Network Topology Properties

Shubham Gandhi-801115372

Sai Krishna Telukuntla-

Alexey Smirnov

Robert Bland

Under the guidance of
Dr.Qiong Cheng
Department of Computer Science
College of Computing And Informatics
University of North Carolina at Charlotte
ITCS 6114 + ITCS 8114

Fall 2019

Let $G = \langle V, E \rangle$ be a graph. Let $A = (a_{ij})$ be the adjacency matrix of G

$$a_{ij} = \begin{cases} 1 & \text{if node } i \text{ is connected to node } j \text{ by an edge} \\ 0 & \text{otherwise} \end{cases}$$

In particular, $a_{ii} = 0$ for each node i.

We assume an unweighted graph, or equivalently a uniform weighting function $w_{ij} = 1$ for every i, j.

0.1 - Degree of a node and degree distribution

For a given node i, the degree of i is the number of other nodes connected to i by a single edge. For a directed graph, we distinguish between the *in*-degree, and the *out*-degree - being the number of edges that begin with i and the number of edges that end with i, respectively, related by the following formulae:

$$k_{\text{in}}(i) = \sum_j a_{ji}$$

$$k_{\text{out}}(i) = \sum_j a_{ij}$$

$$k(i) = \sum_j \max\{a_{ij}, a_{ji}\}$$

Observe $\max\{k_{\text{in}}(i), k_{\text{out}}(i)\} \leq k(i) \leq k_{\text{in}}(i) + k_{\text{out}}(i)$. In the case of an undirected graph (in which case A is a symmetric matrix, ie $a_{ij} = a_{ji}$), we have $k_{\text{in}}(i) = k_{\text{out}}(i) = k(i)$ for every node i.

We abbreviate $k_i = k(i)$

Source code:

```
public static int[] degreeDist(graph G) {
    // returns array k[0..N-1],
    // where k[i] = k_i for each node i
    int[] k = new int[G.nodes.size()];
    int[][] A = G.adj_matrix;

    for(int i = 0; i < k.length; i++) {
        // for each node, sum over j

        for(int j = 0; j < k.length(); j++) {
            k[i] += max(A[i][j], A[j][i]);
        }

    }
    return k;
}
```

Runtime: $\Theta(V^2)$
The reason: The code simply loops over every (i, j) with $0 \leq i, j \leq |V|$.

The distribution of degrees in G induces a probability distribution, $P(k) =$ probability of a given node i having degree $k_i = k$. It can be computed by the following formula:

$$P(k) = \frac{N(k)}{N}$$

Where $N(k) =$ number of nodes with degree k, and $N = |V|$ is the number of nodes

Pseudocode:

```
public static float[] probies(graph G) {
    int sort_deg[] = degreeDist(testGraph);
        Arrays.sort(sort_deg);
        // we sort the array of degrees
        // to make counting N(k) easier
        int K = sort_deg[sort_deg.length-1];
        float[] probies = new float[K];
        int N = countInRange(sort_deg, V, 0, K);
        // ^auxiliary method to count length of runs
        for(int i = 0; i < K; i++){
                probies[i] = (float) countInRange(sort_deg, V,
                     i+1, i+1) / N;
        }
}
```

Runtime: $\Theta(V^2)$
The reason: most this method's runtime is contributed by calling **degreeDist** in the first line, which already requires $\Theta(V^2)$ time. Sorting the resulting array then requires $\Theta(V \log V)$ time, and running through it to count runs requires $\Theta(V)$ time, each of which are $\mathcal{O}(V^2)$, thus the runtime is dominated by **degreeDist** and is therefore $\Theta(V^2)$

0.2 - Strength of a node, and strength distribution

Strength of a node is summation of number of possible interaction between i_{th} and j_{th} in a network then strength s_i of a node i is defined in the following way

$$s_i = \sum_j \overline{a}_{ij} w_{ij}$$

Here $\overline{a}_{ij} = \max a_{ij}, a_{ji}$ is the undirected completion of A, and w_{ij} is the total weight between nodes i and j. Since we assume an unweighted graph, we take w_{ij} as 1

Source code:

```
public static int[] strengthNode(PWPlainGraph graph)
{
    int[] strength = new int[graph.getNodesNum()];
        for(int i = 0; i < strength.length; i++)
    {
                Node nodei = graph.getNode(i);
                for(int j = 0; j < strength.length; j++)
        {
            int possibleInteractions=0;
                        if (j == i) continue;
                        Node nodej = graph.getNode(j);
                        for(int e = 0; e < graph.getEdgesNum()
                            ; e++)
            {
                        Edge edge = graph.getEdge(e);
                            if (edge.getStartNode().equals(
                                nodei)&& edge.getEndNode().
                                equals(nodej))
                {
                    possibleInteractions++;
                                }
                            if (edge.getStartNode().equals(
                                nodej)&& edge.getEndNode().
                                equals(nodei))
                {
                     possibleInteractions++;
                                }
                        }
                strength[i]=strength[i]+possibleInteractions;
                }

        }
        return strength;
}
```

Runtime: $\Theta(V^2)$
The reason: the code simply loops through every pair of vertices in two nested loops, ie looping over every (i, j) such that $0 \leq i, j \leq |V|$.

The spread in the strength of a node has been characterized by a distribution function $P(s)$; where

$$P(s) = \frac{N(s)}{\sum N(s)}$$

N(s) being number of nodes with strength s

Source Code:

```
public static float[] calculateSpread(int [] strength,int
    distinct_count_of_strength,int N)
{
    float [] spread=new float[distinct_count_of_strength];
    int index=0;
    for(int i=0;i<strength.length;i++)
    {
        int count=1;
```

```
            for(int j=i+1;j<strength.length;j++)
            {
                if(strength[i]==strength[j])
                {
                    count=count+1;
                }
                else
                {
                    i=j-1;
                    break;
                }
            }
            if(index<distinct_count_of_strength)
            {
                spread[index]=(float)count/N;
                index++;
            }
        }
        return spread;
    }
```

Runtime: $\Theta(V^2)$

The reason: the algorithm loops over all pairs (i,j) such that $0 \le i < j \le |V|$, for a total of $V(V+1)/2 = \Theta(V^2)$ executions of the innermost loop

0.4 - Characteristic path length of a network

Let L_{ij} = length of the shortest path from node i to node j, and $L_{ij} = \infty$ if no such path exists.

A weakly connected graph is one for which $\min\{L_{ij}, L_{ji}\} < \infty$ for every i, j, and a strongly connected graph is one for which $L_{ij} < \infty$ for every i, j. In an undirected graph, $L_{ij} = L_{ji}$.

We use Floyd's algorithm to compute L_{ij} in $\Theta(V^3)$ time:

```
public static int[][] shortest_path(graph G) {
    int N = G.nodes.size();
    int[][] L = new int[N][N];

    /* the algorithm requires arithmetic with infinity,
     * it was accomplished in our code by defining
     * an auxilliary method 'extendSum' which
     * simulated Integer.MAX_VALUE as infinity
     */

    final int INF = Integer.MAX_VALUE;

    L = copy(G.adj_matrix);

    // here: replace all entries of 0 in L with INF

    // Floyd-Warshall alg:
    for (k = 0; k < N; k++) {
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                int sum;
                if (L[i][k] == INF || L[k][j] == INF) {
                    sum = INF;
                }
                else {
                    sum = L[i][k] + L[k][j];
                }
                if (sum < L[i][j]) {
                    L[i][j] = sum;
                }
            }
        }
    }
    return L;
}
```

Runtime: $\Theta(V^3)$
The reason: As can be clearly seen, the algorithm simply loops over all triples (i, j, k) with $0 \leq i, j, k \leq |V|$, performing simple arithmetic in the deepest loop.

The Characteristic Path Length L is simply the average of L_{ij} over every distinct pair i, j

$$L = \frac{1}{N(N-1)} \sum_{i,j} L_{ij}$$

Any entries of $L_{ij} = \infty$ would crush the average, so here we take the average only over the finite entries

```
    public static double CPL(graph G) {
        int[][] L = shortest_path(G);
        final int INF = Integer.MAX_VALUE

        int sum = 0, count = 0;
        for(int i = 0; i < L.length; i++) {
            for(int j = 0; i < L[i].length; j++) {
                if(L[i][j] == INF) continue;
                else {
                    sum += L[i][j];
                    count++;
                }
            }
        }
        return (double) sum / count;
    }
```

Runtime: $\Theta(V^3)$
The reason: the bulk of the runtime is contributed by calling **shortest_path** in the first line, which we have already seen is $\Theta(V^3)$. If we ignore this, however, the rest of the algorithm simply takes an average over a matrix containing exactly V^2 entries, thus a further runtime of $\Theta(V^2)$. However, the $\Theta(V^3)$ from Floyd-Warshall dominates.

0.5 - Clustering coefficient of a network

The average clustering coefficient C is a measure of how likely the network is to clump or form local cliques. Let N_i = the neighborhood of node i, that is, the set of all nodes connected to i by an edge. The local clustering coefficient C_i at a node i is the ratio between the number of edges connecting the elements of N_i to each other, and the total possible number of edges between the elements of N_i, which is given in terms of the degree k_i of node i

$$C_i = \frac{2e_i}{k_i(k_i - 1)}$$

Where e_i = the number of edges connecting between the neighbors of node i. Then, the average clustering coefficient C is given by

$$C = \frac{1}{N} \sum C_i$$

Pseudocode:

```
float C_avg = 0;
int clusters = 0;
int[] clusterNodes = localNodes(testGraph);
float[] C = new float[degrees.length];
for(int i = 0; i < C.length; i++){
    if (degrees[i] > 1) {
        C[i] = (float) clusterNodes[i] / (original_degrees[i]
            * (original_degrees[i] - 1));
        C_avg += C[i];
        clusters += 1;
    }
}

C_avg = C_avg/(V);
```

Runtime: $\Theta(V)$

The reason: Ignoring the fact that this method requires the degrees to already be known (the algorithm we offer for that runs in $\Theta(V^2)$ time), this algorithm simply loops through every node, accessing the degree of each node i and then computing e_i, which takes at most k_i accesses of neighbors of each node i, thus the code runs in $\Theta(V \cdot k_i) = \Theta(V)$ time.

0.6 - Small World" property of a network

A network is said to follow the small-world property if $C >> C_i$ and $L >> L_i$, for most nodes i. In this case, the characteristic path length and the clustering coefficient can be approximated by the expressions $L \approx \frac{\log N}{\log k}$ and $C \approx \frac{k}{N}$, where k is the average degree of the network

No pseudocode or runtime analysis necessary for this one, as this property is calculated based on other properties that are already known, using simple math methods and executing in $\Theta(1)$ time.

0.8 - Mixing behavior of nodes

Interaction dynamics of a network is a very interesting phenomena. The pattern of connectivity among the nodes of varying degrees affects the interaction dynamics of the network. If the high-degree nodes in a network tend to be connected with other high-degree nodes, then the network is 'assortative'. On the other hand, the network is said to be 'disassortative' if the high-degree nodes tend to be connected with other low-degree nodes.

To study the tendency for nodes in networks to be connected to other nodes that are like (or unlike) them, we calculate the Pearson correlation coefficient r of the degrees at either ends of an edge. r is given by the following expression suggested by Newman

$$r = \frac{M^{-1}\sum_i j_i k_i - \left(M^{-1}\sum_i \frac{j_1+k_i}{2}\right)^2}{M^{-1}\sum_i \frac{j_i^2+k_i^2}{2} - \left(M^{-1}\sum_i \frac{j_1+k_i}{2}\right)^2}$$

Here j_i and k_i are the degrees of the vertices at the ends of the i-th edge, with $i = 1, \ldots M$; and M is the total number of edges. The networks having positive r values are assortative in nature.

Source code:

```
public static double mixingBehaviourOfNodes(PWPlainGraph graph
    ) throws Exception
{
        double A = 0 ;
        double B = 0;
        double C = 0;
        double D = 0;
        double r = 0;

        try {

                int V = graph.getNodesNum();

                int nodeArrayDegrees[] = new int[V];
                int E = graph.getEdgesNum();

                double edgeCount  = graph.getEdgesNum();

                for(int i= 0; i < edgeCount; i++ ) {

                        Edge edge = graph.getEdge(i);

                        nodeArrayDegrees[edge.getStartNode().
                            getIndex()]++;
                        nodeArrayDegrees[edge.getEndNode().
                            getIndex()]++;
                }

                for(int i= 0; i < edgeCount; i++) {

                           Edge edge = graph.getEdge(i);
```

```
                            // A Represents summation for j'k'
                            A = A + ((nodeArrayDegrees[edge.
                                getEndNode().getIndex()]) * (
                                nodeArrayDegrees[edge.
                                getStartNode().getIndex()]));


                            // B holds Summation for 0.5 * (j'
                                + k')
                            B = B + ( (0.5) * ((
                                nodeArrayDegrees[edge.
                                getEndNode().getIndex()])  +  (
                                nodeArrayDegrees[edge.
                                getStartNode().getIndex()]) ) )
                                ;

                            // C represents summation for 0.5 *
                                 (j'^2  + k'^2)
                            C = C + 0.5 * ( Math.pow(
                                nodeArrayDegrees[edge.
                                getEndNode().getIndex()] ,  2 )
                                  +  Math.pow(nodeArrayDegrees[
                                edge.getStartNode().getIndex()
                                ], 2)  ) ;
                }

            double inverseEdgeCount =  (1.0 / edgeCount) ;

                A = A * inverseEdgeCount;

                //System.out.println("A:" + A);
        B = B * inverseEdgeCount;

        B = Math.pow(B, 2);

        C = C * inverseEdgeCount;

      // Note: For some input grp file if r value is 'NaN',
           it means One cannot predict the mixing behavior, it
            is equivalent to r = 0"
    r = (A - B) / (C - B);

        }catch(ArithmeticException e) {

                r = 0;

        }
        catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        } finally {


        return r;
        }
```

```
}
```

Runtime: $\Theta(E)$

The reason: Though the expression looks massive, it is composed of four simple sums, each taken over every i from $0 \ldots M-1$, where $M = |E|$ is the number of edges. Each sum is calculated in a parallel manner, thus the runtime is at most $\Theta(4E+1) = \Theta(E)$.

0.10 - Closeness centrality

Closeness Centrality is average distance of a i_{th} node to all other nodes.Node with high closeness value have shorter distance to all nodes. The formula to calculate node x is given by

$$C(x) = \frac{N-1}{\sum d(x,y)}$$

where the sum is taken over all $y \neq x$, $d(x,y)$ is the geodesic distance between node x and node y, and N is the number of nodes in the network.

Source code:

```
public static void closenessCentrality(PWPlainGraph graph)
{
    map = new HashMap<String, String>();
    int V = graph.getNodesNum();
    edgeCount = graph.getEdgesNum();
        nodes2DMatrix = new int[V][V];
        for(int i = 0; i < V ; i++) {
                for(int j = 0; j < V; j ++) {
                    if(i != j) {
                        nodes2DMatrix[i][j] = Integer.
                            MAX_VALUE;
                        }
                }
        }

        int edgeCount = graph.getEdgesNum();

        for(int i = 0; i < edgeCount; i++ ) {
                Edge edge = graph.getEdge(i);

                int startNode =  edge.getStartNode().getIndex
                    ();
                int edgeNode =   edge.getEndNode().getIndex();

                if(map.containsKey(String.valueOf(startNode)))
                     {

                        String endNodes = map.get(String.
                            valueOf(startNode));

                        endNodes = endNodes + "␣" + edgeNode ;
                    map.put(String.valueOf(startNode),
                        endNodes);

            }
            else{

                    map.put(String.valueOf(startNode),
                        String.valueOf(edgeNode));

            }

            // back
            if(map.containsKey(String.valueOf(edgeNode)))
                {

                    String startNodes = map.get(String.
                        valueOf(edgeNode));

                    startNodes = startNodes + "␣" +
                        startNode ;

                    map.put(String.valueOf(edgeNode),
                        startNodes);

            }
            else{
            map.put(String.valueOf(edgeNode), String.
                valueOf(startNode));

            }

    }

    for(int i = 0 ; i < V ; i++) {

            for(int j = 0; j < V ; j++) {

               DFS(i, j, 0, i);

            }
    }

double[] nodes = new double[V];
for(int i = 0 ; i < V ; i++) {

    for(int j = 0; j < V ; j++ ) {

            nodes[i] = nodes[i] + nodes2DMatrix[i][j];

            }

    nodes[i] = (double)V/(double)nodes[i];
            int value=i+1;
    System.out.println("Closeness␣centrality␣for␣each␣node
        "+value+"->"+nodes[i]);
}
```

```
}
```

Runtime: $\Theta(V^3)$

The reason: the algorithm requires to know the shortest path between any two nodes, which was already discussed in 0.4 to take $\Theta(V^3)$ time by Floyd-Warshall. Then, this algorithm sums over each row of the distance matrix, requiring $\Theta(V)$ extra time to compute $C(x)$ for *each* of the $|V|$ remaining nodes x, thus the total runtime of the algorithm is $\Theta(V^3 + V^2) = \Theta(V^3)$, and the Floyd-Warshall algorithm dominates again.