**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

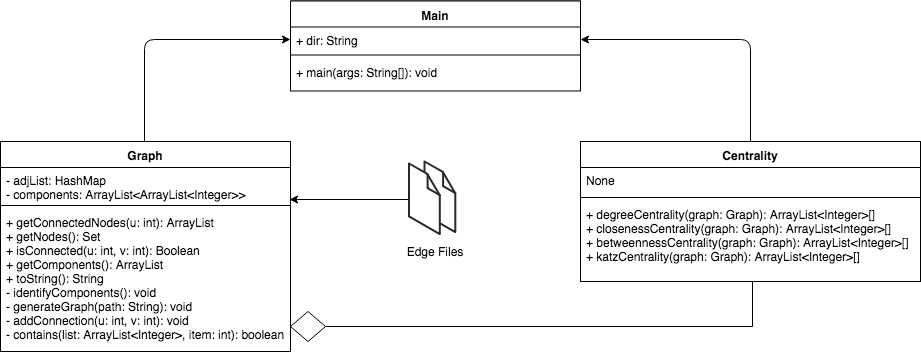


Figure 1 depicts a general UML diagram of our entire process. main classes are usually not shown, however for the purpose of clarity, it is represented as a class in the diagram. The entire project is based off three classes, graph, centrality and main.

­­­

**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.

A BFS begins at an arbitrary node in a graph, and explores neighbor nodes first, before moving to the next level. BFS uses a queue, as queue is First In First Out (FIFO) and checks whether a vertex has been visited, and if it hasn’t, it adds it to the queue.

The reason why BFS is the primary algorithm chosen is due to the unweighted and undirected nature of the Graphs presented by the project.

In general, 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 is used specifically 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. An Boolean result is returned if a collection contains a particular item. If the middle item is smaller, narrow the search interval to the lower half of the collection. Otherwise, narrow the interval to the upper half. Repeat this process until the value is found, or the interval is empty.

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

**2.3 Brandes Algorithm**

For node , Brandes Algorithm requires the shortest path from s to every other node . 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:

Therefore:

Brandes Algorithm for a non-tree case uses an algorithm dubbed ‘Ultimate MAGIC’. When there is 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:

**3 Class and Centrality Implementation**

**3.1 Graph Class**

The Graph class written contains methods 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 represented by the nodes, and the corresponding values represented by an ArrayList of integers, each integer inside the list being a connected node. This allows quick access to all nodes 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 ArrayLists of 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:

**3.1.2 addConnection**

addConnection(int u, int v) adds connections between nodes u and v with reference to the adjList HashMap. If adjList already contains 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, it is added into a connected ArrayList, and then the collection is sorted. This is to perform the binary search that is present in contains(). The node v is then added to the existing ArrayList value of corresponding key u.

**3.1.3 generateGraph**

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

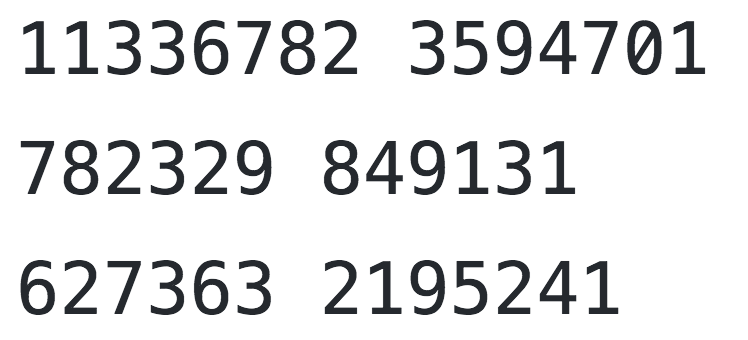


Figure 2: Example of file structure

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 in and addConnection() is used to add them to adjList. An important thing to note is that the edges are added twice, as they are mutual.

**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 compared with the current highest value highest and updated appropriately when necessary.

When the amount of incident nodes for a particular node is greater than highest, the old ArrayList is cleared, and a new list is created with this node.

If the amount of incident nodes is the same as highest and is not already a part of the list, it is added to the list, provided the list does not already have 5 nodes as specified in the project.

The return type is an array of ArrayList integers (ArrayList<Integer>[]). Each index of the array contains an ArrayList of integers representing the nodes, which have the highest degree centrality for each component.

**3.3 Closeness Centrality**

Closeness centrality is calculated by analyzing each component of the graph. A binary 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, by using getComponents(). 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 closeness is then returned.

**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.

Brandes Algorithm’s dependency accumulation algorithm is then run to compute the betweenness centrality for each node, and due to the undirected nature of the graph, the result is halved.

The centrality for each node is then stored as a Node object, and sorted based off centrality. These results are added into the betweenness ArrayList and returned.

**3.5 Katz Centrality**

Text here

End of text here

**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 connected nodes at each node. This complexity can be described as:

**4.2 Closeness Centrality**

The Closeness centrality iterates through each node and at each iteration performs a Breadth First Search.

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 dominant meaning the complexity becomes:

**4.3 Betweenness Centrality**

The Betweenness centrality iterates through each node and at each iteration performs a Breadth First Search. It then runs the dependency accumulation algorithm, which goes through a stack, taking time complexity.

For each node it will take:

Therefore the total complexity is:

As mentioned previously, for a sparse graph, the complexity does not change. However for a dense graph, the complexity simplifies to:

**4.4 Katz Centrality**

Katz centrality iterates through each node at each iteration and performs a Breadth First Search.

As stated before, for a sparse graph, the complexity does not change, but for a dense graph the complexity simplifies to:

**5 Execution**

(Project Deliverable 1 – main())

more text

End of text here

**6 Conclusion**

Text here

End of text here

**7 References**