CITS2200 Centrality Project

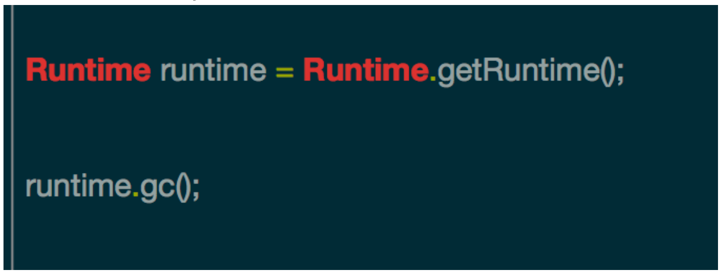
Nicholas Walters, 22243339
James Caldon, 22226341

In this Paper we will discuss our submitted code for the CITS2200 centrality project. We will explain the code in-depth and explain how each centrality measure was calculated, and how much the cost (time complexity of each measure). We have used one algorithm (brandes) to calculate all centrality measures, so we will separate each component in this report for better understanding

Performance Study

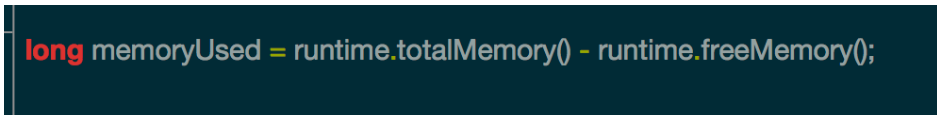
For this analysis of the code, we have measured the time used for various sizes on inputs. Also, the method to measure the memory used is included in the java runtime environment.

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



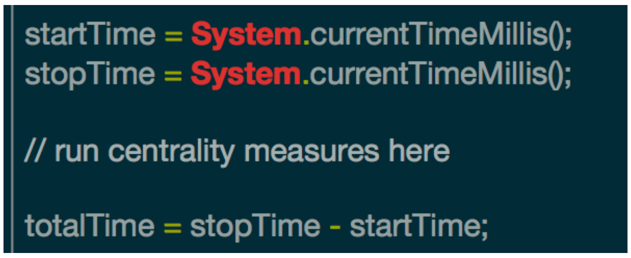
```
Runtime runtime = Runtime.getRuntime();


runtime.gc();
```

Then we can calculate the total memory used.



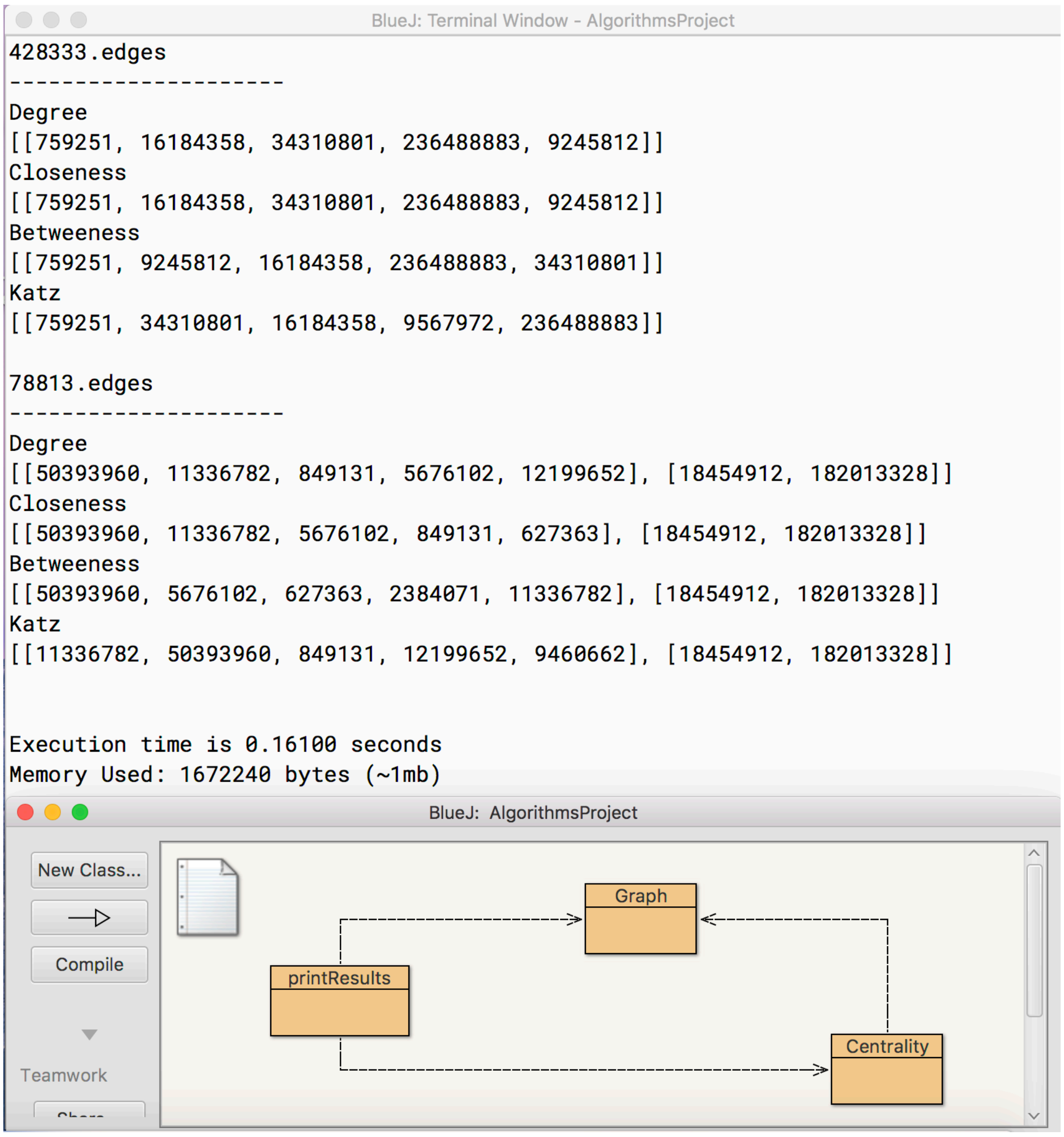
```
long memoryUsed = runtime.totalMemory() - runtime.freeMemory();
```

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



```
startTime = System.currentTimeMillis();
stopTime = System.currentTimeMillis();

// run centrality measures here

totalTime = stopTime - startTime;
```

OUTPUT:



```
428333.edges
---------------------
Degree
[[759251, 16184358, 34310801, 236488883, 9245812]]
Closeness
[[759251, 16184358, 34310801, 236488883, 9245812]]
Betweeness
[[759251, 9245812, 16184358, 236488883, 34310801]]
Katz
[[759251, 34310801, 16184358, 9567972, 236488883]]

78813.edges
---------------------
Degree
[[50393960, 11336782, 849131, 5676102, 12199652], [18454912, 182013328]]
Closeness
[[50393960, 11336782, 5676102, 849131, 627363], [18454912, 182013328]]
Betweeness
[[50393960, 5676102, 627363, 2384071, 11336782], [18454912, 182013328]]
Katz
[[11336782, 50393960, 849131, 12199652, 9460662], [18454912, 182013328]]


Execution time is 0.16100 seconds
Memory Used: 1672240 bytes (~1mb)
```

The following was compiled on a 2017 MacBook Pro

Explaining the Program and Output:

The following program makes use of the two provided twitter datasets. These text files are analysed, and outputs are printed respectively. Looking at the output of “428333.edges” its centrality measures are printed out. **There is one array holding all components of the graph**. Thus “428333.edges” is a connected graph. If the provided graph is an unconnected graph (has subgraph), then the output has more than one component.
If the graph has more than one component, there will be another array inside of the final array:

Centrality Measure:
[[component 1 outputs] [component 2 outputs]]

You can see this implementation on “78813.edges” as it has more than one component. But the number of nodes of the second component is not equal to 5, there were only 2 nodes printed in the second component array (Nodes 1845912 and 182013328 belong to the second component only). Execution time is also calculated for the whole program, to give idea of how efficient code is.

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)

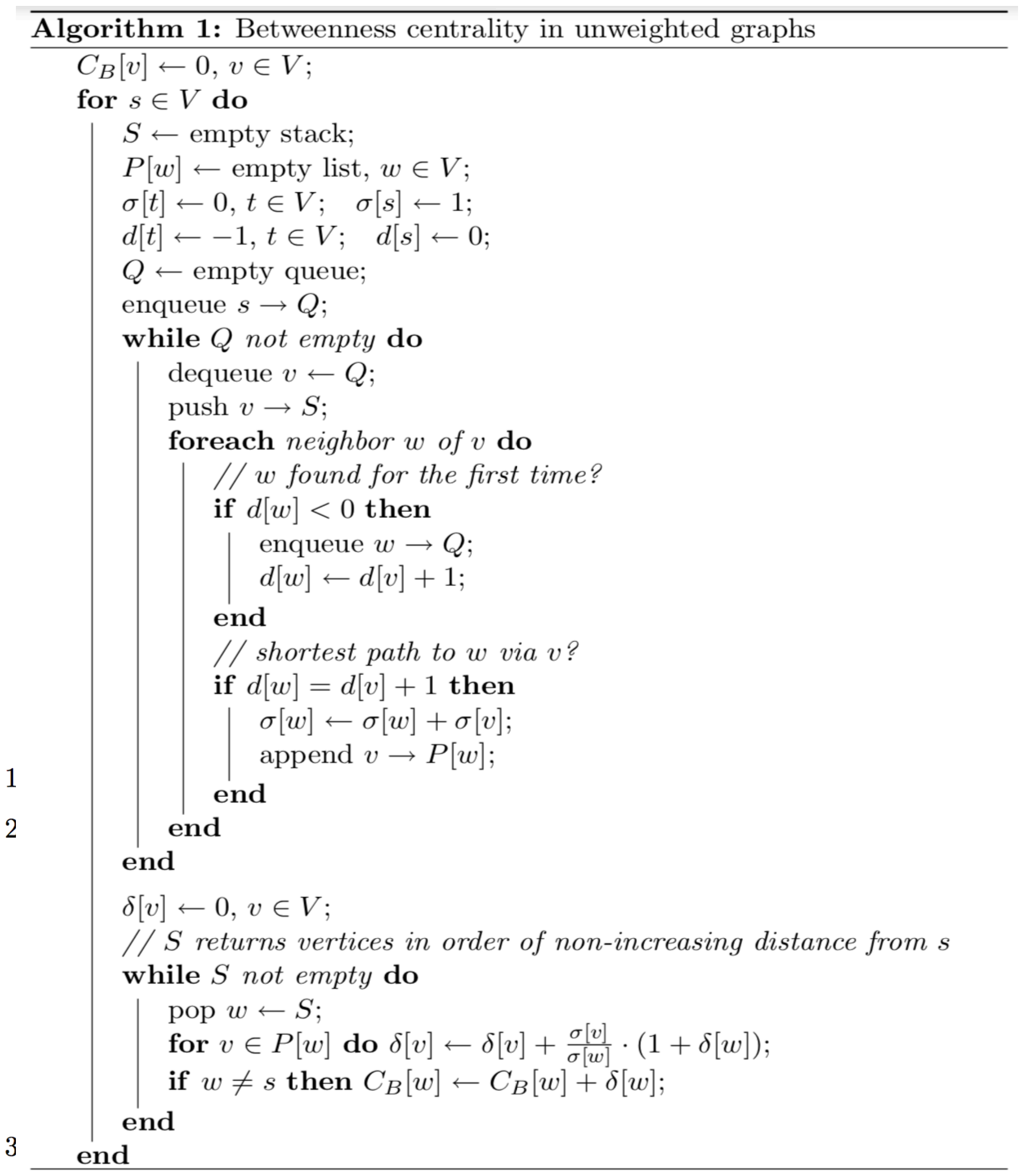
A Graph class was created to go through the edge file and create an adjList, which is the graph representation of the read in file. We have not included this in our complexity analysis for each algorithm.

Brandes algorithm was the main algorithm used in this program, to calculate all centrality measures. Brandes algorithm is the most efficient algorithm for calculating Betweenness centrality, requiring O(nm) time.

Not only has it been used for calculating Betweenness centrality, it has also been used to extract all the nodes for closeness, degree and Katz and calculation much more efficient, requiring a centrality measure to only be executed once to gain all values (one cycle).

Every other algorithm apart from Brandes requires a time complexity of O(n^3) or above (much less efficient).
We can see this Brandes algorithm is not a power-based algorithm because we can use each vertex exactly once, and the neighbours of each vertex only once.

Brandes algorithm implements a **breadth first search** and scans through all the nodes in a graph. In the process of breadth first search, the program implements many data structures such as **Queues, Array Lists, Hash Sets, Stacks and Priority Queues.** Where Hash Sets are used to store the adjacency matrix information, Priority Queues are used to store ALL of the nodes (and associated value by priority), Queues are used by the BFS for scanning each neighbour of the current node incrementally and Stacks are used to calculate the Betweenness centralities of each node incrementally.



```
Algorithm 1: Betweenness centrality in unweighted graphs
    $C_B[v] \leftarrow 0, v \in V$;
    for $s \in V$ do
        $S \leftarrow$ empty stack;
        $P[w] \leftarrow$ empty list, $w \in V$;
        $\sigma[t] \leftarrow 0, t \in V$;   $\sigma[s] \leftarrow 1$;
        $d[t] \leftarrow -1, t \in V$;   $d[s] \leftarrow 0$;
        $Q \leftarrow$ empty queue;
        enqueue $s \rightarrow Q$;
        while $Q$ not empty do
            dequeue $v \leftarrow Q$;
            push $v \rightarrow S$;
            foreach neighbor $w$ of $v$ do
                // $w$ found for the first time?
                if $d[w] < 0$ then
                    enqueue $w \rightarrow Q$;
                    $d[w] \leftarrow d[v] + 1$;
                end
                // shortest path to $w$ via $v$?
                if $d[w] = d[v] + 1$ then
                    $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$;
                    append $v \rightarrow P[w]$;
                end
            end
        end
        $\delta[v] \leftarrow 0, v \in V$;
        // $S$ returns vertices in order of non-increasing distance from $s$
        while $S$ not empty do
            pop $w \leftarrow S$;
            for $v \in P[w]$ do $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w])$;
            if $w \neq s$ then $C_B[w] \leftarrow C_B[w] + \delta[w]$;
        end
    end
```

findTopFive ()

the Find Top Five method is one of the most important methods in the program. This runs in O(5) time, as its proportional to the size of the priority queue which is 5 (the top five) As-well as iterating through the priority queue for the nodes, the method will also create a final output array (of size 5) for the specified centrality measure (aL = final array).

Parameter: pqList,   (Type: PriorityQueue<Node>)
takes in the brandes calculated priority queue.

Parameter: aL,   (Type: ArrayList)
takes in an array of a specified centrality measure. Whether it would be a degreeCentrality array, closenessCentrality array or katzCentrality array (etc.)

Parameter: g,   (Type: Graph)
The graph being used, whether it would be from any dataset.

Algorithm Analysis

Degree Centrality

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

```
for (int i = 0; i < adjList.size(); i++) {
                              pqList.get(component[i]).add(new Node(i, (float)
adjList.get(i).size()));
                    }

          findTopFive(pqList, degreeCentralities, g);
```

Degree centrality is the easiest to compute. After implementing brandes algorithm and adding all the nodes to a Priority Queue List, we are able to get the number of nodes from the priority queue, and its associated values. This is compacted into one small loop.

Time complexity

$$O(n)$$

as expected of a method like this, this will run in time proportional to input
O(n) time complexity

Closeness Centrality

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.

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

O(n^2)

With a large enough input, time is O(n^2). For smaller inputs (n^2 + n)

For ALL centrality measures 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 a data structure such as a list.
- Assigning a new value to a data structure
- Arithmetic operations

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

+O(n^2) or even +O(n*m)
Scan each neighbour and calculate Betweenness and closeness respectively.

```
While(!queue.isEmpty()) {

// scan each neighbour of current node, and update stack
// this will calculate both betweeness and katz

}
```

```
for (Integer currentNeighbor: adjacent) {

            // scan each neighbour of the current node

}
// increment add a new distance to the closeness centrality O(n)
closeness[startingNode] += distances[v];
```

+O(2n) – This iterates over all the nodes in the respective centrality arrays and adds them to a priority queue that top five uses, after brandes is executed and gets the associated values for the specified centrality measure.

```
        for (int k = 0; k < numNodes; k++) {
          pqList.get(component[k]).add(new Node(k, closeness[k]));
                    }

        findTopFive(pqList, closenessCentralities, g);
                    }
```

Data Structures:

Priority Queue

This allows us to get the nodes with the greatest closeness centrality, in an ordered fashion (descending)

Betweenness Centrality

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.

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);
                 }
              }
```

Katz Centrality

O(n^2)

With a large enough input, time is O(n^2). For smaller inputs (n^2 + 2n)

Like closeness centrality, Katz is also calculated within one execution of brandes algorithm.

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

            // rest of program here

}
```

Katz centrality is calculated within the induction of brandes Betweenness, so it runs in almost the same time as Betweenness. But katz has extra steps and loops involved, to calculate Katz weights. Where brandes scans all the nodes using a BFS, and their neighbours, this is done in one process making it more efficient. The loop above, and below combined makes Katz a nested for loop. Which individually calculates the node weights.

+O(n) where Katz has its own loop within brandes algorithm, proportional to the number of nodes.

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


               if (visited[i]) {
                         catz[startingNode] = catz[startingNode] + (float)
(adjList.get(i).size()*Math.pow(alpha,weightsOfShortestPaths[startingNode][i]));
               }

          }
```

+O(2n) where Katz is proportional to the number of nodes, and findTopFive

```
        for (int i = 0; i < numNodes; i++) {
                              pqList.get(component[i]).add(new
Node(i, catz[i]));
                         }

        findTopFive(pqList, katzCentralities, g);
```