***CITS2200 Centrality Project***

Nicholas Walters, 22243339

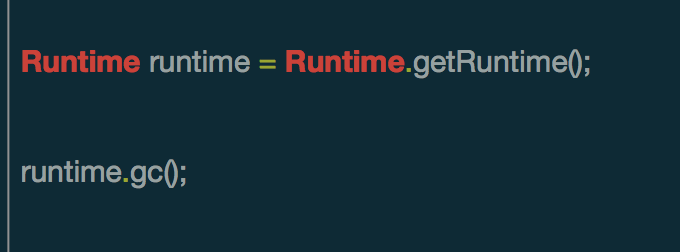
James Caldon, 22226341

**Performance Study**

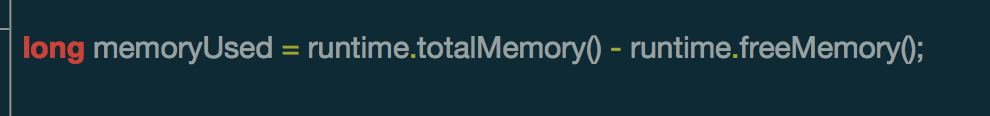
For this analysis of the code, we have measured the time used for various sizes on inputs.

Also, the method to measure the memory used is included in the java runtime environment.

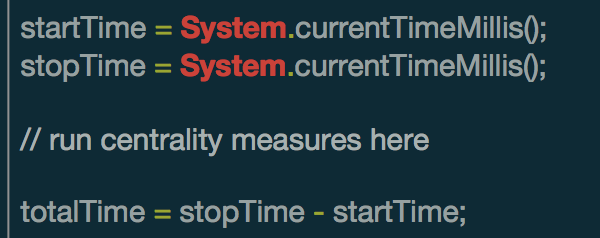
Runtime is calculated, gc() is passed, as the garbage collector can be run to ensure we only calculate the in use memory.

****

Then we can calculate the total memory used.

****

The total time is calculated also by using java built in system

****

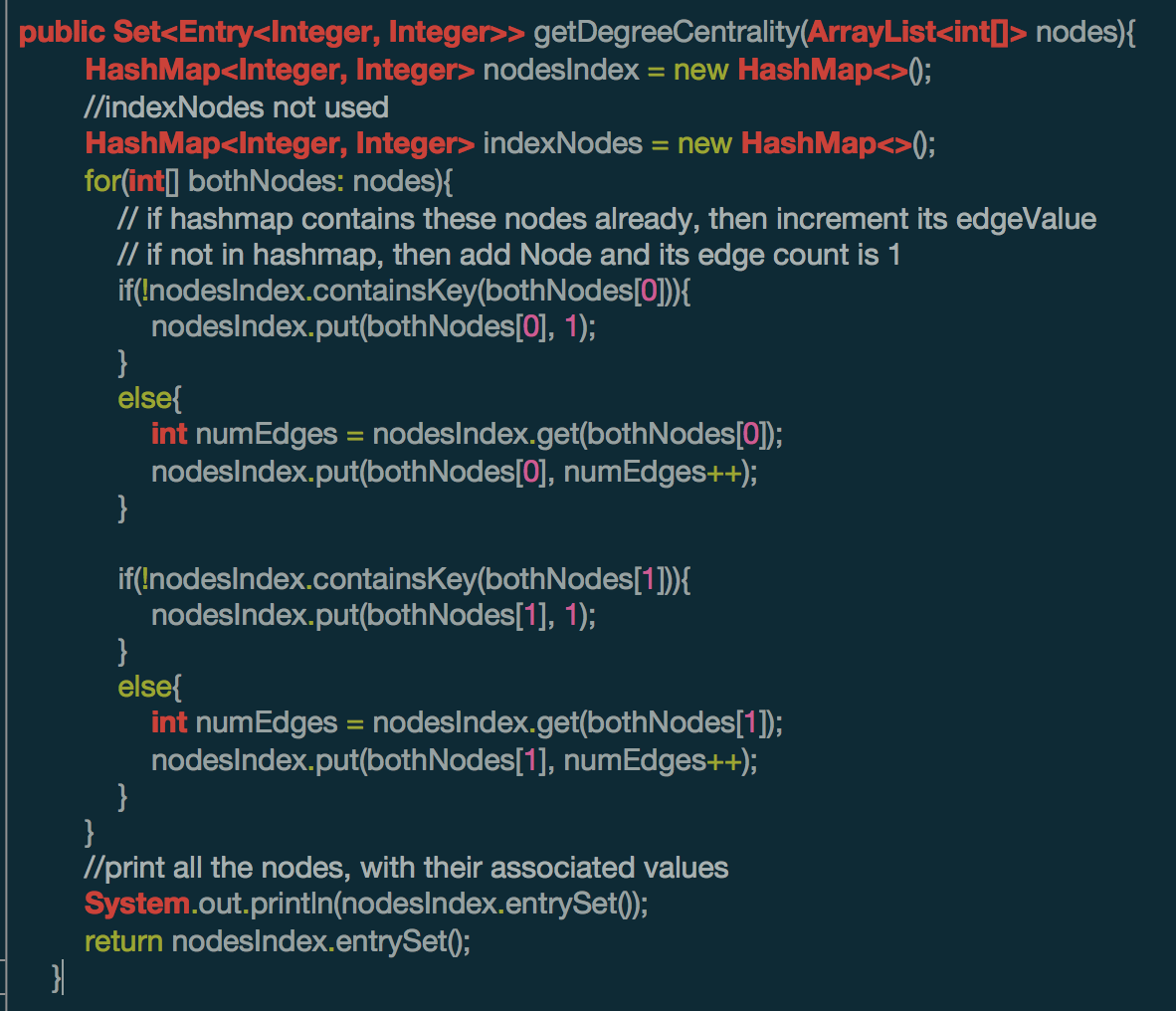
**Algorithm Analysis**

**getDegreeCentrality ()**

The degree centrality is the most connected Node on a graph with edges. The highest number of edges incident on a node.

getDegreeCentrality () returns a large set, containing the NodeID’s and number of edges respectively.

We used a Hash Map implementation to store these two values. For each new value in the nodes list, we added it to the HashMap and set a starting value of 1 edge. For each existing node, we have incremented the existing edge value by one.



1

Time complexity

O(n)

as expected of a method like this, this will run in time proportional to input

O(n) time complexity

This is because we have to scan through the provided nodes list one by one and increment the Hash Map’s each time. Also, as evident of the text box “1” in the image, there is a loop representing this

**getClosenessCentrality ()**

**Closeness** is a measure of the degree to which an individual is near all other individuals in a network. It is the inverse of the sum of the shortest distances between each node and every other node in the network. **Closeness** is the reciprocal of farness. A high degree of closeness is when a node is very close to other most other nodes in the network, without taking too long to reach them.

* This method needs an adjacency matrix, containing all the connecting edges in order to generate closeness
* This method returns an array of integers, indicating all of the closeness values for each node in graph

****

2

1

We implemented an adapted Dijkstra's Algorithm, published by Edsger W. Dijkstra in 1959. It creates trees of shortest paths starting from the initial vertex and ending at every other vertex. We find matching tree with the ending vertex and return the distance of the tree.

**Time Complexity - Closeness**

Best Case:

O(n)

Worst Case:

O (s \* n)

n = size of the adjacency matrix

s = length of the adjacency matrix

i.e. (items in first loop) \* (items in second loop)

(adj. length \* size)

For ALL methods in this report, we will assume that the computer can operate these following commands in a single instruction cycle.

* Looking up values in array (e.g. visited)
* Assigning values to array
* Comparing two values
* Getting a value from HashMap with a key
* Assigning a value to a HashMap with a key
* Arithmetic operations

Therefore, the only operations that will affect the following methods are the loops within it.

For the method getClosenessCentrality () we have to scan through the adjacency matrix twice with 2 loops.

**Data Structures:**

**Priority Queue**

This allows us to get the vertexes with the greatest closeness centrality ordered for us

**getBetweenessCentrality ()**

Betweenness centrality, get the number of shortest paths pass through a vertex. It is one of the most important network analysis concepts for assessing the (relative) importance of a vertex. The famous state-of-art algorithm of Brandes [2001] computes the Betweenness centrality of all vertices in O(mn) worst-case time on an n-vertex and m-edge graph. This is OUR OWN implementation of Brandes algorithm.

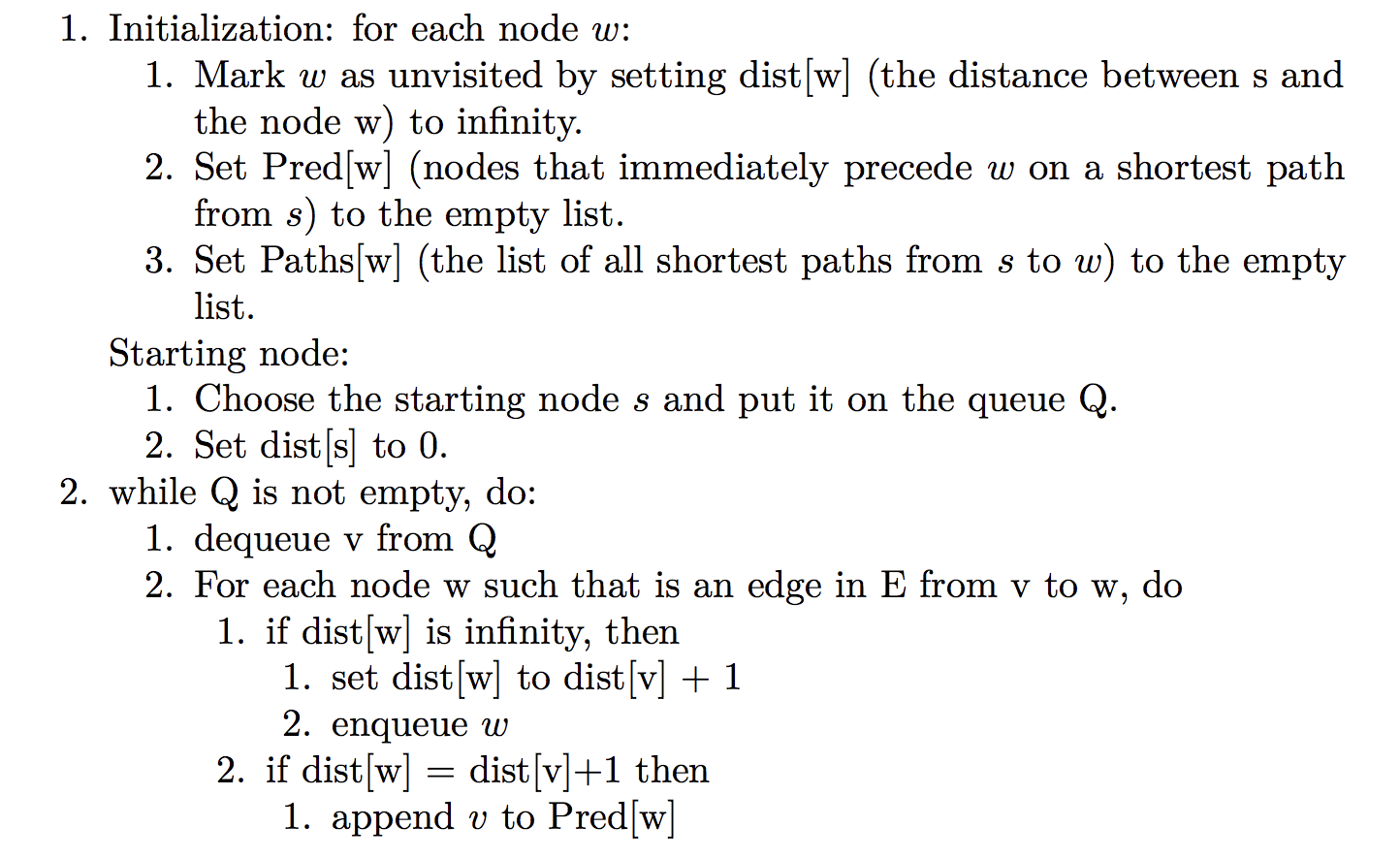
Brande’s Algorithm:

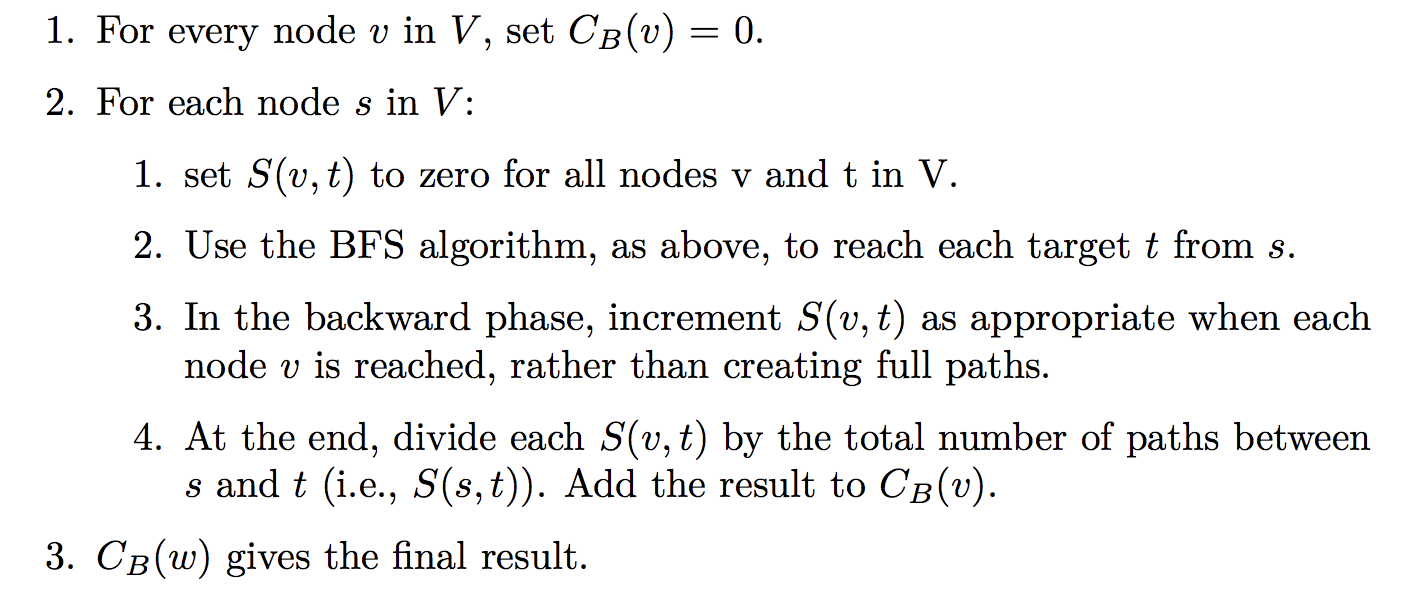
* <https://people.csail.mit.edu/jshun/6886-s18/papers/BrandesBC.pdf>
* <http://www.cl.cam.ac.uk/teaching/1617/MLRD/handbook/brandes.pdf> (Pseudo code below)

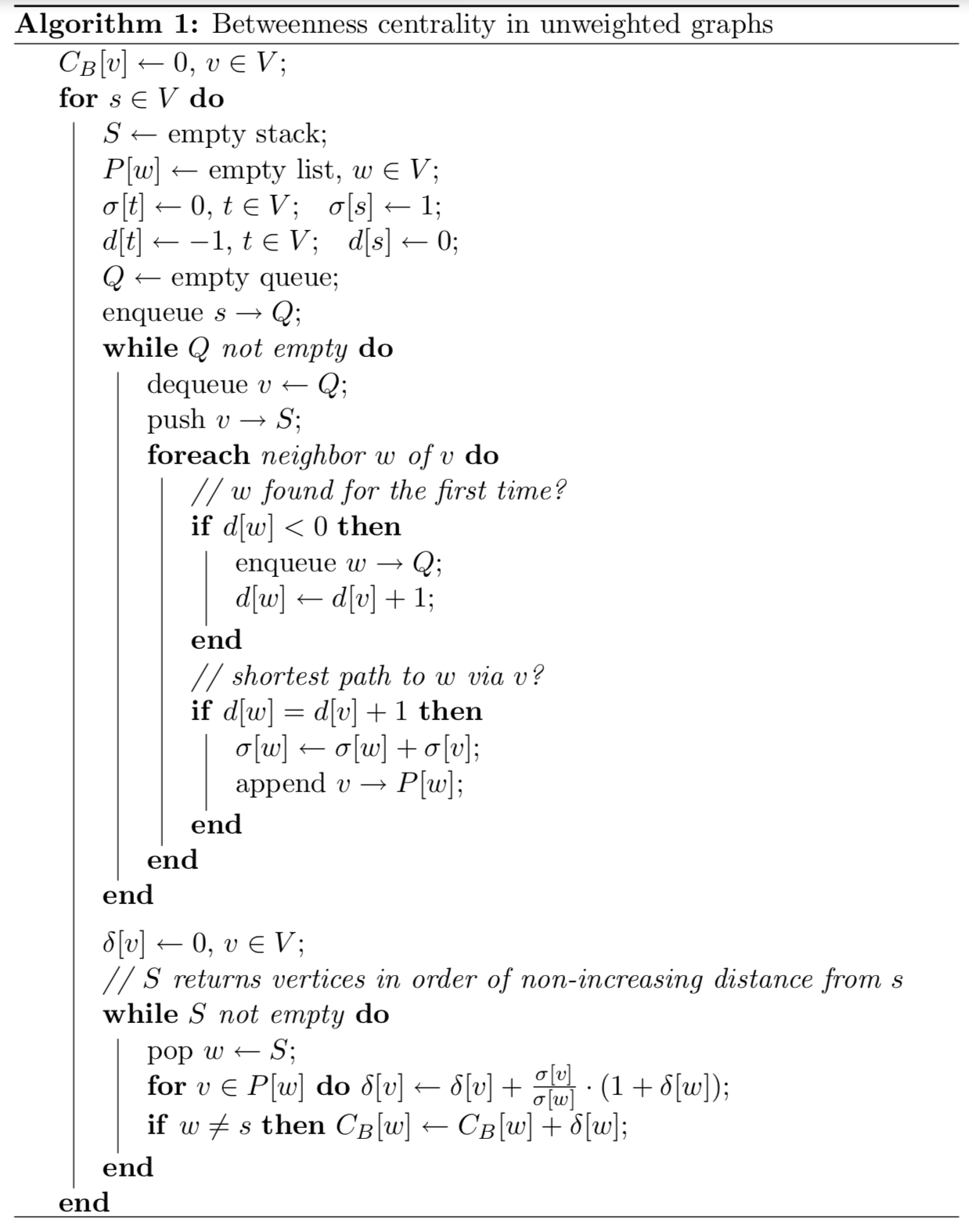
Every other algorithm apart from Brandes requires a time complexity of O(n^3) or above.

We can see this Brandes algorithm is not a power-based algorithm because we can each vertex exactly once, and the neighbours of each vertex only once.

**BFS Algorithm Used in Brandes (Pseudo code Explained):**



**BRANDES ALGOTRITHM COMBINED**:



**Data Structures:**

Due to the specifications of the academic paper, I must use the **data structures** provided, such as **Stacks, Queues, Array Lists, 2D arrays** and **Iterators**.

**Time Complexity – Brandes Betweenness:**

O(nm)

n = vertex

m = edges

The first for loop of this program scans the index of each node, thus adding O(n) time to the code:

for(int startingNode = 0; startingNode < numNodes; startingNode++)

{

// rest of program here

}

The second for loop of our code sets up the data structures to use in this algorithm. The second for loop does not search any array, but it does assign values.

As you remember from earlier, assigning a value to an array uses only one execution cycle, so it has little impact on the time complexity of this code.

From now on, we will disregard for and while loops in our code as having any impact, if they just contain assignment statements.

for (int i = 0; i<numNodes; i++)

{

paths[i] = new ArrayList ();

sigma[i] = 0;

distances[i] = -1;

}

The other biggest time impact is the following code: It adds O(e) time to the program, because it scans all the neighbours. This is proportional to the amount of edges there is for the neighbours.

for(int currentNeighbor = 0; currentNeighbor< numNodes; currentNeighbor++)

{

if(edgeMatrix[v][currentNeighbor] == 1)

{

if(distances[v]<0)

{

queue.add(currentNeighbor);

distances[currentNeighbor] = distances[v]+1;

}

if(distances[currentNeighbor] == distances[v]+1)

{

sigma[currentNeighbor] += sigma[v];

paths[currentNeighbor]. add(v);

}

}

}