Eva Kuntz & Merritt Harvey

COMS311: Project 2

Report

First, we will discuss the data structures that we used for Q and visited, then we will describe our graph. Lastly, we will report some of the statistics regarding our graph using the GraphProcessor class.

For Q we used a queue and for visited we used a hashset. We used a linked-list queue for q because it was the easiest way to implement a BFS traversal, and it makes sense. We want to visit nodes in the order we “see” them, so, we want a LILO data structure, and a queue provided the functionality we needed. We used a hashset for visited because it gives us quick (near O(1)) lookup times to check if an element is in the set and, since that the “contains” method is the only operation we need to perform on the set, the performance of the other methods for a hashset do not matter to us.

Our graph in WikiCS.txt had a total of 200 vertices and 2867 edges.

The vertex in our WikiCS.txt graph with the highest out degree was /wiki/Computer\_Science with a total of 199 outgoing edges.

The diameter of our graph was 400.

The vertex in our WikiCS.txt graph with the highest centrality was /wiki/Computer\_Science with a degree of centrality of 9062.

The runtimes of the public methods in GraphProcessor are as follows:

1. outDegree

For this method, we find the vertex we want in our graph, then return the size of the HashTable used to store all the outgoing edges from the vertex. So:

Find vertex v in graph G //this takes O(1) because we use a hashTable to store the vertices

Return size of v’s edges //this takes O(1) because we use a hashTable to store edges

Thus, the runtime of this method is O(1\*1) = O(1).

1. bfsPath

For this method, we do a bfs traversal of the graph starting from the vertex u and keeping track of a “parent” list. This way, once we have finished the bfs traversal, we can traverse “backwards” through the parent list (starting from vertex v) to obtain the proper bfs path. So, we have: (Note, we used a HashMap for our “parent” array, which gives us O(1) time to check if a given element is contained in the map.)

Queue Q = {u}

Add u to visited

Parent(u) = null //our starting vertex does not have a parent

While( Q is not empty)

Remove Vertex vertex from Q

For each outgoing edge <vertex,y> from vertex

If y is not in visited

Add y to Q

Add y to visited

Parent(y) = vertex //set parent

//end of while loop

If(parent array does not contain vertex v)

Return empty path

Add v to our path

Parent = get parent of v

While(parent != null)

Add parent to path

Parent = parent of the current parent //this is our “backwards” traversal

//end of while loop

For each vertex vert in our path

Push vert onto stack

While(stack is not empty)

add push.pop() to path

return path

Now, the runtime of this algorithm is (n+m)—for the first while loop as we iterate over all the vertices and all the edges in our graph, plus (1) for checking if our parent array contains v, plus n + n + n for adding vertices to our path via the parent array, for adding each vertex from the path to a stack, then for removing each vertex from the stack and adding them to the path once again. Thus, the runtime of our bfsPath algorithm is: n+m+1+n+n+n = O(n+m).

1. centrality

For centrality, we check the bfs traversal for each pair of vertices in the graph and check if the path contains the input vertex x. So:

Input: String x

Int count = 0;

For each vertex v in graph G

For each vertex u in graph G

Compute bfsPath(v,u)

If(bfs path contains x) //takes O(n) time worst case because we use ArrayList

Count++

Now, this method takes n \* n \* (n+m + n) = O(n2(m+n)).

1. diameter

For diameter, we check the bfs traversal for each pair of vertices in the graph and keep track of the maximum. So:

For each vertex v in graph G

For each vertex u in graph G

Compute bfsPath(v,u)

If(size of path = 0)

Return 2n

If(size of path > max path length so far)

Max path length so far = size of path

So, this method takes n \* n \* (bfsPath time) = n \* n \* (n+m) = O(n2(n+m)).