# Programming Project 2018
# CITS2200 – Data Structures and Algorithms

Bruce How (22242664) & Haolin Wu (21706137)

# 1 Introduction

Centrality is an important measure of the global influence of a vertex in a graph. This measure is used extensively in large social network graphs, often for information diffusion and marketing. Many different measures of centrality exist. The four explored in our project are Degree, Closeness, Betweenness and Katz.

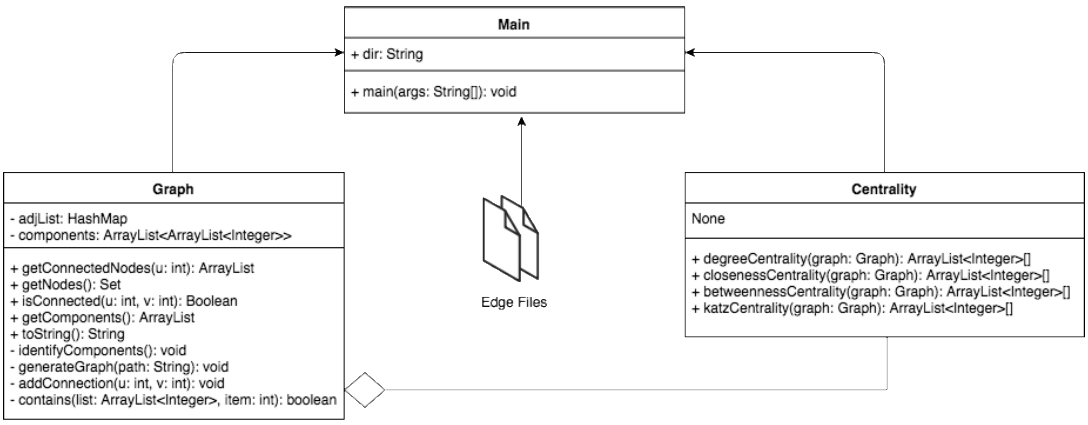


**Figure 1: UML diagram of overall project**

Figure 1 depicts a general UML diagram of our entire process. `main` classes are usually not shown, however for the purpose of clarity, it is represented as a class in the diagram. The entire project is based off three classes, `graph`, `centrality` and `main`.

# 2 Algorithms

## 2.1 Breadth First Search

In our project, the Breadth First Search (BFS) algorithm is used heavily in both classes. It is used in the `identifyComponents()` method in the `Graph` class, and the Closeness and Katz centralities in the `Centrality` class.

The BFS algorithm begins at an arbitrary node in a graph and explores neighbor nodes first before moving to the next level. The BFS algorithm uses a queue data structure due to their First In First Out (FIFO) nature. At every iteration, new unvisited neighbor nodes are added to the queue. The result produces a level by level search.

The reason why BFS is the primary algorithm chosen is due to the unweighted and undirected nature of the Graphs that will be used in this project.

The complexity of a Breadth First Search is represented as O(V + E) where V represents the number of vertices and E represents the number of edges in a graph.

## 2.2 Binary Search

Binary search is a fast search algorithm that searches a sorted collection for a particular item. In this project, it has been implemented in the `contains` method of the Graph class.

Binary search searches for a particular item by comparing the middle most item of the collection to the search item. A Boolean result is returned stating whether or not the collection contains a particular item. If the middle item is smaller, the search interval is narrowed to the lower half of the collection. Otherwise, the interval is narrowed to the upper half. This process is repeated until either the item is found, or the interval is empty.

The complexity of a Binary Search is O(n log(n))

## 2.3 Brandes Algorithm

For node $s \in V$, Brandes Algorithm requires the shortest path from s to every other node $t \in V$. These paths are stored for each pair `s,t` and is achieved by performing a Breadth First Search.

In Brandes Algorithm, for a certain node v, the ratio of shortest paths between s and t that go through v and the total number of shortest paths between s and t is called the pair-wise dependency:

$$\delta(s,t|v) = \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

Therefore:

$$C_B(v) = \sum_{s,t \in V} \delta(s,t|v)$$

Brandes Algorithm for a non-tree case uses an algorithm dubbed 'Ultimate MAGIC'. When there are alternative shortest paths that bypass v, the situation becomes more complex. A proportion of these shortest paths to nodes go through v, but a proportion doesn't. Ultimate MAGIC determines this ratio using a mathematical algorithm that is further explained in Appendix 1.

In terms of the code, the dependency of each source node of the BFS is different, and the betweenness centrality of the node is calculated from the summation of all dependency values, also known as the dependency accumulation. The dependency is described as:

$$\delta(v) = \sum_{w} \frac{\sigma_{s,v}}{\sigma_{s,w}}(1 + \delta(w))$$

where $\sigma$ represents the number of shortest paths between two nodes.

# 3 Class and Centrality Implementation

## 3.1 Graph Class

The Graph class contains methods with similar functionalities to one present in the CITS2200 Graph interface, however also has key differences. The primary storage of nodes and connected nodes resides in a HashMap `adjList`, where the keys are Nodes represented by the Integers, and its corresponding values represented by an ArrayList of integers, each integer inside the list being a connected node. This allows for quick access to the connected nodes of every node present in the graph. Furthermore, a key feature of the graph is to accommodate components. This is important because a graph can have multiple components, and finding the centrality will have to adapt to these components accordingly. For this reason, an ArrayList `components` is created to store of ArrayLists nodes for each component. Both these variables are private to maintain the security and integrity of the project.

A Graph class is required to have many methods such as `getNodes()`, `isConnected()`, `getComponents()` and `toString()`. These are basic methods and are explained in the JavaDoc. The unique methods that Graph implements are `identifyComponents()`, `generateGraph()`, `addConnection()` and contains.

### 3.1.1 identifyComponents

`identifyComponents()` runs a Breadth First Search on a graph to identify multiple components in a graph. Each component is represented by an ArrayList of integers indicating nodes, and these components are stored in another ArrayList containing each component.

The way this is achieved is by keeping track of the overall amount of nodes, and how many nodes have been visited. As a node is discovered, it is added to a queue. The first in-first out (FIFO) nature of a queue allows for a layer by layer search of the component. If the queue is empty, indicating all nodes in

the current component have been explored, and the amount of visited nodes are not equal to the total amount of nodes in the graph, it can be inferred that there is one or more additional components to the graph.

In the case where multiple components are present, an unvisited node is chosen and the process is repeated. The process runs until all components are searched, and all nodes are visited. The complexity of this process can be displayed in Big O notation (see section 4) as $O(V + E)$.

### 3.1.2 addConnection

`addConnection(int u, int v)` adds a connection between nodes u and v with reference to the `adjList` HashMap. If `adjList` does not contain a key u, a new ArrayList is created and v is added to it to represent the value for key u in the HashMap. However, if the key already exists, the node v is then added to the existing ArrayList value of corresponding key u. The ArrayList is then sorted. This is required to perform the binary search that is present in `contains()`.

### 3.1.3 generateGraph

`generateGraph()` generates a graph from from a given path to a list of edges.

```
11336782 3594701
782329 849131
627363 2195241
```

Figure 2: Example of file structure

As seen from the above figure, the general structure of a file is a line represents each connection. As the nodes are separated by a space, a split is made across this space to read each node and `addConnection()` is then used to add them to `adjList`. An important thing to note is that the edges are added twice, as they are mutual in an undirected graph.

### 3.1.4 contains

`contains()` performs a binary search for an item in a given ArrayList. As the ArrayList is previously sorted in `addConnection()`, a binary search can be implemented. As explained in 2.2, binary searches are very efficient in terms of complexity, and acts as a helper method in Graph.

## 3.2 Degree Centrality

To calculate Degree Centrality, each component of the graph must be analyzed. A for loop iterates through each node in each component. At every iteration, the number of incident nodes (`graph.getConnectedNode(node).size()`) is added to the ArrayList of node objects, `centralityValue`.

Once the iteration has complete, the nodes are sorted according to degree centrality using a private subclass `Node`, which implements the Comparable interface. The top 5 integer nodes for each component(s) with the highest degree centrality are added appropiately to the ArrayList of ArrayList integers, `degree`.

The return type is an ArrayList of ArrayList integers (`ArrayList<ArrayList<Integer>>`). Each index of the ArrayList contains an ArrayList of the top 5 integers, which represents the nodes, that have the highest degree centrality for each component.

## 3.3 Closeness Centrality

Closeness centrality is calculated by analyzing each component of the graph. A Breath First Search is used to calculate it due to the nature of the graph being unweighted and undirected. This means that at every node level, if a new node is found, it is by default the shortest path from the source node.

To implement the Breath First Search, a HashMap `distance` is used. The HashMap contains integer keys representing the node, and integer values representing the distance of the key from the source node.

The BFS is implemented over every node in each component of the graph. The source is initially added to the distance HashMap with the value 0. To keep track of the current location of the search, a variable `current` is defined, which begins at the source.

The shortest path distance is then calculated from the source to each node, and added to the `centralityValue` ArrayList. From there, the nodes are sorted according to closeness centrality, and top 5 integers, which represents the nodes, is then returned. In a similar manner to Degree Centrality, the sorting process is done using the same private subclass, `Node`, which implements the Comparable interface.

## 3.4 Betweenness Centrality

Betweenness centrality is calculated by finding the nodes with the highest betweenness centrality for each component of the graph. As defined in the project guidelines, this is the node which passes through the most shortest paths in a graph.

This implementation of betweenness centrality is implemented using Brandes algorithm and a Breadth First Search. Firstly, a for loop is written to ensure the algorithm iterates over all components of the graph. A HashMap centrality stores the

betweeness Centrality value for each node. A for each loop then uses a Breadth First Search to find the shortest distance to all other nodes, the preceding nodes that pass within all of the shortest paths, and the number of shortest paths from the source node.

### 3.5 Katz Centrality

Katz centrality is calculated by analyzing each component of the graph. In a similar manner to Closeness Centrality, a Breath First Search is used to calculate the distance or level, of each node to every other node in a graph.

To implement the Breath First Search, a HashMap `level` is used. The HashMap contains integer keys representing the node, and integer values representing the level of the key from the source node. Note that this HashMap will serve the same functionality as the `distance` HashMap used in the Closeness Centrality.

The BFS is implemented over every node in each component of the graph. The source is initially added to the `level` HashMap with the value 0. To keep track of the current location of the search, a variable `current` is defined, which begins at the source.

Each node is given a theoretical weight which can calculated by $\alpha^k$ where $\alpha$ is an attenuation factor between 0 and 1, and k is the level of the node from a given source node. The total summation of each theoretical weight of every node is then added to the `centralityValue` ArrayList. From there, the nodes are sorted according to its Katz centrality, and top 5 integers, which represents the nodes, is then returned. In a similar manner to Betweenness Centrality, the sorting process is done using the same private subclass, `Node`, which implements the Comparable interface.

## 4 Complexity

Complexity for this project is represented using Big O notation, where V represents the number of nodes, and E representing the number of edges.

### 4.1 Degree Centrality

The Degree centrality iterates through each node and checks for the amount of incident edges at each iteration. This complexity can be described as $O(V)$

### 4.2 Closeness Centrality

The Closeness centrality iterates through each node where at every iteration, p a Breadth First Search is performed. The overall complexity is $O(V * (V + E))$ which equates to $O(V^2 + VE)$.

For a sparse graph, E tends to V, so the complexity does not change. However, for a dense graph, the number of edges E is much more significant in terms of size, meaning the complexity becomes $O(VE)$

### 4.3 Betweenness Centrality

The Betweenness centrality iterates through each node where at every iteration, it performs a Breadth First Search. It then runs the dependency accumulation algorithm which iterates through a stack taking $O(V + E)$ time. For each node it will take $O((V + E) + (V + E))$. Therefore the total time complexity is $O(V(2V + 2E))$ which equates to $O(V^2 + VE)$.

As mentioned previously, for a sparse graph, the complexity does not change. However, for a dense graph, the complexity simplifies to $O(VE)$.

### 4.4 Katz Centrality

Katz centrality iterates through each node where at every iteration it performs a Breadth First Search.

This takes $O(V * (V + E))$ time which equates to $O(V^2 + VE)$

As stated before, for a sparse graph, the complexity does not change, but for a dense graph the complexity simplifies to $O(VE)$.

## 5 Execution

### 5.1 Preparing & Compiling

The first step for a proper execution of the program is to ensure that all the required files are in the same directory. These include the three provided java files, `Main.java`, `Centrality.java` and `Graph.java`, as well as any edgelist files that you may have.

Before the program can be executed, the `Main.java` file will need to be compiled to a bytecode class file. To do so, ensure that you are in the directory which contains all the provided files and run the following command.

```
javac Main.java
```

### 5.2 Project Execution

Once the file has been compiled, execute the java file with the following command,

```
java Main filepath alpha
```

The command takes twos parameters, a required `filepath`, and an optional `alpha` value. Replace `filepath` with the file path of edges file including any file extensions. In a similar way, replace `alpha` with the desired alpha value which will be used to compute Katz's centrality. This parameter is optional and will use a default value of $\alpha = 0.5$, if not defined.

The output of the execution should return the degree, closeness, betweenness and katz centres for each component of the graph.

## 6 Conclusion

The overall process of implementing centralities allowed a wider view into the importance of using centralities to identify important nodes, regardless of method used. By using Java 8, Degree, Closeness, Betweenness and Katz centralities were successfully implemented, using efficient complexity algorithms. The project also executes correctly and efficiently on the command line, and in addition to that, uses external algorithms such as Brandes Algorithm to compute the Betweeness centrality with a better complexity than suggested in the project brief.

## 7 References


Brandes Algorithm – University of Cambridge, 2008
http://www.cl.cam.ac.uk/teaching/1617/MLRD/handbook/brandes.pdf

Ultimate MAGIC – University of Konstanz, Ulrick Brandes, 2001
http://www.algo.uni-konstanz.de/publications/b-fabc-01.pdf

CITS2200 Project Documentation – University of Western Australia, 2018
http://teaching.csse.uwa.edu.au/units/CITS2200/Labs/project-2018/project-2018.html