Title

Ian Benlolo 260744397
McGill University
October 15, 2017

1. Since a strongly connected component of a graph is a strongly connected subgraph. The number of strongly connected components can only really decrease.
   Let's start with $n = 2$ separate nodes. If you connect them using an edge, you still have two strongly connected components. Now if you add another edge, you go down to one strongly connected component which decreased the amount of strongly connected components.
   This logic follows for any number of strongly connected components because say you have n nodes with $m \leq n$ strongly connected components, if you add an edge between two nodes, they will either remain two separate strongly connected components or will merge into one (as shown above) by creating a cycle.

2. We know the following:

   - if there is an edge from u to v then u cannot be the celebrity.
   - If there is no edge from u to v then v cannot be the celebrity.

   So every time we check if there is an edge between two vertices we can eliminate one of them until we are left with one node, the celebrity (say c).

```
Algo findCeleb(V, E)
input --> a graph of vertices V and edges E
output --> a vertex V
put all vertices in a list, L
while there are more than 2 elements in L
  x = remove(L)
  y = remove(L)
  if there is edge from x to y AND an edge from y to x
    continue //meaning remove both from the list, neither can be a celebrity
  if there is no edge from x to y AND no edge from y to x
    continue //again, remove both
  if there is an edge from x to y and none from y to x
    remove x
  if there is no edge from x to y but there is an edge from y to x
    remove y
end while
check if the final candidate is a celebrity and return answer
```

   This algorithm removes at least 1 candidate (or vertex) at every iteration so the loop can only run O(n) times, the algorithm therefore runs in O(n) time.

3. This is a graph traversal problem, where the path has to be smaller than a certain length. We have a set of all the flights (edges) and their start/finish places (nodes).
   For a runtime of O(m + nlog n) we will sort sort (using a heap) the flights by shortest flights first and in increasing amount of time taken for the flight $(d_k - t_k)$ in O(n log n) and also run a BFS through all the airports.
   To run this BFS algorithm we will keep track of a few things as we go through every k-layer (at k^{th} flight).
   We check if the time of departure and the time of arrival of the previous flight overlap. If they do not

then you can continue on this "layer". If they do not overlap then terminate BFS for this node.
Now, we check whether the departure time is less than the arrival time and if it is we continue the BFS algo.
If the currentNode is headed to the original airport is the destination airport and the times work (so s is less than t) then return true. Else continue.
If we reach outside the main for loop then return false.
The run time for this algorithm uses heapify, so O(n log n) and the BFS takes O(m), so the algorithm runs in O(m+ n log n).

4. We will solve this in a brute force manner. First, note that for this algorithm to work we must have two or more paths from s to t. Also, at every step there is a check whether any adjacent edge is empty; meaning no stone on adjacent node.

   We will first run two BFS algorithms; one from s and the other from t. As we run them, we will store every path from s to t and vice versa by an array of the sequence of nodes from s to t (and t to s).
   We then compare all the paths of similar sizes (starting from 3 otherwise they'd be adjacent). For each comparison, we will start at the first vertices of each path and test that they are neither the same or adjacent. If we get to the end of the paths without failing the tests at each step we can return that the minimum steps required are the size of the paths -1.

5. To find the diameter of any tree t on n nodes, we will use two methods. The first called BFSlast(t, n) on a tree t and any node n which will return the last node, say x, reached by the iterative BFS. This would be implemented with a queue. This part would run in O(m+n+1) since for every node in the tree we need to keep track of their distance from the input node.
   The following step in finding the diameter of a tree would be to call a method called BFSmaxdist(t, n) on the node returned from BFSlast(). This would assign a distance to each node from the node by maintaining a count that starts at 0 and increments by 1 at every call. At termination, this method returns the distance of the furthest node reached. This is the diameter of the tree. Assigning a count to every node runs in O(n). Initializing and returning the count each take O(1). BFSmaxDist() runs in O(m+n+n+2), where $m = n - 1$ since there are no cycles.
   The whole algorithm therefore runs in O(3n+2+2n)=O(5n+1)=O(n).

6. Let the labels be denoted L(v) for vertex v.

```
For every undiscovered vertex v in G starting with the L value
  min(v)=L(v)
  Mark v discovered
  DFS_reverse(v)
    for every undiscovered vertex u discovered by this DFS
      min(u) = min(v)
      mark u discovered
return min
```

   What this does is it starts a DFS but on the reverse graph starting from the node in question. In the main loop of the DFS though, it considers L(v) in increasing values. If vertex u is in the depth first tree starting at v then min(u)=v.