6 DFS inspired Articulation Point Detection

Graduate Students Only

> *Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.*

(This needs to be fine-tuned and Bi-connectivity needs to be added)

Given a connected simple graph, then an articulation point[5] is a node s such that, if it were to be removed, the graph breaks apart in two or more disconnected components. A good motivation to find articulation points is to detect vulnerabilities in any connection network: if the node is removed (or disabled, such as 'down for repairs') then there are at least two other remaining nodes in the network that are no longer connected. Thus, s is an articulation point if there are at least two other nodes, u_1 and u_2 such that s is on every path that connects these two nodes. Detecting articulation points is a challenge, certainly if you want to exhaustively check for all possible paths for all pairs of nodes. An articulation point is a structural property of the graph, and thus independent of any labeling or any traversals. Yet, the DFS can be used to detect whether or not a node s is (or is not) an articulation point by taking advantage of the DFS-numbering scheme. Although it is true that different choices of starting nodes, and different order of visiting neighboring nodes will lead to different spanning trees, they all have this in common: There is a root, there are nodes with both a parent and children, and there are leaves. So suppose we have completed a DFS of the graph, and suppose we have numbered the nodes in the order of discovery. Just like we did this before when we used CFL to track the Pre- and $Post$-Labels. Except now we do not need the Post-Labels, and we rename the Pre-Labels to Num-labels. Also, during the DFS we labeled the edges as either a tree-edge or a back-edge (we assume the graph is undirected, so there are no cross-edges). So we could look for articulation points in the graph after the DFS has been completed. A node in a graph is one of three types of nodes in the DFS-tree.

1. A node in the graph that is the root of the DFS-tree is an articulation point if-and-only-if it has two or more children in the DFS-tree. (i.e. if there two or more tree-edges adjacent to the node). The algorithm simply counts the tree-children for all nodes, even though this count is only needed for the root.
2. A node in the graph that is a leaf in the DFS-tree is never an articulation point. (Why?)
3. A node in the graph that is an internal node in the DFS-tree is adjacent to at least two other nodes in the graph. Let s be the name of such an internal node, and let $Num(s)$ its rank according to the order that nodes are discovered during the DFS. One of its adjacent nodes in the graph is the parent-node in the DFS-tree, which has a Num-label strictly less than $Num(s)$. Let $B(s)$ be set of all the nodes that have been discovered before s and have a Num-value strictly less than $Num[s]$. Furthermore, let s have k children in the DFS-tree, which we will call for now $c_s(1), c_s(2) \ \ldots \ c_s(k)$. Each child has its own descendants in the tree and for each child define $C(i)$ as the child $c_s(i)$ together with its descendants in the tree. We now have the set of nodes partitioned into $k+2$ sets: The nodes s itself, $B(s)$, i.e. the set of all the nodes that have been discovered *before* node s was discovered (and have a Num-label strictly less than $Num(s)$), and $C(i)$ for each of the k children.

   $\odot$ Now let us discuss the possibility of node s to be an articulation node. Remember, s is an articulation point if there are at least two other nodes in the graph, u_1 and u_2 such that s is on every path in the graph that connects these two nodes. So take u_1 as any node in $B(s)$ and let u_2 be in one of the $C(i)$'s. The node s is *not* an articulation node if *all* of its descendants (the union of the $C_s(i)$'s), are connected with a node in $B(s)$ using a back-edge. On the other hand, the node s *is* an articulation node if *there is at least one* of its descendants that *is not* are connected with a node in $B(s)$ using a back-edge.

   $\odot$ So for every child node $c(i)$, track the nodes that can be cycled back to by taking one or more tree-edges (forward/downward), followed by a back-edge, if any. If there are more nodes that can be cycled back to, then track the node with the lowest value of Num. By following such a back-edge, you reach a node that has already been discovered and does have a discovery number Num already. To track this, introduce a new label $Low[c]$ to associate lowest Num-number for the nodes that can be reached in such a manner. So if node s has a child c, and $Low(c) \geq Num(s)$, then the node s is an articulation node because all paths from the parent of s to the child of s go through s. On the other hand, if $Low(c') < Num(s)$ for all children c', then node s is not an articulation node, because for all children there is an alternate path back to $B(s)$, either through the children c themselves, or one of their descendants. Now consider how the Low-labels are initially created and updated:

   - This $Low[s]$ is initialized as $Low[s] \leftarrow Num[s]$ whenever node s is discovered,
   - This $Low[s]$ is updated in a post-order fashion with information gathered from each w that is adjacent to s:

$$\begin{cases} Low[s] \leftarrow \min\{Low[s], Num[w]\} & \text{if } (s,w) \text{ is a back edge} \\ Low[s] \leftarrow \min\{Low[s], Low[w]\} & \text{if } (s,w) \text{ is a (spanning) tree edge} \end{cases} \tag{2}$$

4. Note that the value of $Low(s)$ is not needed for determining wether or not node s itself is an articulation point, but the value $Low(s)$ is needed to determine whether the parent of s (or earlier ancestor) is an articulation point.

[5] Sometimes called cut point or separation point

5. Observe that, for every node x value for $Low[x]$ is initially set equal to $Num[x]$, and this value might get smaller if a descendent can be used to cycle back to a node that was discovered earlier. Thus at all times, $Low[x] \leq Num[x]$. The value for $Low[x]$ will have reached its final value when all adjacent nodes of this x have been explored and the color of x switches to BLACK.
6. So a node s is an articulation point if it has one or more tree-children that can not be used to cycle back to the set $B(s)$, even though it may have other children that can be used to cycle back to an ancestor.
7. If a node s is an articulation node, it is still possible for this node to have other children that *can* be used to cycle back.
8. The sets $B(s)$, and $C_s(1)\ \ldots\ C_s(k)$ were only introduced to give convincing arguments why it is correct. These need not be implemented. We dó need $Predecessor[\cdot]$ and $Children[\cdot]$ labels.

ArticulationPointsFinder, the algorithm

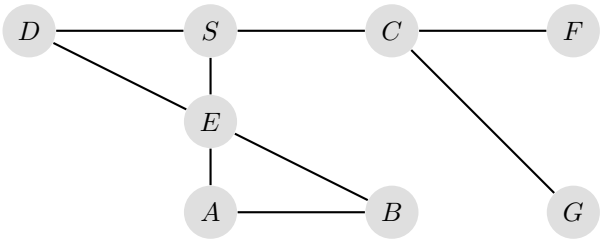
```
LINE 100:  Algorithm ArticulationPointsFinder(graph MyGraph)
LINE 110:  int CFL ← 0,                                          /* global variable to Count First & Last
LINE 120:  bool ArtNode[v] ← {FALSE};                            /* A global Array; a node is NOT an ArtPoint by default.
LINE 130:  int Num[v] ← {0};                                     /* A global Array, indicating the rank of the node in discovery process
LINE 140:  int Low[v] ← {0};                                     /* A global Array, see above
LINE 150:  int CameFrom[v] ← NILL;                               /* A global Array, pointing to the parent of a node in the DFS-ST
LINE 160:  int Children[v] ← {0};                                /* A global Array, counting the number of children per node.

LINE 170:  begin driver ArticulationPointsFinder
LINE 180:  start ← pick any node; (for this course: take S)      /* To get the Articulation Point Finder started
LINE 190:  FindArtPointsHelper(MyGraph, start)
LINE 200:                       //All articulation points have been identified, except perhaps for the root, which is next:
LINE 210:  if Children[start] > 1) then ArtNode[start] ← TRUE end-if
LINE 220:                       //All articulation points have now been identified.
LINE 230:  end-driver
LINE 240:  end algorithm ArticulationPointsFinder
```

where

```
LINE 300:  Algorithm FindArtPointsHelper(graph MyGraph, node start)
LINE 310:  //Precondition: Color[start] == WHITE //
LINE 320:  Color[start] ← GRAY
LINE 330:  Num[start] ← ++CFL
LINE 340:  Low[start] ← Num[start]
LINE 350:  ArtNode[start] ← {FALSE}

LINE 300:  for all w adjacent to start and selected alphanumerically do
LINE 310:      if Color[w] == WHITE
LINE 320:          then do                                       /* A new child w of start has been discovered
LINE 330:              CameFrom[w] ← start                       /* and the edge (start, w) is a new tree edge
LINE 340:              Children[start]++
LINE 350:              FindArtPointsHelper(MyGraph, w)

LINE 360:              if (Num[start] ≤ Low[w]) then ArtNode[start] ← TRUE end if

LINE 370:              Low[start] ← min {Low[start], Low[w]}
LINE 380:              end then                                  /* Done processing a newly discovered child w of start

LINE 390:          else-if (CameFrom[start] ≠ w)                 /* w was already discovered, and (start, w) is indeed a back-edge
LINE 400:              then Low[start] ← min {Low[start], Num[w]} end if

LINE 410:      end if

LINE 420:  end-for all
LINE 430:  Color[start] ← BLACK
LINE 440:  end-algorithm FindArtPointsHelper
```

6.1 Suggested Exercises

SE.42: You are asked to write the resulting values for the labels as indicated for the graph as given, or for the graph that you yourself should create. Make sure it has a cycles as well as some 'dangling' legs.



SE.NotYetNumbered Exercise
The correct statement in the algorithm is
LINE 400: **then** $Low[start] \leftarrow \min\{Low[start],\ Num[w]\}$ **end if**

But you note that $Low[x] \leq Num[x]$ for all nodes x, so it is tempting to adjust the statement to
LINE 3400: **then** $Low[start] \leftarrow \min\{Low[start],\ Low[w]\}$ **end if**

Take an example (a graph with 10 or so nodes) and show the Low numbers when computed both ways. Try to force the two algorithms to conclude with two different conclusions by taking children in a different order.

SE.NotYetNumbered Exercise
The color GRAY has not been used at all. In LINE 400, we only test for new adjacent nodes to be WHITE. If the color of w is GRAY, then w is a direct ancestor of $start$. This is not too useful in the current application, but keep it in mind for future applications.