Daily Coding Problem #262

Problem

This problem was asked by Mozilla.

A bridge in a connected (undirected) graph is an edge that, if removed, causes the graph to become disconnected. Find all the bridges in a graph.

Solution

Let's start with an example. Suppose we have the following graph, represented by an adjacency list.

```
graph = {
    0: [1, 2, 3],
    1: [0, 5],
    2: [0, 3],
    3: [0, 2, 4],
    4: [3],
    5: [1]
}

"""
2 --- 0 --- 1 --- 5
  \   |
   \  |


      3 --- 4
```

```
"""
```

The bridges here should be `(0, 1)`, `(1, 5)`, and `(3, 4)`, since removing any of these would disconnect the graph.

One idea for identifying these bridges would be to look for vertices with only one edge, since it is guaranteed that removing this edge will leave that vertex stranded. While this works for `(1, 5)` and `(3, 4)`, it cannot find edges that lie on a unique path to a one-edge vertex, like `(0, 1)`.

To resolve this issue, we need to find some way to track which vertices are "reachable" from other vertices. If we can manage this, we can identify an edge `(u, v)` as a bridge if we come across `u` before `v`, and `v` cannot reach `u` or any earlier node.

More precisely, we can perform a depth-first search, maintaining an array with the time of appearance of each vertex. If a vertex is explored on the first level of the search, `appeared[v] = 0`. For the second level, `appeared[v] = 1`, and so on. This tells us which nodes come before other nodes.

At the same time, we store an array that holds the lowest level reachable from any vertex, which will initially be its own depth. For example, suppose we start our search with vertex 5 in the graph above. The value of `reach[5]` initially will be `0`. Next we visit vertex 1, where we set `reach[1]` to be 1. We continue our search in this way, increasing the depth at each step. When the stack eventually comes back to 1, `reach[1]` will be unchanged. In this way, we learn that it is impossible to form a path from 1 back to `0`, and so `(0, 1)` must be a bridge.

```
def visit(graph, u, v, depth, reach, appeared, bridges):
    appeared[v] = reach[v] = depth

    for neighbor in graph[v]:
        if appeared[neighbor] == -1:
            visit(graph, v, neighbor, depth + 1, reach, appeared, bridges)

            if reach[neighbor] == appeared[neighbor]:
                bridges.append((v, neighbor))

            reach[v] = min(reach[v], reach[neighbor])

        elif neighbor != u:

            reach[v] = min(reach[v], appeared[neighbor])
```

```
def find_bridges(graph):
    reach = {v : -1 for v in graph}
    appeared = {v : -1 for v in graph}

    start = list(graph.keys())[0]
    depth = 0
    bridges = []

    visit(graph, start, start, depth, reach, appeared, bridges)

    return bridges
```

The time complexity of this algorithm is the same as that of any depth-first search, $O(V + E)$. We will need to store V values in our `reach` and `appeared` arrays, and there cannot be as many bridges as vertices, so our space complexity is bounded by $O(V)$.
