Algorithms 2500

Lecture 1B

Project Two

First Report

Name 1: Brady Snedden

Student I.D. 1: 12422881

Name 2: Ryon Owings

Student I.D. 2: 12433574

Name 3: Alex Shaw

Student I.D. 3: 12423794

Project Goal: Analyze and characterize follower-followee Social Networks with respect to topological metrics

Abstract: There are many social media websites which use the follower-followee structure. By using graph structures, we can see how to improve this follower-followee structure and lead it to better optimization. We will be finding ways to optimize these structures by finding their shortest paths between users and their communities. We plan to do this by running a series of experiments to test the shortest paths in weighted, unweighted, directed, and undirected graphs. We will also find the Betweenness Centrality of vertices and edges, and detect communities within the graphs. Our plan of experiments is to use the Floyd-Warshall algorithm and provide additional functions based on what situation the graph is in as well as the current task, such as keeping track of the lengths of the shortest paths for an unweighted directed graph. We will also be using classes to hold data for our nodes and edges. Coding work will be divided among the three of us, along with the work of writing the reports. We will check each other's work for understanding and to check for mistakes to ensure the most efficient code and teamwork.

Introduction and Motivation: There are many social media websites on the Internet. The most famous of these include Facebook, Twitter, and YouTube. These social media websites use a system of follower-followee: where a user will "follow" another for updates to their profile. Facebook uses "friends" and "likes," Twitter uses "followers," and YouTube uses "subscribers." To understand and improve how the systems of these websites work, studies of the graph structure of the user base are required. These graphs show how users are connected to one another on the website. These can be optimized by limiting the length between any two users on databases and move their data to closer databases for faster processing/computing time. However, in this project we are more concerned with finding the shortest paths in both directed graphs (where user One is following user Two, but user Two is not necessarily following user One – follower-followee) and in undirected graphs (where both users are following one another). We will also be detecting communities, which are subsets that have mutual connections (where many users are all following one another). We will be doing this in hopes of optimizing social media websites to make them easier for the users to browse.

Proposed Solutions:

Universal Functions

Classes will be created for both our vertices (nodes) and edges. We will also create sets of vertices and edges to allow for better management of them. The member variables of the vertex class will include a variable for the name of the vertex, the number of outgoing edges, the number of incoming edges, and the number of times the vertex is used in a shortest path. For the edge class, the member variables will include the index of its source vertex in the set of vertices, the index of the target vertex, the edge's weight, the edge's value for use in unweighted graphs, and the number of times the edge is used in a shortest path.

Directed Degree Distribution

Finding the Degree Distribution for both unweighted and weighted directed graphs will be done in a preprocessing phase. While the input is being processed, several components will be stored per vertex. This includes the number of outgoing edges and the number of incoming edges. This takes care of the both the Out-degree and In-degree for every vertex. To calculate the largest degree value discovered for each vertex, the following pseudo-code will be used. This code works for both unweighted and weighted graphs and has a time complexity of O(V) where V is the number of vertices in the graph.

```
HighestDegree(V)      //Input is the list of vertices of the graph
{
    for i=1 to numVertices
    {
        //Check to see if current in-degree is greater than the stored max value
        if (V[i].num_in_edges > max_in)
        {
            empty vertices_in;    //Set of vertices with the highest in-degree
            add i to vertices_in;
            max_in = V[i].num_in_edges;
        }
        else if (V[i].num_in_edges == max_in)
        {
            add i to vertices_in;
        }
        //Check to see if current out-degree is greater than the stored max value
        if (V[i].num_out_edges > max_out)
        {
            empty vertices_out;//Set of the vertices with the highest out-degree
            add i to vertices_out;
            max_out = V[i].num_out_edges;
        }
        else if (V[i].num_out_edges == max_out)
        {
            add i to vertices_out;
        }
    }
}
```

Unweighted Shortest Path Distribution

For this, we plan to use the Floyd-Warshall algorithm. The Floyd-Warshall algorithm (with several alterations) will be used to determine the length of the shortest path between each possible combination of vertices in a given graph.

The first alteration is that there will be a single matrix-like array (d[ ]) to hold the lengths of the shortest paths between all combinations of vertices - this array will be updated continuously instead of making a new array in each iteration. The runtime complexity of the Floyd-Warshall algorithm is O(V^3) where V is the number of vertices in the graph.

The second alteration is that, in order to retrieve the vertices and edges comprising the shortest path between two vertices, an array of pointers (pred[ ]) will be utilized. Each index (pred[i, j]) will hold a pointer to the vertex k that forms the midpoint between vertices i and j. The vertices and edges that comprise the shortest path can be retrieved in the recursive ShortestPath algorithm (see further below).

```
Floyd-Warshall(graph)
{
    d = [graph.numVertices, graph.numVertices];      //Initialize size of 2D array
                                                     to hold lengths of shortest
                                                     paths between all possible
    for i = 1 to graph.numVertices do                combinations of vertices
        for j = 1 to graph.numVertices do
        {
            d [i, j] = graph.edges.[i, j].weight;    //Initialize the path length
                                                     between the two
            pred[i, j] = nil;                        vertices - if the vertices are
                                                     not adjacent, the weight is
        }                                            initialized to infinity
    for k = 1 to graph.numVertices do
        for i = 1 to graph.numVertices do
            for j = 1 to graph.numVertices do
                if(d[i,k] + d[k,j] < d[i,j])
                {
                    d[i,j] = d[i,k]+d[k,j];
                    pred[i,j] = k;
                }
    return d[1...graph.numVertices, 1...graph.numVertices];
}
```

Unweighted Graph Diameter

To find the Unweighted Graph Diameter, the following pseudo-code will be used. The same algorithm will be used for both the directed and undirected graphs. The diameter of the graph is the same as the Longest Shortest Path between any two vertices within the graph. To find this longest path, we would first use the Floyd-Warshall algorithm to generate a matrix of the vertices and each of their shortest paths to every other vertex. Using this matrix, the rest is fairly straightforward. The algorithm iterates through the entire matrix and if it encounters a path length that is greater than the stored longest length, then it replaces the longest length with the new greater value. It also clears the previously stored lists of vertices which held the longest paths between them, then the new vertices are added. If the algorithm comes across a path length that is equal to the stored longest length, there is no need to update the longest length, so all it will do is add the new set of vertices to the lists of vertices. These lists store all pairs of vertices that generate shortest paths with the longest length. This algorithm maintains a time complexity of $O(V^2)$ where V is the number of vertices.

```
GraphDiameter(d)      //Input parameter is the matrix generated by the
{                                         Floyd-Warshall algorithm
        for i = 1 to numVertices do
        {       for j = 1 to numVertices do
                {
                        if d[i, j] > lspLength   //length of the Longest Shortest Path
                        {                        is initially zero

                                empty SourceVertices and TargetVertices
                                add i to SourceVertices;       //sets of vertices on either end
                                add j to TargetVertices;       of the Longest Shortest Path
                                lspLength = d[i, j];
                        }
                        else if d[i,j] == lspLength
                        {
                                add i to SourceVertices;
                                add j to TargetVertices;
                        }
                }
        }
return (SourceVertices, TargetVertices, lspLength)
}
```

Unweighted Closeness Centrality

To find the Unweighted Closeness Centrality of the graph the following pseudo-code will be used. The same algorithm will be used for both the directed and undirected graphs. The closeness centrality for a vertex is the average of the shortest path lengths for that vertex to every other vertex within the graph. To find this value, we would first use the Floyd-Warshall algorithm similar to what was used prior to finding the Unweighted Graph Diameter shown previously. Given the matrix of shortest paths, this algorithm will iterate through said matrix and calculate the average for each vertex. If the current vertex has no possible path to another vertex, that value will simply be skipped and not included in the average. To find the average, all of the valid path lengths per vertex will be summed and then divided by the number of valid paths. Additionally, this algorithm will monitor these average values and update the stored largest average whenever a larger average value is encountered. The vertex associated with said average value will be stored as well. The runtime complexity for this algorithm is $O(V^2)$.

```
ClosenessCentrality(d)          //Input parameter is the matrix generated by the
{                               Floyd-Warshall algorithm
    for i = 1 to numVertices do
    {
        numPaths = 0; //number of valid paths initialized to zero for each vertex
        for j = 1 to numVertices do
        {
            if(d[i, j] != INFINITY)
            {
                numPaths++;             //increment the number of valid paths
                avg[i] = avg[i] + d[i,j];       //the array of averages is initially just
                                                the sum of lengths for that vertex
            }
        }
        avg[i] = avg[i] / numPaths;     //divide the sum by the number of valid paths
        if avg[i] > highestValue        //highestValue is initialized to zero
        {
            empty Vertices;         //set of vertices with the highest Closeness
            add i to vertices;      Centrality value
            highestValue = avg[i];
        }
        else if avg[i] == highestValue
        {
            add i to vertices;
        }
    }
    return (avg, highestValue, vertices)
}
```

Betweenness Centrality Distribution

The following ShortestPath algorithm is used - after the Floyd-Warshall algorithm has found the shortest path between each vertex - to examine which vertices and edges comprise the shortest path. In order to do this, ShortestPath recursively moves through the array of pointers pred[ ] (see description of Floyd-Warshall algorithm.) This will in turn find the Betweenness Centrality for the sets of both vertices and edges. This algorithm can be used for both unweighted and weighted graphs, as well as directed and undirected.

After ShortestPath has run, the maximum betweenness centrality values can be found by iterating through the member variables of the edges and vertices that hold their betweenness centrality values. The worst-case runtime complexity of ShortestPath is O($\log_2 V$), where V is the number of vertices comprising a graph.

```
    ShortestPath(i, j)
    {
        if (pred[i, j] = nil)     //There's only a single edge between vertices i and j
        {
            edge_i_j.edgePaths = edge_i_j.edgePaths + 1;
            edge_i_j.vert_i_paths = edge_i_j.vert_i_paths + 1;
            edge_i_j.vert_j_paths = edge_i_j.vert_j_paths + 1;
        }
        else
        {
            edge_i_j.edgePaths = edge_i_j.edgePaths + 1;          //Increment the
            edge_i_j.vert_i_paths = edge_i_j.vert_i_paths + 1;    number of
            edge_i_j.vert_j_paths = edge_i_j.vert_j_paths + 1;    times edge
                                                                  (i, j) and
                                                                  vertices i and j are
                                                                  part of a shortest path

            ShortestPath(i, pred[i,j]);
            ShortestPath(pred[i,j], j);
        }
    }
```

Community Detection

First, we will use an algorithm that sorts the edges into ascending order based on the Unweighted Betweenness Centrality for the edges. For this we will be using a quicksort algorithm, which has a worst-case runtime complexity of O(E^2) where E is the number of edges.

Next, we will remove the edge with the highest betweenness centrality from the graph. Then, we plan to use the Floyd-Warshall algorithm to find the shortest path lengths for the new graph. We will then find the new Unweighted Graph Diameter using the algorithm described earlier. This process of removing the edge with the highest betweenness centrality and then finding the new Unweighted Graph Diameter will be repeated five times. The Unweighted Betweenness Centrality for the removed edge and the Unweighted Graph Diameter will be recorded and stored for every iteration to find trends in the data. This will be done for both directed and undirected graphs.

Plan of Experiments: We plan to use the Floyd-Warshall algorithm in a variety of ways to find our solutions. We will also build classes for our vertexes (nodes) and edges to help keep track of the shortest paths and several other useful components. The following information can be found in the table below, but more details will be given here. For Directed Degree Distribution, we will be simply going through the list of vertices and comparing their member variables. For the Unweighted Shortest Path Distribution, we are going to use the Floyd-Warshall algorithm, which is used to find the shortest path. For the Unweighted Graph Diameter, we will iterate through the matrix generated by the Floyd-Warshall algorithm to find the longest shortest path within the graph and find the pairs of vertices associated. For the Unweighted Closeness Centrality, we will again iterate through the Floyd-Warshall generated matrix, but this time we will compute the average lengths of the shortest paths from each vertex to every other vertex. For the Betweenness Centrality of the vertices and edges, we will be using a ShortestPath algorithm after the Floyd-Warshall has completed, to find each occurrence that each edge and vertex is a part of a shortest path. This algorithm will be used for the Unweighted Betweenness Centrality for each edge as well as the Unweighted and Weighted Betweenness Centrality for each vertex. For the community detection, we will first create and utilize a quicksort algorithm that will sort the edges of the graph in ascending order, based on the Unweighted Betweenness Centrality, and remove the edge that has the highest value. Then we will use the Floyd-Warshall algorithm on the new graph and find the Unweighted Graph Diameter for this new graph. This will be repeated five times.

Expected Results: We expect that the shortest paths will be easier to find in directed paths rather than undirected paths, because there is a clear path which the data can travel, rather than an "anything goes" kind of style that the undirected paths take. We also expect that there will be a large difference in finding the shortest path between weighted and unweighted graphs. For the Community Detection, we expect that the undirected paths will have a smaller diameter than the directed paths. This is because in undirected, you can take out any edge, as it does not matter what direction it is pointing.

Task	**Algorithm to use**
Directed Degree Distribution	**No special algorithm** – just look through the member variables of the generated set of vertices
Unweighted Shortest Path Distribution	**Floyd-Warshall**
Unweighted Graph Diameter	**Floyd-Warshall**, then iterate through the matrix to find the longest shortest path
Unweighted Closeness Centrality	**Floyd-Warshall**, then iterate through the matrix and compute the average length of the shortest paths from each vertex to every other vertices
Unweighted Betweenness Centrality of vertex v	**Floyd-Warshall**, then **ShortestPath** to find each instance each vertex is encountered in all of the shortest paths
Unweighted Betweenness Centrality of edge e	**Floyd-Warshall**, then **ShortestPath** to find each instance each vertex is encountered in all of the shortest paths
Weighted Betweeness Centrality of edge e	**Floyd-Warshall**, then **ShortestPath** to find each instance each edge is encountered in all of the shortest paths
Community Detection	1. Quicksort algorithm to sort edges in ascending order based on the Unweighted Betweenness Centrality and remove highest edge 2. **Floyd Warshall**¸ then **GraphDiameter** to find the diameter of the new graph (Repeat 5 times)

Team Members Roles: Brady will create the various functions needed to run various experiments and create the classes for the vertices and edges. Ryon will create the additional functions required for the other experiments and handle the program's input of the data. Alex will create the basis of the Floyd-Warshall algorithm, which will be used throughout the project, as well as the ShortestPath algorithm. We will all check each other's work to make sure that we are all understanding what algorithms are being put into the project, and how they should be working. We will also collaborate on the reports so everyone gets a say on what is being put into them.