Programming Project II, First Report

Illya Starikov, Claire Trebing, Timothy Ott

Due Date: April 22, 2016

1 Abstract

Social Networks have revolutionized the way we communicate, meet others, consume information, and essentially influence our day-to-day lives. To show Social Network's prominence, here are the percentages of online adults who use social media:

Facebook : 71% Adults

Twitter : 23% Adults

Instagram : 26% Adults

Pinterest : 28% Adults

LinkedIn : 28% Adults

This is unprecedented. Seeing as an overwhelming majority of adults have some sort of social media account, this can be used to model real world relationships — through social graphs.

In this experiment we would like to examine the social graph through the follower-followee relationship.

2 Introduction and Motivation

As stated above, social networks play a dominant role in our lives. Through social graphs, we can examine real world relationships.

There are many reasons for this to be valuable, such as common interest, community detection, influence amongst followers, etc. For this experiment, we would just like to test on the following measures:

Degree Distribution Test the direct amount of followers compared to followees a person has.

Shortest Path Distribution Test the degree of separation between a follower-followee.

Graph Diameter Test the breadth of a community.

Closeness Centrality Test the dependence of a degree of separation on a singular person.

Betweenness Centrality Distribution Same as previous.

Community Detection Based on Closeness Centrality, find the diameter of a community.

This will give us a reasonable dataset for the relationship of a community in a social graph.

3 Proposed Solutions

3.1 Uweighted In Degree Distribution

Unweighted In Degree Distribution goes through each entry to determine what the origin of each edge is. An array with cells for each possible vertex (total) is created and once the origin is determined the respective cell in total is increased by one. After all of the entries have been checked, the function goes through the totals.

The functions begins by assuming `total[0]` is the maximum number of In Degree edges and 0 is added to `arrayMax`. `arrayMax` is the array which holds all of the vertexes with the maximum number of in degree edges. Each entry in total is compared to max. If the total entry is larger than the current max, then the current max is replaced with current total. The `arrayMax` list is cleared and the current total origin is added to the `arrayMax`. If the current total is equal to the max, then the origin of current total is added to `arrayMax`.

```
void Unweighted_InDegree_Distribution(&int arrayMax[])
{
    N = the number of vertices
    int * total;
    total = new int [maxVertice];

    for(i = 0 to N){
        total[origin of i]++;
    }
    int degreeMax = total[0];
    add 0 to arrayMax

    for(i = 1 to maxVertice){
        if(total[i] > degreeMax){
            degreeMax = total[i];
            clear arrayMax
            add i to arrayMax
```

```
        }
        if(total[i] == degreeMax){
            add i to arrayMax
        }
    }
    delete[] total;
    return;
}
```

The time complexity of this function is $2n = \mathcal{O}(n)$. The function must traverse through the array twice before finishing.

3.2 Unweighted Out Degree Distribution

Unweighted Out Degree Distribution goes through each entry to determine what the destination of each edge is. An array with cells for each possible vertex (total) is created and once the destination is determined the respective cell in total is increased by one. After all of the entries have been checked, the function goes through the totals. The functions begins by assuming `total[0]` is the maximum number of Out Degree edges and 0 is added to `arrayMax`. `arrayMax` is the array which holds all of the vertexes with the maximum number of out degree edges. Each entry in total is compared to max. If the total entry is larger than the current max, then the current max is replaced with current total. The `arrayMax` list is cleared and the current total destination is added to the `arrayMax`. If the current total is equal to the max, then the destination of current total is added to the `arrayMax`.

```
void Unweighted_OutDegree_Distribution(&int arrayMax[])
{
    N = the number of vertices
    int * total;
    total = new int [maxDestination];

    for(i = 0 to N){
        total[Destination of i]++;
    }
    int degreeMax = total[0];
    add 0 to arrayMax

    for(i = 1 to maxOrigin){
        if(total[i] > degreeMax){
            degreeMax = total[i];
            clear arrayMax
            add i to arrayMax
        }
        if(total[i] == degreeMax){
```

```
            add i to arrayMax
        }
    }
    delete[] total;
    return;
}
```

3.2.1 Complexity Analysis

The time complexity of this function is $2n = \mathcal{O}(n)$. The function must traverse through the array twice before finishing.

3.3 Weighted In Degree Distribution

Weighted In Degree Distribution goes through each entry to determine what the origin of each edge is and the sum of the weighted edges. An array with cells for each possible vertex (total) is created and once the origin is determined the respective cell in total is increased by weight of that edge. After all of the entries have been checked, the function goes through the totals.

The functions begins by assuming `total[0]` is the maximum number of Weighted In Degree edges and 0 is added to `arrayMax`. `arrayMax` is the array which holds all of the vertexes with the maximum number of weighted in degree edges. Each entry in total is compared to max. If the total entry is larger than the current max, then the current max is replaced with current total. The `arrayMax` list is cleared and the current total origin is added to the arrayMax. If the current total is equal to the max, then the origin of current total is added to the `arrayMax`.

```
void Weighted_InDegree_Diribution(&int arrayMax[])
{
    N = the number of vertices
    int * total;
    total = new int [maxOrigin];

    for(i = 0 to N){
        total[Origin of i]+= (Weight of i);
    }
    int degreeMax = total[0];
    add 0 to arrayMax

    for(i = 1 to maxOrigin){
        if(total[i] > degreeMax){
            degreeMax = total[i];
            clear arrayMax
            add i to arrayMax
        }
```

```
        if(total[i] == degreeMax){
            add i to arrayMax
        }
    }
    delete[] total;
    return;
}
```

3.3.1 Complexity Analysis

The time complexity of this function is $2n = \mathcal{O}(n)$. The function must traverse through the array twice before finishing.

3.4 Weighted Out Degree Distribution

Weighted Out Degree Distribution goes through each entry to determine what the destination of each edge is and the weight of those edges. An array with cells for each possible vertex (total) is created and once the destination is determined the respective cell in total is increased by the value of that edge. After all of the entries have been checked, the function goes through the totals. The functions begins by assuming `total[0]` is the maximum number of Weighted Out Degree edges and 0 is added to `arrayMax`. `arrayMax` is the array which holds all of the vertexes with the maximum number of weighted out degree edges. Each entry in total is compared to max. If the total entry is larger than the current max, then the current max is replaced with current total. The `arrayMax` list is cleared and the current total destination is added to the arrayMax. If the current total is equal to the max, then the destination of current total is added to the `arrayMax`.

```
void Weighted_OutDegree_Diribution(&int arrayMax[])
{
    N = the number of vertices
    int * total;
    total = new int [maxDestionation];

    for(i = 0 to N){
        total[Destination of i]+= (Weight of i);
    }
    int degreeMax = total[0];
    add 0 to arrayMax

    for(i = 1 to maxDestination){
        if(total[i] > degreeMax){
            degreeMax = total[i];
            clear arrayMax
            add i to arrayMax
```

```
        }
        if(total[i] == degreeMax){
            add i to arrayMax
        }
    }
    delete[] total;
    return;
}
```

The time complexity of this function is $2n = O(n)$. The function must traverse through the array twice before finishing.

3.5 Shortest Path

For our Shortest Path algorithm we decided to implement the Floyd-Warshall algorithm. We chose this algorithm because of the need to examine the shortest path for all pairs of vertices as well as the lower complexity of the dynamic programming solution. The Floyd-Warshall algorithm creates an array of VxV size keeping a running tally of the shortest path while adding verticies to the list of possible intermediate points on those paths.

```
V = the number of vertices
distance = new array[V][V]
Initialize distance to -1   //since there are no negative weights this will
                            //represent infinity
Floyd(V):
    for each vertex v:
        distance[v][v] = 0
    for each edge (u,v)
        distance[u][v] = w(u,v) //the weight of the edge (u,v)
    for k from 1 to V
        for i from 1 to V
            for j from 1 to V
              if distance[i][j] > distance[i][k] + distance[k][j]
                distance[i][j] = distance[i][k] + distance[k][j]

shortest = new array[100]
initialize shortest to 0
for i from 1 to V
    for j from 1 to V
        if distance[i][j] > 0
          if distance[i][j] > shortest.length
              temp = new array[distance[i][j]+10]
              for i from 0 to shortest.length
                  temp[i] = distance[i]
              delete distance
```

```
            distance = temp
          shortest[distance[i][j]]++

for i from 1 to shortest.length
    if shortest[i] > 0
        output shortest[i]
```

3.5.1 Complexity Analysis

As we know, the Floyd-Warshall algorithm operates on n^3 time. Combining this with the algorithm to iterate over the resulting V x V matrix results in a $n^2 + n^3$ complexity — or, $\mathcal{O}(n^3)$ time.

3.6 Unweighted Graph Diameter

Our solution first creats an $N \times N$ array to hold all of the shortest paths from each vertex to every other vertex. For an unweighted graph the shortest path from $i \rightarrow j$ is equal to the shortest path from $j \rightarrow i$. After creating and filling the matrix, the function assumes that the value at `distance[0][1]`. The function then tests every value in the matrix and compares it to the max. If the value is greater than the max then the max is replaced with `distance[i][j]`. Clear the `GraphDiameterList` and then add (i, j) to the graph diameter list. If `distance[i][j]` is equal to max then add (i, j) to the `GraphDiameterList`.

The pseudocode is as follows.

```
Unweighted Graph Diameter {
    N = the number of vertices
    distance = new array[N][N]
    all distances start at -1

    for(i = 0 to N){
        for(j = i+1 to N){
            distance[i][j] = shortest path(i, j)
        }
    }

    max = Diameter[0][1]
    Add (0,1) to GraphDiameterList

    for(i = 0 to N){
        for(j = 1 to N){
            if(distance[i][j] > max){
                clear GraphDiameterList
                max = distance[i][j]
                add (i,j) to GraphDiameterList
            }
```

```
            if(distance[i][j] = max){
                add (i,j) to GraphDiameterList
            }
    }
    return GraphDiameterList
}
```

3.6.1 Complexity Analysis

The time complexity of this code is $\mathcal{O}(n^2)$. You must go through the matrix twice to complete this function.

3.7 Weighted Graph Diameter

This functions first created an $N \times N$ array to hold all of the shortest paths from each vertex to every other vertex. After creating and filling the matrix, the solution assumes that the value at `distance[0][1]`. The function then tests every value in the matrix and compares it to the max. If the value is greater than the max then the max is replaced with `distance[i][j]`. Clear the `GraphDiameterList` and then add (i, j) to the graph diameter list. If `distance[i][j]` is equal to max then add (i, j) to the `GraphDiameterList`.

The pseudocode is as follows:

```
Weighted Graph Diameter{
    N = the number of vertices
    distance = new array[N][N]
    all distances start at -1

    for(i = 0 to N){
        for(j = 0 to N){
            if(i != j){
                distance[i][j] = shortest path(i, j)
            }
        }
    }

    max = Diameter[0][1]
    Add (0,1) to GraphDiameterList

    for(i = 0 to N){
        for(j = 0 to N){
            if(i != j)
            if(distance[i][j] > max){
                clear GraphDiameterList
                max = distance[i][j]
                add (i,j) to GraphDiameterList
```

```
            }
            if(distance[i][j] = max){
                add (i,j) to GraphDiameterList
            }
    }
    return GraphDiameterList
```

3.7.1 Complexity Analysis

The time complexity of this code is $\mathcal{O}(n^2)$. You must go through the matrix twice to complete this function.

3.8 Closeness Centrality

The Closeness Centrality is defined as $C_c(i) = [\sum_{j=1}^{N} d(i,j)]^{-1}$. So in order to find the Closeness Centrality we first must find the sum of all the shortest paths for each vertices. Once we find this sum we simply record it as a float by dividing one by the sum and storing it in an array of Closeness values.

```
sum = 0
closeness = new array[100]
initialize closeness to 0
for i from 0 to V
    for j from 1 to V
        sum += distance[i][j]
    closeness[i] = 1/sum
    output closeness[i]
    sum = 0
```

3.8.1 Complexity Analysis

Because we are iterating over the entire matrix resulting from the Floyd-Warshall algorithm of $V \times V$ size. We arrive at a complexity of $\mathcal{O}(n^2)$.

3.9 Undirected Betweenness Centrality Distribution

This algorithms takes a vertex i from the main function. The function then looks at every shortest path possible. Starting at vertex 0 the function and going to the largest vertex. If i is the starting point (j) or the ending point (k) it is ignored. K is defined as $j+1$ If the shortest path because in undirected graphs, the shortest distance from $j \to k ==$ the shortest distance from $k \to j$. If the shortest distance from j to k includes the vertex i, the value of the betweenness is increased by 1. The total betweens is then returned to the main function.

```
Undirected Betweenness Centrality Distribution of Vetex i(vertex i){
    N = max vertex
    B = betweenses = 0
```

```
    for(j = 0 to N){
        if(i != j){
            for(k = j+1 to N){
                if(i != k){
                    shortestpath between j and k
                    if i is included, increase betweeneses by 1
                }
            }
        }
    }

    return B;
}
```

This function has a complexity of n. Each vertex check every vertex larger than itself to determine all of the shortest paths. The first vertex must check nearly other ever vertex, except i. This gives it a complexity of $\mathcal{O}(n)$.

3.10 Directed Betweenness Centrality Distribution

This algorithm takes a vertex i from the main function. The function then looks at every shortest path possible. Starting at vertex 0 the function and going to the largest vertex. If i is the starting point (j) or the ending point (k) it is ignored. The path is also ignored if j and k are equal. If the shortest distance from j to k includes i, then the betweenness is increased by 1. The total betweenness is then returned to the main function.

```
Directed Betweenness Centrality Distribution of Vertex i(vertex i){
    N = max vertex
    B = betweenses = 0
    for(j = 0 to N){
        if(i != j){
            for(k = 0 to N){
                if(i != k && j != k){
                    shortest path between j and k
                    if i is included, increase betweenses by 1
                }
            }
        }
    }
}
```

3.10.1 Complexity Analysis

This function has a complexity of n^2. Each vertex must check every other vertex to determine all of the shortest paths. This gives it a complexity of $\mathcal{O}(n^2)$.

3.11 Undirected Unweighted Betweenness Centrality Distribution

This algorithm takes an edge e from the main function. The function then looks at every shortest path possible. Starting at vertex 0 the function and going to the largest vertex. K is defined as j + 1 because the shortest path in undirected graphs, the shortest distance from j -¿ k == the shortest distance from k -¿ j. If the shortest distance from j to k includes the edge e, the value of the betweenness is increased by 1. The total betweenness is then returned to the main function.

```
Undirected Unweighted Betweenness Centrality Distribution of Edge e(edge e){
    N = max vertex
    B = betweenses = 0
    for(j = 0 to N){
        for(k = j+1 to N){
            shortest path between j and k
            if e is included, increase betweeneses by 1
            }
        }
    }
    return B;
}
```

3.11.1 Complexity Analysis

This function has a complexity of n. Each vertex check every vertex larger than itself to determine all of the shortest paths. The first vertex must check nearly other ever vertex. This gives it a complexity of n.

3.12 Directed Unweighted Betweenness Centrality Distribution of Edge

This algorithm takes an edge e from the main function. The function then looks at every shortest path possible. Starting at vertex 0 the function and going to the largest vertex. If the shortest distance from j to k includes the edge e, the value of the betweenness is increased by 1. The total betweenness is then returned to the main function.

```
Directed Unweighted Betweenness Centrality Distribution of Edge e(edge e){
    N = max vertex
    B = betweenses = 0
    for(j = 0 to N){
        for(k = 0 to N){
            if(j != k){
                shortest path between j and k
                if e is included, increase betweeneses by 1
```

```
            }
        }
    }
    return B;
}
```

3.12.1 Complexity Analysis

This function has a complexity of n^2. Each vertex must check every other vertex to determine all of the shortest paths. This gives it a complexity of $\mathcal{O}(n^2)$.

3.13 Undirected Weighted Betweenness Centrality Distribution of Edge e

This algorithm takes an edge e from the main function. The function then looks at every shortest path possible. Starting at vertex 0 the function and going to the largest vertex. K is defined as j + 1 because the shortest path in undirected graphs, the shortest distance from $j \to k ==$ the shortest distance from $k \to j$. If the shortest distance from j to k includes the edge e, the value of the betweenness is increased by 1. The total betweenness is then returned to the main function divided by the weight of e.

```
Undirected Weighted Betweenness Centrality Distribution of Edge e(edge e){
    N = max vertex
    B = betweenness = 0
    for(j = 0 to N){
        for(k = j+1 to N){
            shortest path between j and k
            if e is included, increase betweenness by 1
            }
        }
    }
    return B/(weight of e);
}
```

3.13.1 Complexity Analysis

This function has a complexity of n. Each vertex check every vertex larger than itself to determine all of the shortest paths. The first vertex must check nearly other ever vertex. This gives it a complexity of $\mathcal{O}(n)$.

```
Directed Weighted Betweenness Centrality Distribution of Edge e(edge e){
    N = max vertex
    B = betweenness = 0
    for(j = 0 to N){
        for(k = 0 to N){
            if(j != k){
```

```
                shortest path between j and k
                if e is included, increase betweeneses by 1
            }
        }
    }
    return B/(weight of edge e);
}
```

This algorithms takes an edge e from the main function. The function then looks at every shortest path possible. Starting at vertex 0 the function and going to the largest vertex. If the shortest distance from j to k includes the edge e, the value of the betweenness is increased by 1. The total betweenness is then returned to the main function divided by the edge e.

3.13.2 Complexity Analysis

This function has a complexity of n^2. Each vertex must check every other vertex to determine all of the shortest paths. This gives it a complexity of $\mathcal{O}(n^2)$.

3.14 Community Detection

To detect communities of vertices within the data set we set out to remove those edges that are most likely between those communities to make those communities more apparent. To do this we take the betweenness centrality of edges and sort them in decending order. Then we remove the edge with the highest value (or edges if there are more than one of the same value). Once this is complete we simply recalculate the matrix of shortest paths using the Floyd-Warshall algorithm and the graph diameter using the algorithm detailed above. We repeat this for a total of five times and the resulting diameters are our likely communities.

```
UWBetween[] = Unweighted Betweenness Centrality Edges
V = Original Network

For k from 0 to 4:
    DescendSort(UWBetween)
    UWBetween[0] = x
  while (UWBetween[0] == x)
      V.remove(UWBetween[i])
      UWBetween.remove(UWBetween[0])
  Floyd(V)            //Run Floyd-Warshall Algorithm on revised data set
  Diameter(V)         // Calculate and output max diameter based on new matrix from above
  Betweenness(V)      // Calculate Betweenness centrality based on
                      // new matrix giving us a new UWBetween array
```

3.14.1 Complexity Analysis

Because the Community detection algorithm calls the algorithms to find all shortest paths, the diameter and the unweighted betweenness centrality edges within it and since we are executing this algorithm a total of five times, the complexity of this algorithm is five times the sum of the complexities of these algorithms.

4 Plan of Experiments

The major purpose of our experiment is to dissect and examine networks, so we propose the following plan of experiments:

1. Extrapolate data and store in a relevant data structure — in our case, a sorted `map<int, vector<pair<int, double>>>`
    - The `int` is the key to be used, signifying the origin.
    - The `vector<pair>` holds all the adjacent edges.
2. Iterate over the entirety of the data structure to determine degree distribution.
    - For *weighted and unweighted out degree*, it is simply counting the the `vector` size with respect to each key.
    - For *weighted and unweighted in degree*, making an efficient algorithm is still difficult — not only is by default $\mathcal{O}(n)$ but there is an efficient way of finding where the edges lead to. We accomplish this by a Binary Search Tree. By iterating over every vertex and storing their destination in a binary search tree, we have achieved a sufficient algorithm.
3. Detect the shortest path via the Floyd-Warshall algorithm for directed graphs.
    - Make the graph undirected by making it symmetric about the $a_{i,i}$ elements $\forall i \in$ edges.
    - Test again.
4. Detect closeness centrality and betweenness.
5. Implement community detection.
6. Output results to user.

5 Team Roles

- Illya Starikov
  - Project Manager
  - Official Write-up
- Timothy Ott
  - Pseudocode Write-up
  - Algorithm Analysis
- Claire Trebing
  - Pseudocode Write-up
  - Algorithm Analysis