CMSC 451: Lecture 4
Bridges and 2-Edge Connectivity
Thursday, Sep 7, 2017

Reading: Not covered in our readings.

Higher-Order Graph Connectivity: (The following material applies *only* to undirected graphs!)

Let $G = (V, E)$ be an *connected* undirected graph. We often assume that our graphs are connected, but sometimes it is desirable to have a higher degree of connectivity. For example, if a graph can be disconnected through the removal of a single edge or vertex, the connectivity is rather "fragile." Here are some definitions:

Bridge: Any edge whose removal results in a disconnected graph (see Fig. 1(a)).

2-Edge Connected: A graph is *2-edge connected* if it contains no bridges (see Fig. 1(b)). In general a graph is *k-edge connected* if the removal of any $k - 1$ edges results in a connected graph.

Here are also vertex-based equivalents:

Cut Vertex: Any vertex whose removal (together with the removal of any incident edges) results in a disconnected graph (see Fig. 1(a)).

Biconnected: A graph is *biconnected* if it contains no cut vertices (see Fig. 1(c)). In general a graph is *k-connected* if the removal of any $k - 1$ vertices results in a connected graph.

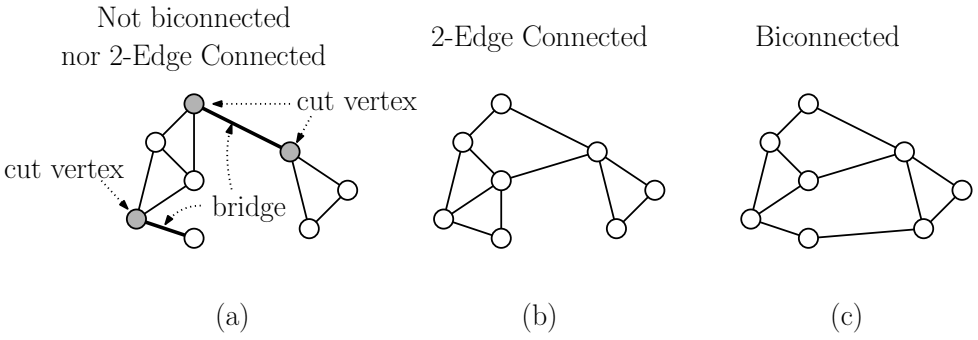


Fig. 1: Higher-order connectivity in graphs

We will present an $O(n + m)$-time algorithm for computing all the bridges of an undirected graph. (The algorithm can be modified to compute the cut vertices as well.)

Although we will not prove it, a useful fact about 2-edge connected graphs is that between every pair of distinct vertices, there are at least two edge-disjoint paths between them. (This is a consequence of a more general result called Menger's Theorem.) If a graph is biconnected, then for every pair of edges there is a simple cycle (that is a cycle that does not repeat any vertices) that contains both edges.

Finding Bridges through DFS: An obvious, but slow, way for computing bridges would be to delete each edge and then apply DFS to determine whether the resulting graph remains connected. However, this would take $O(m(n+m))$ time. We will see that it is possible to identify all the bridges with a single application of DFS in $O(n+m)$ time.

We assume that G is connected, which implies that there is a single DFS tree. Recall that the DFS tree consists of two types of edges: *tree edges*, which connect a parent with its child in the DFS tree, and *back edges*, which connect a (non-parent) ancestor with a (non-child) descendant.

Suppose that we are currently processing a vertex u in DFSvisit, and we see an edge (u, v) going to a neighbor v of u. If this edge is a back edge (that is, if v is an ancestor of u) then (u, v) cannot be a bridge, because the tree edges between u and v provide a second way to connect these vertices. Therefore, we may limit consideration to when (u, v) is a tree edge, that is, v has not yet been discovered, and so we will invoke DFSvisit(v). While we are doing this, we will keep track of the back edges in the subtree rooted at v. Observe that all these back edges remain entirely within this subtree (see Fig. 2(a)) then (u, v) is a bridge, since its removal completely disconnects this subtree from the rest of the tree. On the other hand, if there is even a single back edge leading out from this subtree, then (u, v) is *not* a bridge. Such a back edge must go from within the subtree to a proper ancestor of v. By the Parenthesis Lemma, this means that it leads to a vertex whose discovery time is strictly smaller than v's discovery time. (Recall a vertex's ancestors are discovered before it.) In summary, we have established the following claim.

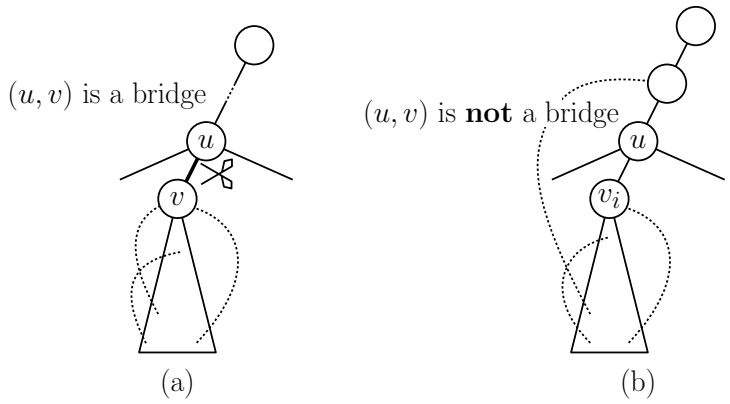


Fig. 2: Conditions for a vertex to be a cut vertex.

Claim: An edge (u, v) is a bridge if and only if it is a tree edge and (assuming that u is the parent of v) there is no back edge within v's subtree that leads to a vertex whose discovery time is strictly smaller than v's discovery time.

Tracking Back Edges: The above claim provides us with a structural characterization of bridges. How can we design an algorithm that tests this condition? To do this, we will introduce an auxiliary quantity, which will be computed as the DFS runs. We define

$$\text{Low}[u] \;=\; \min(d[u], \min\{d[w] : \exists \text{ back edge } (v, w) \text{ where } v \text{ is a descendant of } u\}.$$

Note that we use the term "descendant" in the nonstrict sense, that is, v may be u itself.

Intuitively, Low$[u]$ is the closest to the root that you can get in the tree by taking any one back edge from either u or any of its descendants. (Beware of this notation: "Low" means low discovery time, not "low" in our drawing of the DFS tree. In fact Low$[u]$ tends to be "high" in the tree, in the sense of being close to the root.) Also note that you may consider *any* descendant of u, but you may only follow *one* back edge.

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$[u] = \min(\text{Low}[u], d[v])$. 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 (see Fig. 3(a)).

Tree edge (u, v)**:** Low$[u] = \min(\text{Low}[u], \text{Low}[v])$. 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 (see Fig. 3(b)).

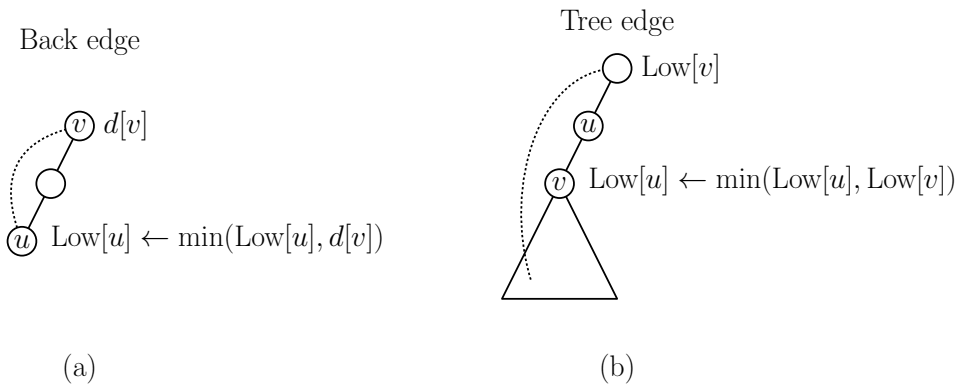


Fig. 3: Cut Vertices and the definition of Low$[u]$.

The code block below shows how to compute Low$[u]$ for all vertices. We do not both computing finish times, since they are not needed for our purposes. Note that there is a subtlety in determining whether an edge is a back edge. Clearly, such an edge must go to a previously discovered vertex (which is why it is in the else-clause), but we also need to check that this vertex is not u's parent. Recall that every edge of an undirected graph is reflected twice in the adjacency list (with v as a neighbor of u and u as a neighbor of v). We need to check that we are not simply seeing the tree edge again, but from the child back to the parent. To do this, we check v is *not* u's parent, that is $v \neq \text{pred}[u]$.

Observe that once Low$[u]$ is computed for all vertices u, we can test whether a given tree edge $(\text{pred}[v], v)$ is a bridge by testing whether there is no back edge in v's subtree going to an earlier discovered vertex, that is, $d[v] = \text{Low}[v]$. (Actually, the test is more naturally stated as $d[v] \geq \text{Low}[v]$, but by definition $\text{Low}[v] \leq d[v]$, so testing for equality is equivalent.) The final code is shown in the code block below.

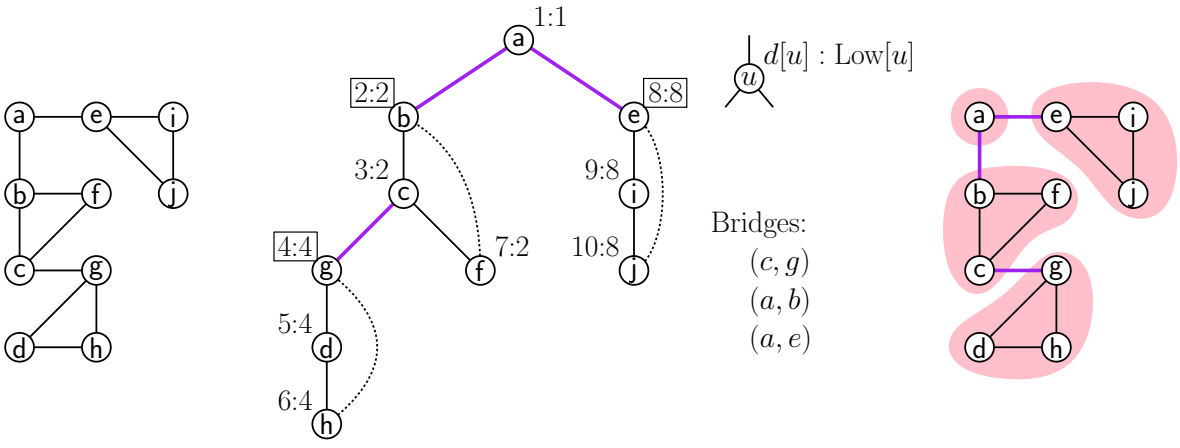
Modification of DFSvisit for Low computation

```
DFSvisit(u) {
    mark[u] = discovered
    Low[u] = d[u] = ++time                      // set discovery time and init Low
    for each (v in Adj(u)) {
        if (mark[v] == undiscovered) {          // (u,v) is a tree edge
            pred[v] = u                         // v's parent is u
            DFSvisit(v)
            Low[u] = min(Low[u], Low[v])        // update Low[u]
        }
        else if (v != pred[u]) {                // (u,v) is a back edge
            Low[u] = min(Low[u], d[v])          // update Low[u]
        }
    }
}
```

Computing Bridges via DFS

```
findAllBridges(G) {
    time = 0
    for each (u in V)                           // initialize
        mark[u] = undiscovered

    for each (u in V)
        if (mark[u] == undiscovered)            // undiscovered vertex?
            DFSVisit(u)                         // ...start a new search here
    for each (v in V) {                         // check for the bridges
        u = pred[v]
        if (u != null and d[v] == Low[v])
            output (u, v) as an bridge
    }
}
```



(a) (b) (c)

Fig. 4: Computing bridges via DFS.

Wrapup: We have shown how to compute bridges in an undirected graph. There are a number of interesting problems that we still have not discussed. First, we claim that it is possible to adapt this algorithm to compute cut vertices as well. (The computation of Low is the same, but a different condition is applied to determine which vertices are cut vertices.) Second, if a graph fails to be 2-edge connected, it may be desirable to partition the vertices of the graph into 2-edge connected components. For example, in Fig. 4(c) the components consist of $\{d, g, h\}$, $\{b, c, f\}$, $\{a\}$, and $\{e, i, j\}$. This can be done by simple extension as well. (The vertices are stored on a stack, and whenever a bridge is detected, we pop off an appropriate subset of the stack. We will leave the details as an exercise.) A similar approach can be applied to computing the biconnected components of a graph, which is a partition of the edge set of the graph.