Sander VanWilligen

Zackery Lovisa

**Com S 311 – Project 2 Report**

**Data Structures for Q and Visited:**

Text

**Questions:**

In the WikiCS.txt graph, there are 500 vertices (including the root given), and there are 23954 edges (no duplicates).

The vertex with the largest out degree is /wiki/Computer\_Science, with a degree of 499 (meaning the program had found 500 links total by the first page).

There are 9 strongly connected components.

The largest component is 492 (this means there is one massive component, and the other 8 are all single node SCCs, with no connections back to the other pages).

**GraphProcessor Data Structures:**

For the actual graph that we created from wikiCS.txt, we used a hashmap of integer keys and GraphNode objects. A GraphNode object is just an object containing the string for the given node, and a list of outward connections. The integer key is a hash of the node string. By storeing the map in a hashmap, we have constant (on average) search time for a specific string(node) in the map.

We stored the SCCs as an arraylist of hashmaps, but these hashmaps are not the same as the graph hashmap. Since none of the methods need to know the edges in each SCC, just the nodes, we do not store a hashmap of Integer, GraphNode pairs, but just one of Integer, String pairs. By doing this, we still get the constant search time for a node, but do not need the extra memory space. The SCC hashmaps themselves are stored in a simple arraylist. The number of SCCs is small.

Finally, we had one final data structure that we use during the BFS method to search for a path between two nodes. I will go into this in the method analysis below.

**Method Analysis:**

outDegree:

Because we store the nodes in a hashmap, we have constant access time to retrieve a node from the graph. Once we have the node, getting the outDegree is just a matter of retrieving the size variable of the connections list. The runtime is O(1).

sameComponent:

We now have constant access time for a node, but we still have to go through each SCC. Given n SCCs, the time is O(n). in the worst case, each ndoe would be its own SCC and this would not be great, but on average there are very few SCCs for a web graph. The graphs are almost always strongly connected. So, the actual time is a lot closer to O(1) most of the time.

componentVertices:

Finding the right SCC takes O(n) access time in the worst case, given n SCCs, but again it is closer to O(1) because of the constant access time for a node. After we find the correct SCC, we need to convert it to an arraylist from a hashmap, which takes O(k) time, where k is the number of elements in the SCC. As mentioned, k is often close to n, the size of the graph. So, most of the time, this method is taking O(n) time.

largestComponent:

We keep track of this while building the SCCs, so this is constant O(1) time.

numComponents:

We also keep track of this while building the SCCs, so this is also constant O(1) time.

bfsPath:

This method is a little more complex than the others, but the advantages of using hashmaps still come into play here as they do with the other methods. We use the same algorithm as used in class, but for this method we also need another data structure to keep track of the path so that we can output it. The data structure we use is a type of tree, where each node of the tree contains the string associated with the node, and a reference to the parent. When the path is found, this tree allows us to iterate back upwards to create the path that we output. The algorithm time is the same as the one in class (O(n+m)), + the time to iterate back up the tree. At worst, this would be O(n), so the runtime of the algorithm is still O(n+m).

Overall storing the graph and SCCs in hashmaps saves us a lot of time in all of the methods and in the graph creation.