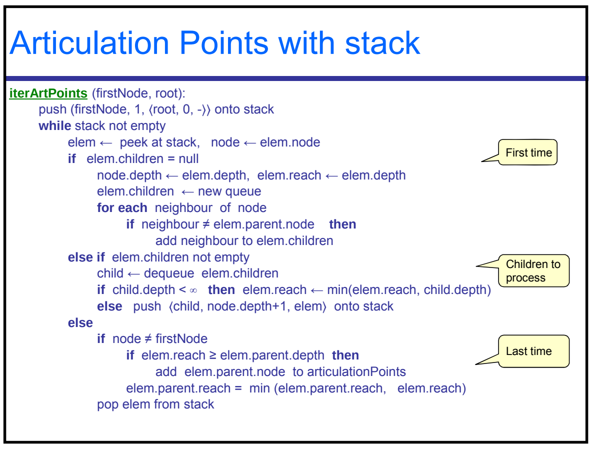
first time:	initialise and construct children
children to process:	poll child; if visited, update; else push on fringe
last time:	determine if parent is articulation point update parent's reach

Articulation Points with Stack

- Still have to deal with the start node specially:

```
Initialise:   for each node: node.depth ← ∞, articulationPoints ← { }

start.depth ← 0,  numSubtrees  ← 0
for each neighbour  of start
    if neighbour.depth = ∞ then
        iterArtPts( neighbour,  start)
        numSubtrees ++
if numSubtrees > 1   then add start to articulationPoints
```



Articulation Points with stack

```
iterArtPoints (firstNode, root):
    push (firstNode, 1, ⟨root, 0, -⟩) onto stack
    while stack not empty
        elem ← peek at stack,   node ← elem.node
        if   elem.children = null
            node.depth ← elem.depth,  elem.reach ← elem.depth
            elem.children  ← new queue
            for each  neighbour  of  node
                if  neighbour ≠ elem.parent.node    then
                    add neighbour to elem.children
        else if  elem.children not empty
            child ← dequeue  elem.children
            if  child.depth < ∞   then  elem.reach ← min(elem.reach, child.depth)
            else   push ⟨child, node.depth+1, elem⟩  onto stack
        else
            if  node ≠ firstNode
                if  elem.reach ≥ elem.parent.depth  then
                    add  elem.parent.node  to articulationPoints
                elem.parent.reach =  min (elem.parent.reach,   elem.reach)
            pop elem from stack
```