CITS2200 PROJECT REPORT

Varun Jain (21963986)
Joshua Goh (21978677)

1 Introduction

The aim of this project is to explore centrality of vertices in a graph. In this project we are required to use algorithms to measure the global influence of a vertex in a graph. The four measures of centrality explored in this project are:

- Degree centrality
- Closeness Centrality
- Betweenness Centrality
- Katz Centrality

Twitter data can be represented as a graph and analysed with each of the four algorithms listed above to provide information about the data set. The graph of Twitter is unweighted meaning the weight or distance between adjacent vertices is always 1.

To be able to use the algorithms we must first map the data as a graph. The data set can be mapped as a graph by connecting vertices via an edge. Vertices are read two at a time where an edge is created if it does not already exist. After all the edges are made the data is stored as one graph which can be analysed using the four algorithms listed above.

1.1 How to run

1) cd to directory containing source files
2) Compile project by entering “javac centralityMain.java”
3) Run by entering “java centralityMain input.edges”
   Note: To input different files change “input” to file name and “.edges” to extension

2 Shortest Path

Shortest path is the path between a pair of vertices in a graph which has the smallest weight. The weight is calculated by summing up all the edge weights between the two vertices. In other words, the shortest path is the shortest distance between two vertices. Since we are dealing with unweighted graphs in this project, the shortest path will also be the path which uses the least number of vertices to get from start to finish.

2.1 Implementation

The algorithm most commonly used method to traverse a graph is the Breadth First Search (BFS) algorithm. We decided to implement the BFS algorithm to find the shortest paths. The BFS algorithm works by taking a starting vertex and traversing the graph until it reaches a destination vertex. This gives us the distance from one vertex to another using the least number of edges possible. Achieving the shortest path is made possible by storing the vertices adjacent to a vertex in a queue in the order they were visited. Vertexes that have not been visited are marked with a value of -1 so that the search knows what it has and hasn’t traversed.

The time complexity of BFS is $O(V^2)$. This is because we have a sparse graph represented by an adjacency matrix.

Below is a pseudo code example of how the BFS algorithm works:

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```

Algorithm for Bread First Search: (Cormen, Leiserson, Rivest, and Stein, 2009)

3 Degree Centrality

Degree centrality is measured as the number of edges connected to a vertex. Since we want the vertices with the highest degree centrality we would be looking for the vertices with the highest number of edges connected to them.

3.1 Implementation

The degree centrality algorithm works by first creating a 2D matrix with the same number of rows as columns. Each row represents a unique vertex listed in a specific order. Each column represents a unique vertex in the same order as the rows. For example, row 5 and column 5 represent the same vertex as do row 9 and column 9. Each pair of vertices in the data set must be connected via an edge and to represent this, a 1 is stored in the corresponding row and column. So, if vertex 3 and vertex 9 are connected then a 1 is stored in row 3 column 9. Once all the vertex pairs have been checked algorithm then computes the sum for each row. This sum is their degree centrality. Another 2D matrix is then created, this time storing the vertices in the first column and the corresponding degree centrality in the second column.

The function *degree_sort* takes this 2D matrix and sorts the vertices in order of degree centrality from highest to lowest. The top five entries are then printed out.

The time complexity of degree centrality is $O(V^2)$. This is because we are using nested for loops to calculate the centrality.

4 Closeness Centrality

The farness of a vertex *j* can be defined as the sum of all shortest paths to all other vertices from vertex *j*. Closeness centrality is simply the inverse of farness. We can use the BFS algorithm to calculate the distance from vertex *j* to every other vertex before simply adding them up and inversing the sum.

4.1 Implementation

The first step in computing closeness centrality is to run the BFS algorithm for our starting vertex to calculate the shortest path to every other vertex. The BFS algorithm can follows the same implementation as described previously in 2.1. All the distances are then stored in an array called *distance* where each index corresponds to a unique vertex.

Our function *distance* takes the array *distance* and loops through every index. If the value in the index is greater than 0 then the value is added to integer sum. At the end of the loop the value for sum is returned which holds the farness of the starting vertex.

The function *closeness* creates a 2D matrix *store*. It contains a loop which stores vertices in the first column and their corresponding closeness centrality in the second column. This loop goes through every vertex in the data set before finally calling the function *closeness_sort*. The function *closeness_sort* takes the 2D matrix *store* and sorts the vertices in order of closeness centrality from highest to lowest. The top five closeness centralities are then printed out.

The time complexity for our closeness centrality implementation is $O(V^2)$. This is because we use BFS on an adjacency matrix, so the complexity is the same as our shortest path algorithm.

5 Betweenness Centrality

Betweenness centrality measures the importance of a vertex based on how many shortest paths pass through it. In order to obtain the betweenness centrality of each vertex we must first be able to find the shortest paths between every vertex pair in the graph. After finding all the shortest paths we are then able to loop through all the paths and sum the number of times a vertex appears in a shortest path for every vertex.

5.1 Implementation

The computation of betweenness centrality relies on using a shortest path algorithm. We decided to use the BFS algorithm in order to calculate the shortest paths. The BFS algorithm used in our function *betweenness* uses the same BFS algorithm described previously in 2.1.

Now that we have all the shortest paths stored in an array, all we have left is to determine how many times all the shortest paths pass through every vertex. This is done in the while loop pictured below:

```
while(!stack.isEmpty())
{
    //pop the topmost vertix from the stack
    currentVertex = stack.pop();
    Iterator<Integer> pathIterator = nodeList[currentVertex].iterator();

    int path;
    //loop until the end of iterator
      while(pathIterator.hasNext())
       {
           path = pathIterator.next();
           delta[path] = delta[path] + (sigma[path]/sigma[currentVertex])*(delta[currentVertex]+1);
       }
      //check wheather currentVertex is not equal to start vertex
      if(currentVertex != s)
      {
          betweenessCentrality[currentVertex] = betweenessCentrality[currentVertex] + delta[currentVertex];
      }
}
```

This takes each vertex in the data set and analyses them one at a time. To calculate a vertex's betweenness centrality, the formula is applied and result is stored in array *betweenessCentrality* at index *currentVertex*. The function *betweenness_sort* takes the array *betweenessCentrality* and sorts the vertices in order of betweenness centrality from highest to lowest. The top five entries are then printed out.

The time complexity of betweenness centrality is O(VE).

Below is the pseudo code for the betweenness centrality algorithm we used:

Algorithm 1: Betweenness centrality in unweighted graphs

```
C_B[v] ← 0, v ∈ V;
for s ∈ V do
    S ← empty stack;
    P[w] ← empty list, w ∈ V;
    σ[t] ← 0, t ∈ V;   σ[s] ← 1;
    d[t] ← −1, t ∈ V;   d[s] ← 0;
    Q ← empty queue;
    enqueue s → Q;
    while Q not empty do
        dequeue v ← Q;
        push v → S;
        foreach neighbor w of v do
            // w found for the first time?
            if d[w] < 0 then
                enqueue w → Q;
                d[w] ← d[v] + 1;
            end
            // shortest path to w via v?
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                append v → P[w];
            end
        end
    end
    δ[v] ← 0, v ∈ V;
    // S returns vertices in order of non-increasing distance from s
    while S not empty do
        pop w ← S;
        for v ∈ P[w] do δ[v] ← δ[v] + (σ[v]/σ[w]) · (1 + δ[w]);
        if w ≠ s then C_B[w] ← C_B[w] + δ[w];
    end
end
```

Algorithm for Betweenness Centrality: (Brandes, 2001)

6 Katz Centrality

Katz centrality is essentially an extension of degree centrality. Instead of only measuring the number of edges immediately connected to a vertex, Katz centrality takes into consideration vertices that are further than one vertex away. It is defined by the formula:

$$Katz(i) = \sum_{k=1}^{\infty} \sum_{j=1}^{N} \alpha^k A^k$$

We define α to be whatever value we like. The power *k* of alpha represents how many edges away a vertex is from our starting vertex. If we have a vertex three edges away from our starting vertex then we would have α^3. This is done for all the other vertices in the graph until we have an α value for every vertex. The sum of all the α is the Katz centrality of a vertex.

6.1 Implementation

Our implementation of Katz centrality once again requires the shortest path algorithm BFS. The BFS algorithm used in our Katz centrality function is the same as the BFS algorithm described previously in 2.1.

The function takes a starting vertex then calculates its degree centrality. This is multiplied by the α^k value of all the other vertices. α^k is calculated using *Math.pow(0.5, distance[i])*. The value chosen for α is 0.5 and the power *k* is calculated with distance*[i]* which is the number of edges a vertex is from the starting vertex. This is all then stored in *sum* which is the final Katz centrality for the starting vertex. All the Katz centralities are stored in an array which is sorted using the method *closeness_sort* from highest to lowest. The top five entries are then printed out.

The time complexity for our Katz centrality implementation is $O(V^2)$. This is because we are using BFS which has the highest priority.

Note: To change the value of α, change "*0.5*" in *Math.pow(0.5, distance[i])* to the new value (found in the *Centrality.java* file).

7 Splitting Components

In order to split the components of the data sets up we implemented a depth first search (DFS) algorithm. A DFS is similar in many ways to a BFS but differs in that it traverses as far as possible down a branch before backtracking and starting on the next branch. DFS has a time complexity of O(V+E). We were able to split the graph for the disconnected edge file but we were unable to analyse them separately.

Code referenced from: https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/

8 Outputs

Our project outputs the name of the centrality followed by its top five centralities and its running time. This is done for all four centralities.

9 References

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2009). Introduction to algorithms (3rd ed., p. 595). Cambridge, Massachusetts: The MIT Press.

Brandes, U. (2001). A Faster Algorithm for Betweenness Centrality. Retrieved from http://www.algo.uni-konstanz.de/publications/b-fabc-01.pdf

Connected Components in an undirected graph - GeeksforGeeks. (2018). Retrieved from https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/