ComS 311
Recitation 3, 2:00 Monday
Homework 4

Sean Gordon

October 25, 2019

Algorithm 1 Define G^2 from G using paths of length 2, excluding cycles.

```
Assume G is stored in "G"

Create empty adjacency list named "G2"

#For every vertex...
for all list in G do
    start = current vertex
    G2.add(start)

    #For every vertex this points to...
    for all vertex in list do
        innerList = G.get(vertex)

        #For every vertex that that vertex points to...
        for all vertex in innerList do

            #If this vertex is the start (u == v)
            if vertex == start then
                continue
            end if

            #Add this edge (of length 2) to the new graph
            G2.get(start).add(vertex)
        end for
    end for
end for
```

The runtime of this algorithm is as follows:
1 for loop through every vertex $\Rightarrow$ O(V)
1 for loop through every edge $\Rightarrow$ O(E) with
1 for loop through every edge $\Rightarrow$ O(E)
This combines to become $O(V^*E^2)$

Algorithm 2 Find the number of shortest paths from s to vertex i.

```
Assume G is stored in adjacency list "G"

Create object Pair that stores two Integers
Create an array paths of size V
The array will store path length and count for each vertex in a Pair obj

//Perform breadth first search on the graph ————————————

//Create a queue for BFS that holds depth and the vertex in a Pair
LinkedList<Pair> queue = new LinkedList<Pair>();
boolean visited = new boolean[V];

//Mark the current node as visited, add it to the array, and enqueue it
visited[s] = true;
paths[s] = new Pair(0, 1);
queue.add(new Pair(0, s));

while queue.size() != 0 do
    //Dequeue a vertex
    Pair pair = queue.poll();
    int depth = vertex.depth;
    int vertex = vertex.node;

    Iterator iterator = G[vertex].listIterator();
    while iterator.hasNext() do
        int v = iterator.next();

        if !visited[v] then
            visited[v] = true;
            paths[s] = new Pair(depth+1, 1);
            queue.add(new Pair(depth+1, v));

        else if paths[v].length == depth+1 then
            //If this depth == the one already stored, this is a shortest path
            paths[v].count = paths[v].count + 1;
        end if
    end while
end while
return paths[i].count;
```

Proof by Induction:

Starting at vertex 1, we loop through each vertex 1_2 it is linked to, and in turn each vertex 1_3 those are linked to, creating an edge in our graph from $1 \rightarrow$ each 1_3. This ensures each edge is of length 2 in G.
Next, we assume this works for vertex k.
Then, the vertex next in line in the adjacency list would be vertex (k+1). There is no difference in the process between each vertex, and each is inspected separately from the others. Therefore, all verticies $1 \rightarrow$ n will be treated the same.

Runtime for above algorithm:
1 while loop through each vertex $\Rightarrow$ O(V)
1 while loop through each edge of each vertex $\Rightarrow$ O(E)
These two combine to become O(V+E)

3a) *Prove that every DAG (Directed Acyclic Graph) has a sink.*

Let G be a directed graph with number of verticies n, each with at least one outgoing edge. To prove the claim we show that if there is no sink, there must be a cycle.
Picking any vertex u, we begin to follow each edge outward. If there are no sinks, we will be able to continue to node v, then w, and so on. However, with a graph of order n, we must eventually reach a previously seen vertex after at most n+1 steps. This is clearly a cycle, breaking the acyclic assumption made earlier.

Algorithm 3 Compute topological ordering of a DAG.

```
Require: G is stored in adjacency list "G"
  Create an array visited of size V, with all indicies initialized to false
  Create an empty queue queue to store vertex order

  topSort(0) //Call recursive function with first vertex

  function topSort(int vertex)
      visited[vertex] = true
      List linked = G.get(vertex)

      for all vertex v in linked do
          if visited[v] then
              continue
          end if

          topSort(v)

          queue.add(v)
      end for
  end function

  Print out queue, or do something else with it
```

This algorithm computes the topological ordering by counting on the fact that it will eventually reach a sink vertex and be able to return up the chain. Without a sink/with a cycle, this algorithm cannot perform.

4) Prove that G' is a DAG:

It is given that verticies v and u in G' each represent a strongly connected component in G, and that each edge e in G' represents a connection between two strongly connected components in G.
We know that G' is a DAG because it is required that v and u must be *distinct* strongly connected components. If G' were not a DAG, there is the possibility that for a single edge from v $\rightarrow$ u, there would be an edge running from u $\rightarrow$ v. This however would mean that v and u in G' and their counterpart groups in G are strongly connected, breaking the design rules of the graph.