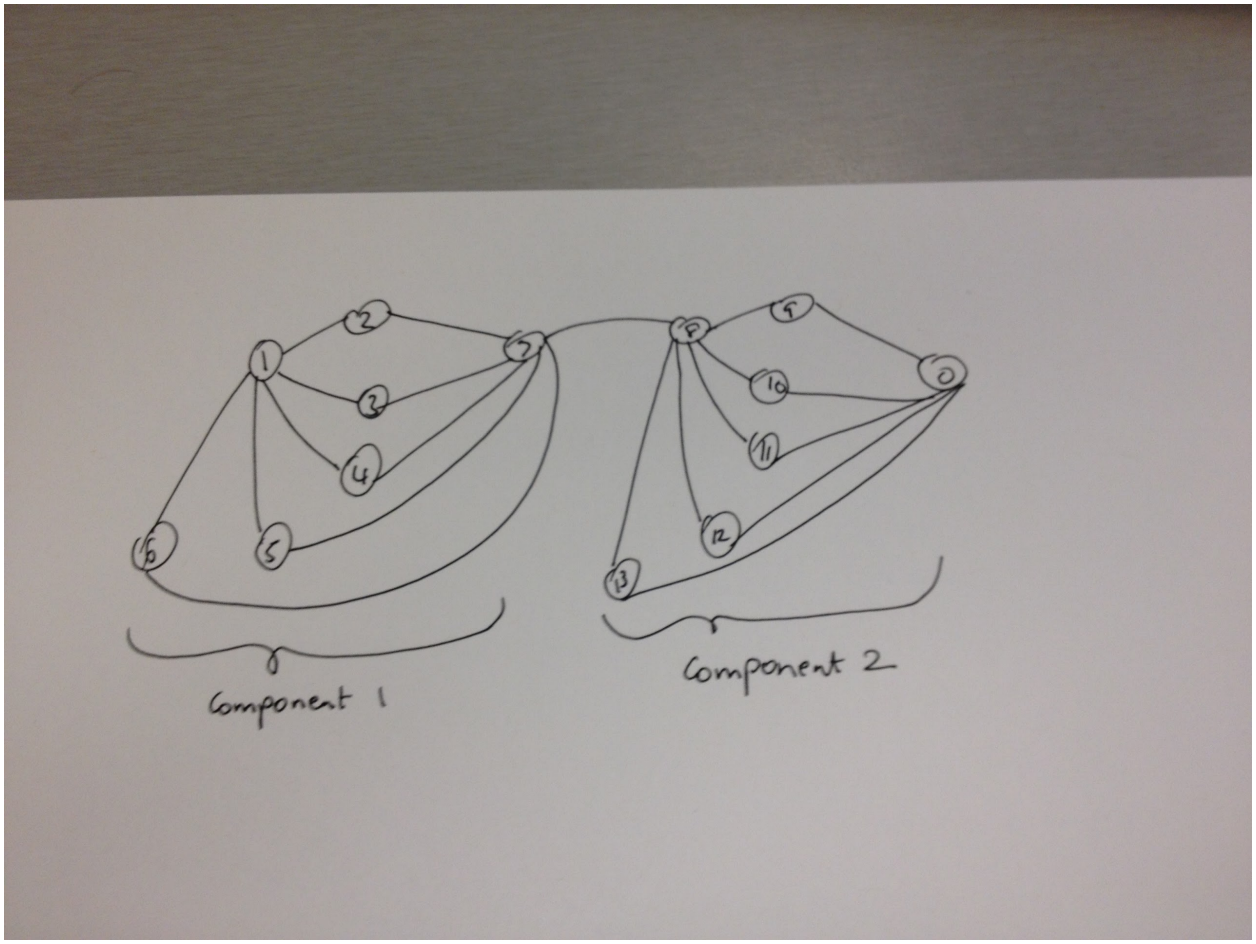
Mihir Jham
CS251
Project 4 Report
04/23/2014

CS251 - Project 4 Report

1) - Breadth first search the graph that has been given with the the first vertex(v)
- Breadth first search the graph that has been given with the second vertex(w)
- Go through all the vertices in the graph, and check if the first BFS and the second BFS have a path to the current vertex in the loop.
- If both BFSs have a path to the current vertex, sum up the current distances of both v -> current vertex and w -> current vertex.
- Check if the previous computed distance is smaller than the current computed distance. If so, update the minimum distance and set the ancestor to be the current vertex.

2)



Edge 7-8 appears the most in the generated spanning trees. The other edges have a random nature in terms of frequency. As can be seen from the image above, edge 7-8 connects the 2 “components”. The reason why this edge appears in all the generated

spanning trees is because this edge has to be passed to get to the other side. All the other edges are enclosed within their connected components.

3)

Undirected Graph	Clustering Coefficient
tinyG.txt	0.307692
tiny2G.txt	0.533333
circlesG.txt	0.000000
starG.txt	0.583333
mediumG.txt	0.633382

4) |V| = # of vertices in graph
|E| = # of edges in graph

Initializing EdgeCounter table = O(|E|)
Assigning random weight to each edge = O(|E|)
Computing MST of graph(using Prim's algorithm) = O(|E|log|V|)
Checking whether edges appear in the generated spanning trees = $O(|E|^2)$
Computing the clustering coefficient = $O(|V||E|^3)$
Sorting the EdgeCounter table = O(|E|log|E|)

Checking whether the edges in the generated spanning trees is $O(|E|^2)$ because we go through all the edges that the MST algorithm creates, which in the worst case is O(|E| - 1). And, for every edge in the MST, we check all the edges in the EdgeCounter table to see increment the count. Therefore, it's complexity is $O(|E|(|E| -1) = O(|E|^2 - |E|) = O(|E|^2)$

Computing the clustering coefficient is $O(|V||E|^3)$ because we go through the adjacency lists of all the vertices and we check to see if the edges exists in the whole graph.

O(CriticalEdges.java) = $O(|V||E|^3)$

Mihir Jham
CS251
Project 4 Report
04/23/2014

5) Highest Betweenness Centrality : 7, 8, 0
Lowest Betweenness Centrality : 9, 10, 11
Highest Closeness Centrality : 7, 8, 13
Lowest Closeness Centrality : 11, 0, 1

Betweenness centrality measures the number of times a vertex acts as a bridge along the shortest path between 2 other vertices.
Closeness centrality is a measure of how long it will take to spread information from one vertex to all other vertices sequentially.

6) |V| = # of vertices in the graph
|E| = # of edges in the graph

Initializing VerticeCounter Table = O(|V|)
Computing the betweenness and closeness centrality
$= O(|V|(|E| + |V|\log|V| + |V|(|E|))$
$= O(|V||E| + |V|^2\log|V| + |V|^2|E|)$
$= O(|V|^2|E|)$
Sorting the tables to find out highest betweenness and closeness centrality
$= O(|V|\log|V|)$

Computing the betweenness and closeness centrality takes $O(|V|^2|E|)$ as it goes through all the vertices in the graph, and for each vertex, we compute the shortest path from that vertex to all the other vertices. Assuming Dijkstra's shortest path algorithm takes $O(|E| + |V|\log|V|)$, we check if there is a shortest path available for all the other vertices, and if so, we sum up the distances. We also go through all the edges that are a part of the path to every other vertex, and increment how many times each vertex occurs in all the shortest paths for all vertices.

O(CriticalVertices.java) = $O(|V|^2|E|)$