CMSC 451: Lecture 11
Articulation Points and Biconnected Components
Tuesday, Oct 6, 1998

Read: This material is not covered in the text (except as Problem 23–2).

Articulation Points and Biconnected Graphs: Today we discuss another application of DFS, this time to a problem on undirected graphs. Let $G = (V, E)$ be a **connected** undirected graph. Consider the following definitions.

Articulation Point (or Cut Vertex): Is any vertex whose removal (together with the removal of any incident edges) results in a disconnected graph.

Bridge: Is an edge whose removal results in a disconnected graph.

Biconnected: A graph is *biconnected* if it contains no articulation points. (In general a graph is k-connected, if k vertices must be removed to disconnect the graph.)

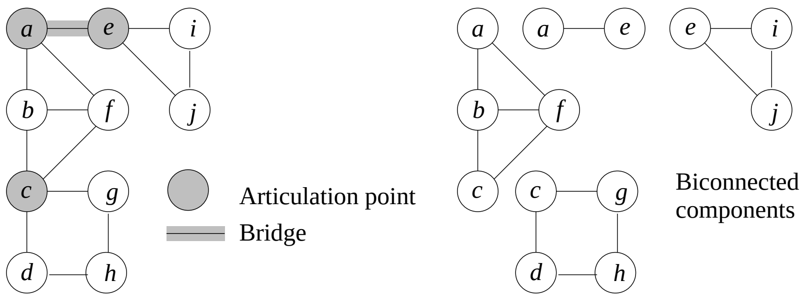

Figure 1: Articulation Points and Bridges

Biconnected graphs and articulation points are of great interest in the design of network algorithms, because these are the "critical" points, whose failure will result in the network becoming disconnected.

Last time we observed that the notion of mutual reachability partitioned the vertices of a digraph into equivalence classes. We would like to do the same thing here. We say that two edges e_1 and e_2 are *cocyclic* if either $e_1 = e_2$ or if there is a simple cycle that contains both edges. It is not too hard to verify that this defines an equivalence relation on the edges of a graph. Notice that if two edges are cocyclic, then there are essentially two different ways of getting from one edge to the other (by going around the the cycle each way).

Biconnected components: The biconnected components of a graph are the equivalence classes of the cocylicity relation.

Notice that unlike strongly connected components of a digraph (which form a partition of the vertex set) the biconnected components of a graph form a partition of the edge set. You might think for a while why this is so.

We give an algorithm for computing articulation points. An algorithm for computing bridges is simple modification to this procedure.

Articulation Points and DFS: In order to determine the articulation points of an undirected graph, we will call depth-first search, and use the tree structure provided by the search to aid us. In particular, let us ask ourselves if a vertex u is an articulation point, how would we know it by its structure in the DFS tree?

We assume that G is connected (if not, we can apply this algorithm to each individual connected component). So we assume is only one tree in the DFS forest. Because G is undirected, the DFS tree has a simpler structure. First off, we cannot distinguish between forward edges and back edges, and we just call them back edges. Also, there are no cross edges. (You should take a moment to convince yourself why this is true.)

For now, let us consider the typical case of a vertex u, where u is not a leaf and u is not the root. Let's let $v_1, v_2, \ldots, v_k$ be the children of u. For each child there is a subtree of the DFS tree rooted at this child. If for some child, there is no back edge going to a proper ancestor of u, then if we were to remove u, this subtree would become disconnected from the rest of the graph, and hence u is an articulation point. On the other hand, if every one of the subtrees rooted at the children of u have back edges to proper ancestors of u, then if u is removed, the graph remains connected (the backedges hold everything together). This leads to the following.

Observation 1: An internal vertex u of the DFS tree (other than the root) is an articulation point if and only there exists a subtree rooted at a child of u such that there is no back edge from any vertex in this subtree to a proper ancestor of u.

Please check this condition carefully to see that you understand it. In particular, notice that the condition for whether u is an articulation point depends on a test applied to its children. This is the most common source of confusion for this algorithm.

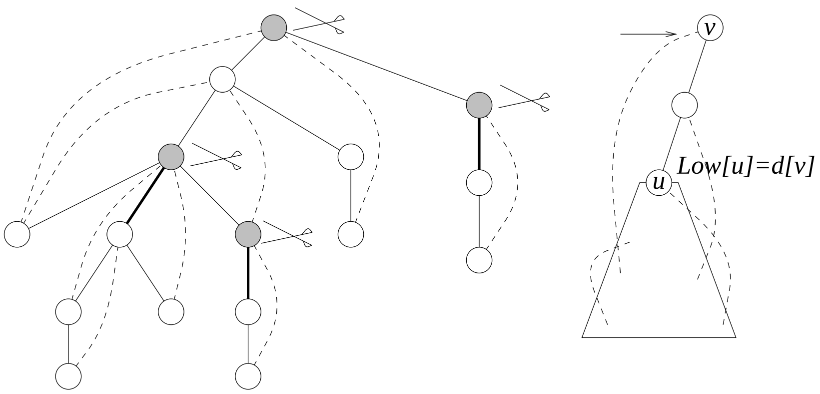

Figure 2: Articulation Points and DFS

What about the leaves? If u is a leaf, can it be an articulation point? Answer: No, because when you delete a leaf from a tree, the rest of the tree remains connected, thus even ignoring the back edges, the graph is connected after the deletion of a leaf from the DFS tree.

Observation 2: A leaf of the DFS tree is never an articulation point. Note that this is completely consistent with Observation 1, since a leaf will not have any subtrees in the DFS tree, so we can delete the word "internal" from Observation 1.

What about the root? Since there are no cross edges between the subtrees of the root if the root has two or more children then it is an articulation point (since its removal separates these two subtrees). On the other hand, if the root has only a single child, then (as in the case of leaves) its removal does not disconnect the DFS tree, and hence cannot disconnect the graph in general.

Observation 3: The root of the DFS is an articulation point if and only if it has two or more children.

Articulation Points by DFS: Observations 1, 2, and 3 provide us with a structural characterization of which vertices in the DFS tree are articulation points. How can we design an algorithm which tests these conditions? Checking that the root has multiple children is an easy exercise. Checking Observation 1 is the hardest, but we will exploit the structure of the DFS tree to help us.

The basic thing we need to check for is whether there is a back edge from some subtree to an ancestor of a given vertex. How can we do this? It would be too expensive to keep track of all the back edges from each subtree (because there may be $\Theta(e)$ back edges. A simpler scheme is to keep track of back edge that goes highest in the tree (in the sense of going closest to the root). If any back edge goes to an ancestor of u, this one will.

How do we know how close a back edge goes to the root? As we travel from u towards the root, observe that the discovery times of these ancestors of u get smaller and smaller (the root having the smallest discovery time of 1). So we keep track of the back edge (v, w) that has the smallest value of $d[w]$.

Low: Define $Low[u]$ to be the minimum of $d[u]$ and

$$\{d[w] \mid \text{where } (v, w) \text{ is a back edge and } v \text{ is a descendent of } u\}.$$

The term "descendent" is used in the nonstrict sense, that is, v may be equal to u. Intuitively, $Low[u]$ is the highest (closest to the root) that you can get in the tree by taking any one backedge from either u or any of its descendents. (Beware of this notation: "Low" means low discovery time, not low in the tree. In fact $Low[u]$ tends to be "high" in the tree, in the sense of being close to the root.)

To compute $Low[u]$ we use the following simple rules: Suppose that we are performing DFS on the vertex u.

Initialization: $Low[u] = d[u]$.

Back edge (u, v): $Low[u] = \min(Low[u], d[v])$. Explanation: We have detected a new back edge coming out of u. If this goes to a lower d value than the previous back edge then make this the new low.

Tree edge (u, v): $Low[u] = \min(Low[u], Low[v])$. Explanation: Since v is in the subtree rooted at u any single back edge leaving the tree rooted at v is a single back edge for the tree rooted at u.

Observe that once $Low[u]$ is computed for all vertices u, we can test whether a given nonroot vertex u is an articulation point by Observation 1 as follows: u is an articulation point if and only if it has a child v in the DFS tree for which $Low[v] \geq d[u]$ (since if there were a back edge from either v or one of its descendents to an ancestor of v then we would have $Low[v] < d[u]$).

The Final Algorithm: There is one subtlety that we must watch for in designing the algorithm (in particular this is true for any DFS on undirected graphs). When processing a vertex u, we need to know when a given edge (u, v) is a back edge. How do we do this? An almost correct answer is to test whether v is colored gray (since all gray vertices are ancestors of the current vertex). This is not quite correct because v may be the parent of v in the DFS tree and we are just seeing the "other side" of the tree edge between v and u (recalling that in constructing the adjacency list of an undirected graph we create two directed edges for each undirected edge). To test correctly for a back edge we use the predecessor pointer to check that v is not the parent of u in the DFS tree.

The complete algorithm for computing articulation points is given below. The main procedure for DFS is the same as before, except that it calls the following routine rather than `DFSvisit()`.

Articulation Points

```
ArtPt(u) {
    color[u] = gray
    Low[u] = d[u] = ++time
    for each (v in Adj(u)) {
        if (color[v] == white) {          // (u,v) is a tree edge
            pred[v] = u
            ArtPt(v)
            Low[u] = min(Low[u], Low[v])  // update Low[u]
            if (pred[u] == NULL) {        // root: apply Observation 3
                if (this is u's second child)
                    Add u to set of articulation points
            }
            else if (Low[v] >= d[u]) {    // internal node: apply Observation 1
                Add u to set of articulation points
            }
        }
        else if (v != pred[u]) {          // (u,v) is a back edge
            Low[u] = min(Low[u], d[v])    // update L[u]
        }
    }
}
```

An example is shown in the following figure. As with all DFS-based algorithms, the running time is $\Theta(n+e)$. There are some interesting problems that we still have not discussed. We did not discuss how to compute the bridges of a graph. This can be done by a small modification of the algorithm above. We'll leave it as an exercise. (Notice that if $\{u, v\}$ is a bridge then it does not follow that u and v are both articulation points.) Another question is how to determine which edges are in the biconnected components. A hint here is to store the edges in a stack as you go through the DFS search. When you come to an articulation point, you can show that all the edges in the biconnected component will be consecutive in the stack.

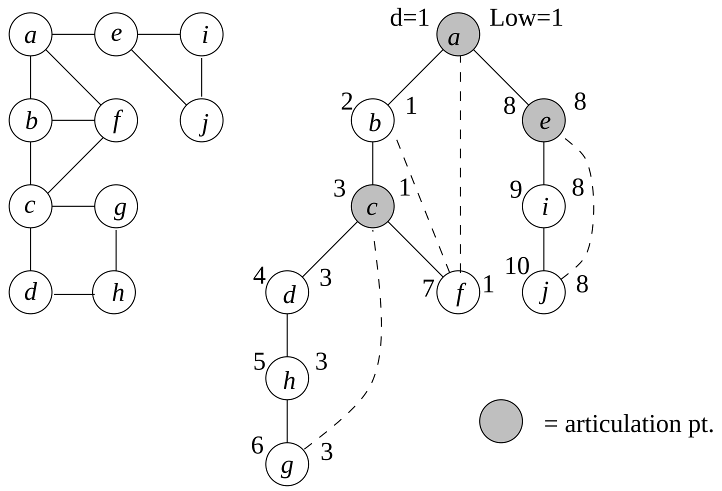

Figure 3: Articulation Points.