SOCIAL NETWORK ANALYSIS USING COMMUNITY DETECTION ALGORITHM

Project Report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology.

in

Computer Science & Engineering

under the Supervision of

Dr. PARDEEP KUMAR

By

APURVA SRIVASTAVA (111242)

Jaypee University of Information Technology

Waknaghat, Solan – 173234, Himachal Pradesh

CERTIFICATE

This is to certify that the work titled "**Social Network Analysis using Community Detection Algorithm** " submitted by " **Apurva Srivastava "** in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science Engineering of Jaypee University of Information Technology, Waknaghat has been carried out under my supervision.

This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

_____

(Signature of Supervisor)

Name of Supervisor : Dr. Pardeep Kumar

Designation **:** Assistant Professor (Senior Grade)

Date ……..

ACKNOWLEDGEMENT

I express my sincere gratitude to my respected project supervisor Asst. Professor Mr. Pardeep Kumar Department of Computer Science And Engineering, Jaypee University of Information Technology, Waknaghat under whose supervision and guidance this work has been carried out. His whole hearted involvement, advice, support and constant encouragement throughout, have been responsible for carrying out this project work with confidence. I am also grateful to him for providing me with required infrastructural facilities that have been highly beneficial to me in undertaking the above mentioned project..

I would also like to thank the laboratory staff of Department of Computer Science and Engineering for their timely help and assistance.

Apurva Srivastava

Date ……..

TABLE OF CONTENTS

Abbreviations and Symbols

SLM : Smart Local Moving Algorithm

C_i : Community to which node I assigned

A_{ij} : edges between node I and j

Q : modularity Function

List of Figures

Abstract

As we know that data in real world is growing at a very fast pace. So it is very difficult to make analysis on this data which can be represented in the form of network. So to ease the computation or analysis of the complex network we divide the complete network into various communities on the basis of characteristics (i.e. distance between the nodes, modularity of sub graph).

In this project I have studied three different algorithms for community detection in a network. These algorithms are based on the idea of optimizing a modularity function. The idea of detecting communities by optimizing a modularity function was proposed by Newman. First one is Louvain algorithm second one is extension of Louvain algorithm with a so called multilevel refinement procedure and last one is smart local moving (SLM) algorithm. For large size network SLM algorithm is preferred over other two algorithms

Problem Statement

What I supposed to do?

As social networks gain prominence, the first obvious question that comes in mind in observing these networks is: how to extract meaningful knowledge from these data?

As we discussed earlier, the increasing complexity of the graph while analyzing. So finding the communities reduces the complexity of a network's original graph. So I have implemented an algorithm for community detection to solve the above mentioned problem.

Chapter 1: Social Network Analysis

1.1 Introduction

Social network analysis (SNA) is a set of research procedures for identifying structures in systems based on the relations among actors. Grounded in graph and system theories, this approach has proven to be a powerful tool for studying networks in physical and social world.

As social networks gain prominence, the first obvious question that comes in observing these networks is: how to extract meaningful knowledge from these data? In seeking a response, the network structure proves to be of utmost importance. Community detection may become a more complicated task given that social networks can be structured on many different levels, yet communities reduce the complexity of a network's original graph in a substantial way, thus revealing its macro-structure.

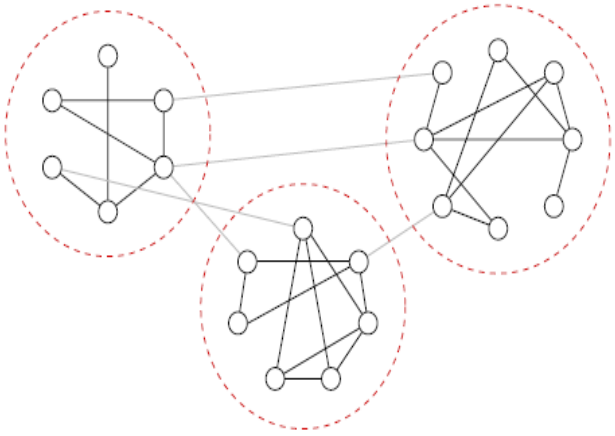


Fig 1: A small network with community structure. In this case there are three communities, denoted by the dashed circles, which have dense internal links but between which there are only a lower density of external links.

1.2 Measures for Network Flow

1.2.1 Centrality –

Location, identifying where an actor resides in a network. Promotes the relationship among its neighbor .group of objects belong to same group .partition set of data on the basis of similarities.

1.2.2 Betweenness

Degree an individual lies between other individuals in the network. The extent to which a node is directly connected only to those other nodes that are not directly connected to each other. Therefore, it's the number of people who a person is connected to indirectly through their direct links. This parameter reflects the popularity of a vertex, in the sense that most popular vertices are those maintaining the highest number of relationships.

1.2.3 Closeness

The degree an individual is near all other individuals in a network (directly or indirectly). Thus, closeness is the inverse of the sum of the shortest distances between each individual and every other person in the network. It is the minimum distance from vertex v to vertex t (the sum of the costs of relationship of all edges in the shortest path from v to t).measure of influence.

1.2.4 Degree

The count of the number of ties to other actors in the network. No. Of shortest path travelling to s & t through v/ no of shortest path travelling to s&t.

1.2.5 Clustering coefficient –

It measures the transitivity of a network. Which suggests that if two vertices that both are neighbors of the same third vertex have a more probability of also being neighbors of each other? This is similar to the fact that if two of your friends will have a higher probability of knowing each other than two people chosen at random from the population, on basis of their common acquaintance with you.

1.2.6 Modularity

Modularity is a measure of strength of sub graph of a network into communities. Networks with high modularity will have a compact structure among the nodes of the same community but sparse connections between nodes in different communities.
Modularity measures the capacity of a given graph partition to yield the densest groups.
The modularity function of Newman and Girvan (2004) is given as :

$$Q=\frac{1}{2m}\sum_{i,j}\left(A_{i,j}-\frac{k_i k_j}{2m}\right)\delta(c_i, c_j),$$

Where, $m = \frac{1}{2}\sum_{i,j} A_{ij}$

Denotes total number of edges in network.,

$$k_i = \sum_j A_{ij}$$

Denotes degree of node I .

The function $\delta(c_i, c_j)$ indicates whether nodes i and j belongs to same community .It equals 1 if $c_i = c_j$ and 0 otherwise .Higher value of modularity function are supposed to indicate a better community structure.

1.3 Modularity Optimization: Local Moving Heuristic

Most popularly approach used for modularity optimization is the *local moving heuristic*. The basics of *local moving heuristic* involves repeatedly move individual nodes from one community to another in such a manner that movement of each node leads to increase of modularity. It iterates over the nodes in a network in random manner. For each node it is determined whether it is possible to further increase modularity by moving the node from its current community to another community. If increase of modularity is possible .then node is moved to the community which leads to the largest modularity gain. The *local moving heuristic* keeps moving of nodes until there is no possibility of further increase in modularity through individual node movements.

1.4) Applications

Communities in a network might represent real social groupings, perhaps by interest or background; communities in a citation network might represent related papers on a single topic; communities in a metabolic network might represent cycles and other functional groupings; communities on the web might represent pages on related topics; hidden communities might represent potential suspicious activity. Being able to identify these communities could help us understand and exploit these networks more effectively.

Medical science: Suppose we have found formula to cure a particular type of infection. So we can apply community detection on the database of the patients to group together patients having same type of symptoms. We can give same cure to each of these patients.

Recommender systems development: In this people having common school, interests, friends and many more can be grouped together in the same community. This community result must be used by recommender system while recommending friends or pages to others.

Knowledge Management and Collaboration: SNAs can help locate expertise, seed new communities of practice, develop cross-functional knowledge-sharing, and improve strategic decision-making across leadership teams.

Team-building: SNAs can contribute to the creation of innovative teams and facilitate post-merger integration. SNAs can reveal, for example, which individuals are most likely to be exposed to new ideas.

Human Resources: SNAs can identify and monitor the effects of workforce diversity, on-boarding and retention, and leadership development. For instance, an SNA can reveal whether or not mentors are creating relationships between mentees and other employees.

Strategy: SNAs can support industry ecosystem analysis as well as partnerships and alliances. They can pinpoint which firms are linked to critical industry players and which are not.

CHAPTER 2: Algorithms

2.1) Louvain Algorithm

It starts with each node in a network belonging to its own community .Initially, each community is considered as singleton community. Then it applies *local moving heuristic* approach to obtain an improved community structure. So, each individual node is moved from one community to another community until no further increase in modularity is possible. So, it forms a reduced network in which each node corresponds with a community in original network. Edges between the nodes in the same community in original network results in self links in reduced network. Then it proceeds further by assigning each node in the reduced network to its own singleton community .then again *local moving heuristic* is applied in the reduced network, in the same way as it was earlier applied to original network. Again a second reduced network is achieved and its treated in same way as first reduce network is treated. It continues until a network is obtained that can't be reduced further.

Algorithm:

Input:

A: adjacency matrix of a network

C: initial assignment of nodes to communities

Output:

C: final assignment of nodes to communities

//apply local moving heuristic

C=localmovingheuristic(A,C)

If numberofcommunity(c) < numberofnodes(a) then

//construction of reduce network (A,C)

$\mathrm{A_{reduc}} = \mathrm{reducnetwork(A, C)}$

$\mathrm{C_{reduc}} = [1 \ldots . \mathrm{numberofnodes(A_{reduc})}]$

$\mathrm{C_{reduc}} = \mathrm{LouvianAlgo(A_{reduc}, C_{reduc})}$

$C_{old} = C$ //merge community

For i=1 to numberofcommunity(C_{old}) do

If (c[i]=c_{old}[i])

$C_{reduc}[j] = c[i]$

End for

End if

2.2 Louvain algorithm with multilevel refinement

Louvain algorithm achieve solutions that are locally optimal with respect to community merging. It is not optimal with respect to movements of individual nodes between communities .*local moving heuristics* I is applied not only for creating an initial community structure for the nodes in a network but also for refining the final community structure .

Algorithm :

Input:

A: adjacency matrix of a network

C:initial assignment of nodes to communities

Output:

C:final assignment of nodes to communities

//apply local moving heuristic

C=localmovingheuristic(A,C)

If numberofcommunity(c) < numberofnodes(a) then

//construction of reduce network (A,C)

$A_{reduc} = \text{reducnetwork}(A, C)$

$C_{reduc} = [1 \dots . \text{numberofnodes}(A_{reduc})]$

$C_{reduc} = \text{LouvianAlgowithmultirefinement}(A_{reduc}, C_{reduc})$

$C_{old} = C$ *//merge community*

For i=1 to numberofcommunity(C_{old}) do

If (c[i]=c_{old}[i])

$C_{reduc}[j] = c[i]$

End for

C=localMovingheuritic(A,C)

End if

2.3 Smart Local Moving Algorithm

It searches for all possibilities to increase modularity by splitting up communities and by moving sets of nodes from one community to another.
It uses local moving heuristics to improve community structure .It iterates over all communities in present community structure. For each community , a sub-network is constructed , it includes only the nodes belonging to the specific community of interest .It again uses *local moving heuristics* to identify communities in sub-network.

After finding community structure for each of sub-networks , it constructs a reduced network where each node corresponds to a community in one of sub-networks . It initially assigns nodes to community. Hence, for each sub-network ,there is one community in the reduced network. All process repeated to reduced network rather than original network .

Algorithm :

Input:

A: adjacency matrix of a network

C:initial assignment of nodes to communities

Output:

C:final assignment of nodes to communities

//apply local moving heuristic

C=localmovingheuristic(A,C)

If numberofcommunity(c) < numberofnodes(a) then

//For each community ,construct a sub network and run the local moving heuristics

//construction of reduce network based on community structure of sub networks.

$C_{old} = c$

J=0;

For (i=1 to numberofcommunity(C_{old}) do

$A_{sub} = \text{Subnetworks}(A, C_{old}, i)$

$C_{sub} = [1 \dots \dots \dots \text{NumberofNodes}(A_{sub})]$

$C_{sub} = \text{LocalMovingHeuristics}(A_{sub} C_{sub})$

If(C[i]== $C_{old}[i]$)

C[i]=C_{sub} + j;

$C_{reduc}[j+1] \dots \dots \dots \dots [j + \text{numberofcommunities}(C_{sub})]) = i;$

J=j + Numberofcommunity(C_{sub})

End for;

$A_{reduc} = \text{reducnetwork}(A, C)$

$C_{reduc} = \text{SmartLocalMoving}(A_{reduc}, C_{reduc})$

$C_{old} = C$ //merge community

For i=1 to numberofcommunity(C_{old}) do

If (c[i]=$c_{old}[i]$)

$C_{reduc}[j] = c[i]$

End for

End if.

2.4 Input :

Zachary graph:

Over the course of two years in the early 1970s, Wayne Zachary observed social interactions between the members of a karate club at an American university. He constructed networks of ties between members of the club based on their social interactions both within the club and away from it. By chance, a dispute arose during the course of his study between the club's administrator and its principal karate teacher over whether to raise club fees, and as a result the club eventually split in two, forming two smaller clubs, centered around the administrator and the teacher.

It is an undirected and unweighted graph. It consists of 34 vertices and 78 edges.

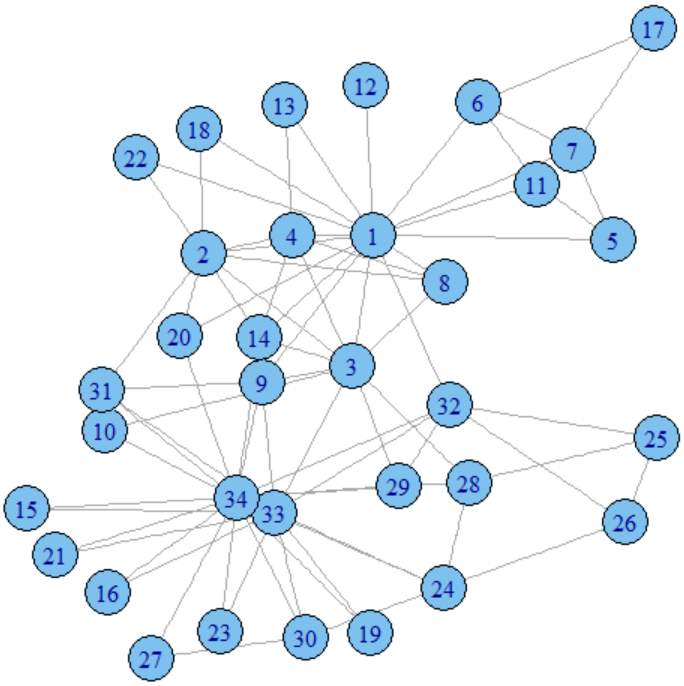


Fig 2: A network structure extracted from Zachary's observations before the split.

Chapter 3: CODE IMPLEMENTATION

3.1) Modularity Optimizer

```
import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.Console;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.util.Arrays;

import java.util.Random;

public class ModularityOptimizer

{

   public static void main(String[] args) throws IOException

   {

      boolean printOutput, update;

      Console console;

      double modularity, maxModularity, resolution, resolution2;

      int algorithm, i, j, modularityFunction, nClusters, nIterations, nRandomStarts;

      int[] cluster;

      long beginTime, endTime, randomSeed;

      Network network;

     Random random;
```

```
String inputFileName, outputFileName;
if (args.length == 9)
{
   inputFileName = args[0];
   outputFileName = args[1];
   modularityFunction = Integer.parseInt(args[2]);
   resolution = Double.parseDouble(args[3]);
   algorithm = Integer.parseInt(args[4]);
   nRandomStarts = Integer.parseInt(args[5]);
   nIterations = Integer.parseInt(args[6]);
   randomSeed = Long.parseLong(args[7]);
   printOutput = (Integer.parseInt(args[8]) > 0);
   if (printOutput)
   {
      System.out.println("algorithm for modularity optimization");
      System.out.println();
   }
}
else
{
   console = System.console();
   System.out.println("algorithm for modularity optimization");
   System.out.println();
```

```
            inputFileName = console.readLine("Input file name: ");
            outputFileName = console.readLine("Output file name: ");
            modularityFunction = 1;
           resolution =Double.parseDouble(console.readLine("Resolution parameter (e.g., 1.0): "));
            algorithm = Integer.parseInt(console.readLine("Algorithm (1 = Louvain; 2 = Louvain with multilevel refinement; 3 = smart local moving): "));
            nRandomStarts = Integer.parseInt(console.readLine("Number of random starts (e.g., 10): "));
            nIterations = Integer.parseInt(console.readLine("Number of iterations (e.g., 10): "));
            randomSeed = Long.parseLong(console.readLine("Random seed (e.g., 0): "));
            printOutput = (Integer.parseInt(console.readLine("Print output (0 = no; 1 = yes): ")) > 0);
            System.out.println();
        }
        if (printOutput)
        {
            System.out.println("Reading input file...");
            System.out.println();
        }
        network = readInputFile(inputFileName, modularityFunction);
        if (printOutput)
        {
            System.out.format("Number of nodes: %d%n", network.getNNodes());
```

```
        System.out.format("Number of edges: %d%n", network.getNEdges() / 2);

        System.out.println();

        System.out.println("Running " + ((algorithm == 1) ? "Louvain algorithm" : ((algorithm == 2) ? "Louvain algorithm with multilevel refinement" : "smart local moving algorithm")) + "...");

        System.out.println();

    }

    resolution2 = (resolution / network.getTotalEdgeWeight()) ;

    beginTime = System.currentTimeMillis();

    cluster = null;

    nClusters = -1;

    maxModularity = Double.NEGATIVE_INFINITY;

    random = new Random(randomSeed);

    for (i = 0; i < nRandomStarts; i++)

    {

        if (printOutput && (nRandomStarts > 1))

            System.out.format("Random start: %d%n", i + 1);

        network.initSingletonClusters();

        j = 0;

        update = true;

        do

        {

            if (printOutput && (nIterations > 1))
```

```
            System.out.format("Iteration: %d%n", j + 1);
        if (algorithm == 1)
            update = network.runLouvainAlgorithm(resolution2, random);
        else if (algorithm == 2)
            update = network.LMultRef(resolution2, random);
        else if (algorithm == 3)
            network.runSmartLocalMovingAlgorithm(resolution2, random);
        j++;
        modularity = network.calcQualityFunction(resolution2);
        if (printOutput && (nIterations > 1))
            System.out.format("Modularity: %.4f%n", modularity);
    }
    while ((j < nIterations) && update);
    if (modularity > maxModularity)
    {
        network.orderClustersByNNodes();
        cluster = network.getClusters();
        nClusters = network.getNClusters();
        maxModularity = modularity;
    }
    if (printOutput && (nRandomStarts > 1))
    {
        if (nIterations == 1)
```

```
                System.out.format("Modularity: %.4f%n", modularity);
            System.out.println();
        }
    }
    endTime = System.currentTimeMillis();
    if (printOutput)
    {
        if (nRandomStarts == 1)
        {
            if (nIterations > 1)
                System.out.println();
            System.out.format("Modularity: %.4f%n", maxModularity);
        }
        else
                                System.out.format("Maximum modularity in %d random starts:
%.4f%n", nRandomStarts, maxModularity);
        System.out.format("Number of communities: %d%n", nClusters);
        System.out.format("Elapsed time: %d seconds%n", Math.round((endTime -
beginTime) / 1000.0));
        System.out.println();
        System.out.println("Writing output file...");
        System.out.println();
    }
```

```
        writeOutputFile(outputFileName, cluster);
    }
    private static Network readInputFile(String fileName, int modularityFunction) throws
IOException
    {
        BufferedReader bufferedReader;
        double[] edgeWeight1, edgeWeight2, nodeWeight;
        int i, j, nEdges, nLines, nNodes;
        int[] firstNeighborIndex, neighbor, nNeighbors, node1, node2;
        Network network;
        String[] splittedLine;
        bufferedReader = new BufferedReader(new FileReader(fileName));
        nLines = 0;
        while (bufferedReader.readLine() != null)
            nLines++;
        bufferedReader.close();
        bufferedReader = new BufferedReader(new FileReader(fileName));
        node1 = new int[nLines];
        node2 = new int[nLines];
        edgeWeight1 = new double[nLines];
        i = -1;
        for (j = 0; j < nLines; j++)
        {
```

```
    splittedLine = bufferedReader.readLine().split("\t");
    node1[j] = Integer.parseInt(splittedLine[0]);
    if (node1[j] > i)
        i = node1[j];
    node2[j] = Integer.parseInt(splittedLine[1]);
    if (node2[j] > i)
        i = node2[j];
    edgeWeight1[j] = (splittedLine.length > 2) ? Double.parseDouble(splittedLine[2]) : 1;
}
nNodes = i + 1;
bufferedReader.close();
nNeighbors = new int[nNodes];
for (i = 0; i < nLines; i++)
    if (node1[i] < node2[i])
    {
        nNeighbors[node1[i]]++;
        nNeighbors[node2[i]]++;
    }
firstNeighborIndex = new int[nNodes + 1];
nEdges = 0;
for (i = 0; i < nNodes; i++)
{
    firstNeighborIndex[i] = nEdges;
```

```
    nEdges += nNeighbors[i];
}
firstNeighborIndex[nNodes] = nEdges;
neighbor = new int[nEdges];
edgeWeight2 = new double[nEdges];
Arrays.fill(nNeighbors, 0);
for (i = 0; i < nLines; i++)
    if (node1[i] < node2[i])
    {
        j = firstNeighborIndex[node1[i]] + nNeighbors[node1[i]];
        neighbor[j] = node2[i];
        edgeWeight2[j] = edgeWeight1[i];
        nNeighbors[node1[i]]++;
        j = firstNeighborIndex[node2[i]] + nNeighbors[node2[i]];
        neighbor[j] = node1[i];
        edgeWeight2[j] = edgeWeight1[i];
        nNeighbors[node2[i]]++;
    }
//standard modularity function
    nodeWeight = new double[nNodes];
    for (i = 0; i < nEdges; i++)
        nodeWeight[neighbor[i]] += edgeWeight2[i];
```

```
        network = new Network(nNodes, firstNeighborIndex, neighbor, edgeWeight2, nodeWeight);
        return network;
    }
    private static void writeOutputFile(String fileName, int[] cluster) throws IOException
    {
        BufferedWriter bufferedWriter;
        int i;
        bufferedWriter = new BufferedWriter(new FileWriter(fileName));
        for (i = 0; i < cluster.length; i++)
        {
            bufferedWriter.write(Integer.toString(cluster[i]));
            bufferedWriter.newLine();
        }
        bufferedWriter.close();
    }
}
```

3.2) Network Module

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
```

```
import java.io.Serializable;
import java.util.Arrays;
import java.util.Random;

public class Network implements Cloneable, Serializable
{
   private static final long serialVersionUID = 1;
   private int nNodes;
   private int[] firstNeighborIndex;
   private int[] neighbor;
   private double[] edgeWeight;
   private double totalEdgeWeightSelfLinks;
   private double[] nodeWeight;
   private int nClusters;
   private int[] cluster;
   private double[] clusterWeight;
   private int[] nNodesPerCluster;
   private int[][] nodePerCluster;
   private boolean clusteringStatsAvailable;

   public static Network load(String fileName) throws ClassNotFoundException,
IOException
   {
```

```
        Network network;
        ObjectInputStream objectInputStream;
        objectInputStream = new ObjectInputStream(new FileInputStream(fileName));
        network = (Network)objectInputStream.readObject();
        objectInputStream.close();
        return network;
    }
    public Network(int nNodes, int[][] edge)
    {
        this(nNodes, edge, null, null, null);
    }
    public Network(int nNodes, int[][] edge, double[] edgeWeight)
    {
        this(nNodes, edge, edgeWeight, null, null);
    }
    public Network(int nNodes, int[][] edge, double[] edgeWeight, double[] nodeWeight)
    {
        this(nNodes, edge, edgeWeight, nodeWeight, null);
    }
    public Network(int nNodes, int[][] edge, double[] edgeWeight, double[] nodeWeight, int[]
cluster)
    {
        double[] edgeWeight2;
```

```
int i, j, nEdges, nEdgesWithoutSelfLinks;
int[] neighbor;
this.nNodes = nNodes;
nEdges = edge[0].length;
firstNeighborIndex = new int[nNodes + 1];
if (edgeWeight == null)
{
   edgeWeight = new double[nEdges];
   for (i = 0; i < nEdges; i++)
      edgeWeight[i] = 1;
}
totalEdgeWeightSelfLinks = 0;
neighbor = new int[nEdges];
edgeWeight2 = new double[nEdges];
i = 1;
nEdgesWithoutSelfLinks = 0;
for (j = 0; j < nEdges; j++)
   if (edge[0][j] == edge[1][j])
      totalEdgeWeightSelfLinks += edgeWeight[j];
   else
   {
      if (edge[0][j] >= i)
         for (; i <= edge[0][j]; i++)
```

```
                firstNeighborIndex[i] = nEdgesWithoutSelfLinks;
            neighbor[nEdgesWithoutSelfLinks] = edge[1][j];
            edgeWeight2[nEdgesWithoutSelfLinks] = edgeWeight[j];
            nEdgesWithoutSelfLinks++;
        }
    for (; i <= nNodes; i++)
        firstNeighborIndex[i] = nEdgesWithoutSelfLinks;
    this.neighbor = new int[nEdgesWithoutSelfLinks];
    System.arraycopy(neighbor, 0, this.neighbor, 0, nEdgesWithoutSelfLinks);
    this.edgeWeight = new double[nEdgesWithoutSelfLinks];
    System.arraycopy(edgeWeight2, 0, this.edgeWeight, 0, nEdgesWithoutSelfLinks);
    if (nodeWeight == null)
    {
        this.nodeWeight = new double[nNodes];
        for (i = 0; i < nNodes; i++)
            this.nodeWeight[i] = 1;
    }
    else
        this.nodeWeight = nodeWeight;

    setClusters(cluster);
}
```

```
    public Network(int nNodes, int[] firstNeighborIndex, int[] neighbor)
    {
        this(nNodes, firstNeighborIndex, neighbor, null, null, null);
    }
    public Network(int nNodes, int[] firstNeighborIndex, int[] neighbor, double[] edgeWeight)
    {
        this(nNodes, firstNeighborIndex, neighbor, edgeWeight, null, null);
    }
    public Network(int nNodes, int[] firstNeighborIndex, int[] neighbor, double[] edgeWeight,
double[] nodeWeight)
    {
        this(nNodes, firstNeighborIndex, neighbor, edgeWeight, nodeWeight, null);
    }
    public Network(int nNodes, int[] firstNeighborIndex, int[] neighbor, double[] edgeWeight,
double[] nodeWeight, int[] cluster)
    {
        int i, nEdges;
        this.nNodes = nNodes;
        this.firstNeighborIndex = firstNeighborIndex;
        this.neighbor = neighbor;
        if (edgeWeight == null)
        {
            nEdges = neighbor.length;
```

```
        this.edgeWeight = new double[nEdges];
        for (i = 0; i < nEdges; i++)
            this.edgeWeight[i] = 1;
    }
    else
        this.edgeWeight = edgeWeight;
    if (nodeWeight == null)
    {
        this.nodeWeight = new double[nNodes];
        for (i = 0; i < nNodes; i++)
            this.nodeWeight[i] = 1;
    }
    else
        this.nodeWeight = nodeWeight;
    setClusters(cluster);
}
public Object clone()
{
    Network clonedNetwork;

    try
    {
        clonedNetwork = (Network)super.clone();
```

```
        if (cluster != null)
            clonedNetwork.cluster = (int[])cluster.clone();
        clonedNetwork.deleteClusteringStats();
        return clonedNetwork;
    }
    catch (CloneNotSupportedException e)
    {
        return null;
    }
}
public void save(String fileName) throws IOException
{
    ObjectOutputStream objectOutputStream;
    objectOutputStream = new ObjectOutputStream(new FileOutputStream(fileName));
    objectOutputStream.writeObject(this);
    objectOutputStream.close();
}
public int getNNodes()
{
    return nNodes;
}   public int getNEdges()
{
    return neighbor.length;
```

```
}
public int[][] getEdges()
{
    int[][] edge;
    int i, j;
    edge = new int[2][neighbor.length];
    for (i = 0; i < nNodes; i++)
        for (j = firstNeighborIndex[i]; j < firstNeighborIndex[i + 1]; j++)
        {
            edge[0][j] = i;
            edge[1][j] = neighbor[j];
        }
    return edge;
}
public double getTotalEdgeWeight()
{
    double totalEdgeWeight;
    int i;
    totalEdgeWeight = totalEdgeWeightSelfLinks;
    for (i = 0; i < neighbor.length; i++)
        totalEdgeWeight += edgeWeight[i];
    return totalEdgeWeight;
}
```

```
public double[] getEdgeWeights()
{
   return edgeWeight;
}
public double getTotalNodeWeight()
{
   double totalNodeWeight;
   int i;
   totalNodeWeight = 0;
   for (i = 0; i < nNodes; i++)
      totalNodeWeight += nodeWeight[i];
   return totalNodeWeight;
}
public double[] getNodeWeights()
{
   return nodeWeight;
}
public int getNClusters()
{
   return nClusters;
}
public int[] getClusters()
{
```

```
    return cluster;
}
public double[] getClusterWeights()
{
    if (cluster == null)
        return null;
    if (!clusteringStatsAvailable)
        calcClusteringStats();
    return clusterWeight;
}
public int[] getNNodesPerCluster()
{
    if (cluster == null)
        return null;
    if (!clusteringStatsAvailable)
        calcClusteringStats();
    return nNodesPerCluster;
}
public int[][] getNodesPerCluster()
{
    if (cluster == null)
        return null;
    if (!clusteringStatsAvailable)
```

```
        calcClusteringStats();
    return nodePerCluster;
}
public void setClusters(int[] cluster)
{
    int i, j;
    if (cluster == null)
        nClusters = 0;
    else
    {
        i = 0;
        for (j = 0; j < nNodes; j++)
            if (cluster[j] > i)
                i = cluster[j];
        nClusters = i + 1;
    }
    this.cluster = cluster;
    deleteClusteringStats();
}
public void initSingletonClusters()
{
    int i;
    nClusters = nNodes;
```

```
    cluster = new int[nNodes];
    for (i = 0; i < nNodes; i++)
        cluster[i] = i;

    deleteClusteringStats();
}
public void findConnectedComponents()
{
    int i, j;
    int[] neighborIndex, node;
    cluster = new int[nNodes];
    for (i = 0; i < nNodes; i++)
        cluster[i] = -1;
    node = new int[nNodes];
    neighborIndex = new int[nNodes];
    nClusters = 0;
    for (i = 0; i < nNodes; i++)
        if (cluster[i] == -1)
        {
            cluster[i] = nClusters;
            node[0] = i;
            neighborIndex[0] = firstNeighborIndex[i];
            j = 0;
```

```
            do
               if (neighborIndex[j] == firstNeighborIndex[node[j] + 1])
                  j--;
               else if (cluster[neighbor[neighborIndex[j]]] == -1)
               {
                  cluster[neighbor[neighborIndex[j]]] = nClusters;
                  node[j + 1] = neighbor[neighborIndex[j]];
                  neighborIndex[j + 1] = firstNeighborIndex[node[j + 1]];
                  neighborIndex[j]++;
                  j++;
               }
               else
                  neighborIndex[j]++;
            while (j >= 0);
            nClusters++;
         }
      deleteClusteringStats();
   }
   public void mergeClusters(int[] newCluster)
   {
      int i, j, k;
      if (cluster == null)
         return;
```

```
    i = 0;
    for (j = 0; j < nNodes; j++)
    {
        k = newCluster[cluster[j]];
        if (k > i)
            i = k;
        cluster[j] = k;
    }
    nClusters = i + 1;
    deleteClusteringStats();
}
public boolean removeCluster(int cluster)
{
    boolean removed;
    if (this.cluster == null)
        return false;
    if (!clusteringStatsAvailable)
        calcClusteringStats();
    removed = removeCluster2(cluster);
    deleteClusteringStats();
    return removed;
}
public void removeSmallClusters(double minClusterWeight)
```

```
{       boolean[] ignore;
    double minClusterWeight2;
    int i, smallestCluster;
    Network reducedNetwork;
    if (cluster == null)
        return;
    reducedNetwork = getReducedNetwork();
    reducedNetwork.initSingletonClusters();
    reducedNetwork.calcClusteringStats();
    ignore = new boolean[nClusters];
    do
    {
        smallestCluster = -1;
        minClusterWeight2 = minClusterWeight;
        for (i = 0; i < reducedNetwork.nClusters; i++)
            if ((!ignore[i]) && (reducedNetwork.clusterWeight[i] < minClusterWeight2))
            {
                smallestCluster = i;
                minClusterWeight2 = reducedNetwork.clusterWeight[i];
            }
        if (smallestCluster >= 0)
        {
            reducedNetwork.removeCluster2(smallestCluster);
```

```
            ignore[smallestCluster] = true;
        }
    }
    while (smallestCluster >= 0);
    mergeClusters(reducedNetwork.getClusters());
}
public void orderClustersByWeight()
{
    orderClusters(true);
}
public void orderClustersByNNodes()
{
    orderClusters(false);
}
public Network getSubnetwork(int cluster)
{
    double[] subnetworkEdgeWeight;
    int[] subnetworkNeighbor, subnetworkNode;
    if (this.cluster == null)
        return null;
    if (!clusteringStatsAvailable)
        calcClusteringStats();
    subnetworkNode = new int[nNodes];
```

```
        subnetworkNeighbor = new int[neighbor.length];
        subnetworkEdgeWeight = new double[edgeWeight.length];
        return getSubnetwork(cluster, subnetworkNode, subnetworkNeighbor,
subnetworkEdgeWeight);
    }
    public Network[] getSubnetworks()
    {
        double[] subnetworkEdgeWeight;
        int i;
        int[] subnetworkNeighbor, subnetworkNode;
        Network[] subnetwork;
        if (cluster == null)
            return null;
        if (!clusteringStatsAvailable)
            calcClusteringStats();
        subnetwork = new Network[nClusters];
        subnetworkNode = new int[nNodes];
        subnetworkNeighbor = new int[neighbor.length];
        subnetworkEdgeWeight = new double[edgeWeight.length];
        for (i = 0; i < nClusters; i++)
            subnetwork[i] = getSubnetwork(i, subnetworkNode, subnetworkNeighbor,
subnetworkEdgeWeight);
        return subnetwork;
```

```
}
public Network getReducedNetwork()
{
    double[] reducedNetworkEdgeWeight1, reducedNetworkEdgeWeight2;
    int i, j, k, l, m, reducedNetworkNEdges1, reducedNetworkNEdges2;
    int[] reducedNetworkNeighbor1, reducedNetworkNeighbor2;
    Network reducedNetwork;
    if (cluster == null)
        return null;
    if (!clusteringStatsAvailable)
        calcClusteringStats();
    reducedNetwork = new Network();
    reducedNetwork.nNodes = nClusters;
    reducedNetwork.firstNeighborIndex = new int[nClusters + 1];
    reducedNetwork.totalEdgeWeightSelfLinks = totalEdgeWeightSelfLinks;
    reducedNetwork.nodeWeight = new double[nClusters];
    reducedNetworkNeighbor1 = new int[neighbor.length];
    reducedNetworkEdgeWeight1 = new double[edgeWeight.length];
    reducedNetworkNeighbor2 = new int[nClusters - 1];
    reducedNetworkEdgeWeight2 = new double[nClusters];

    reducedNetworkNEdges1 = 0;
    for (i = 0; i < nClusters; i++)
```

```
{
    reducedNetworkNEdges2 = 0;
    for (j = 0; j < nodePerCluster[i].length; j++)
    {
        k = nodePerCluster[i][j];
        for (l = firstNeighborIndex[k]; l < firstNeighborIndex[k + 1]; l++)
        {
            m = cluster[neighbor[l]];
            if (m != i)
            {
                if (reducedNetworkEdgeWeight2[m] == 0)
                {
                    reducedNetworkNeighbor2[reducedNetworkNEdges2] = m;
                    reducedNetworkNEdges2++;
                }
                reducedNetworkEdgeWeight2[m] += edgeWeight[l];
            }
            else
                reducedNetwork.totalEdgeWeightSelfLinks += edgeWeight[l];
        }
        reducedNetwork.nodeWeight[i] += nodeWeight[k];
    }
    for (j = 0; j < reducedNetworkNEdges2; j++)
```

```
            {
                reducedNetworkNeighbor1[reducedNetworkNEdges1+j]=
reducedNetworkNeighbor2[j];
                reducedNetworkEdgeWeight1[reducedNetworkNEdges1+j]=
reducedNetworkEdgeWeight2[reducedNetworkNeighbor2[j]];
                reducedNetworkEdgeWeight2[reducedNetworkNeighbor2[j]] = 0;
            }
            reducedNetworkNEdges1 += reducedNetworkNEdges2;
            reducedNetwork.firstNeighborIndex[i + 1] = reducedNetworkNEdges1;
        }
        reducedNetwork.neighbor = new int[reducedNetworkNEdges1];
        reducedNetwork.edgeWeight = new double[reducedNetworkNEdges1];
        System.arraycopy(reducedNetworkNeighbor1, 0, reducedNetwork.neighbor, 0,
reducedNetworkNEdges1);
        System.arraycopy(reducedNetworkEdgeWeight1, 0, reducedNetwork.edgeWeight, 0,
reducedNetworkNEdges1);
        return reducedNetwork;
    }
    public Network getLargestConnectedComponent()
    {
        double maxClusterWeight;
        int i, largestCluster;
        Network clonedNetwork;
        clonedNetwork = (Network)clone();
```

```
    clonedNetwork.findConnectedComponents();
    clonedNetwork.calcClusteringStats();
    largestCluster = -1;
    maxClusterWeight = -1;
    for (i = 0; i < clonedNetwork.nClusters; i++)
        if (clonedNetwork.clusterWeight[i] > maxClusterWeight)
        {
            largestCluster = i;
            maxClusterWeight = clonedNetwork.clusterWeight[i];
        }
    return clonedNetwork.getSubnetwork(largestCluster);
}
public double calcQualityFunction(double resolution)
{
    double qualityFunction, totalEdgeWeight;
    int i, j, k;
    if (cluster == null)
        return Double.NaN;
    if (!clusteringStatsAvailable)
        calcClusteringStats();

    qualityFunction = totalEdgeWeightSelfLinks;
    totalEdgeWeight = totalEdgeWeightSelfLinks;
```

```
    for (i = 0; i < nNodes; i++)
    {
        j = cluster[i];
        for (k = firstNeighborIndex[i]; k < firstNeighborIndex[i + 1]; k++)
        {
            if (cluster[neighbor[k]] == j)
                qualityFunction += edgeWeight[k];
            totalEdgeWeight += edgeWeight[k];
        }
    }
    for (i = 0; i < nClusters; i++)
        qualityFunction -= clusterWeight[i] * clusterWeight[i] * resolution;
    qualityFunction /= totalEdgeWeight;
    return qualityFunction;
}
public boolean runLocalMovingAlgorithm(double resolution)
{
    return runLocalMovingAlgorithm(resolution, new Random());
}

public boolean runLocalMovingAlgorithm(double resolution, Random random)
{
    boolean update;
```

```
double maxQualityFunction, qualityFunction;
double[] clusterWeight, edgeWeightPerCluster;
int bestCluster, i, j, k, l, nNeighboringClusters, nStableNodes, nUnusedClusters;
int[] neighboringCluster, newCluster, nNodesPerCluster, nodeOrder, unusedCluster;
if ((cluster == null) || (nNodes == 1))
   return false;
update = false;
clusterWeight = new double[nNodes];
nNodesPerCluster = new int[nNodes];
for (i = 0; i < nNodes; i++)
{
   clusterWeight[cluster[i]] += nodeWeight[i];
   nNodesPerCluster[cluster[i]]++;
}
nUnusedClusters = 0;
unusedCluster = new int[nNodes];
for (i = 0; i < nNodes; i++)
   if (nNodesPerCluster[i] == 0)
   {
      unusedCluster[nUnusedClusters] = i;
      nUnusedClusters++;
   }
nodeOrder = new int[nNodes];
```

```
for (i = 0; i < nNodes; i++)
   nodeOrder[i] = i;
for (i = 0; i < nNodes; i++)
{
   j = random.nextInt(nNodes);
   k = nodeOrder[i];
   nodeOrder[i] = nodeOrder[j];
   nodeOrder[j] = k;
}
edgeWeightPerCluster = new double[nNodes];
neighboringCluster = new int[nNodes - 1];
nStableNodes = 0;
i = 0;
do
{
   j = nodeOrder[i];

   nNeighboringClusters = 0;
   for (k = firstNeighborIndex[j]; k < firstNeighborIndex[j + 1]; k++)
   {
      l = cluster[neighbor[k]];
      if (edgeWeightPerCluster[l] == 0)
      {
```

```
            neighboringCluster[nNeighboringClusters] = l;
            nNeighboringClusters++;
        }
        edgeWeightPerCluster[l] += edgeWeight[k];
    }
    clusterWeight[cluster[j]] -= nodeWeight[j];
    nNodesPerCluster[cluster[j]]--;
    if (nNodesPerCluster[cluster[j]] == 0)
    {
        unusedCluster[nUnusedClusters] = cluster[j];
        nUnusedClusters++;
    }
    bestCluster = -1;
    maxQualityFunction = 0;
    for (k = 0; k < nNeighboringClusters; k++)
    {
        l = neighboringCluster[k];
        qualityFunction = edgeWeightPerCluster[l] - nodeWeight[j] * clusterWeight[l] *
resolution;
        if ((qualityFunction > maxQualityFunction) || ((qualityFunction ==
maxQualityFunction) && (l < bestCluster)))
        {
            bestCluster = l;
```

```
            maxQualityFunction = qualityFunction;
        }
        edgeWeightPerCluster[l] = 0;
    }
   if (maxQualityFunction == 0)
    {
        bestCluster = unusedCluster[nUnusedClusters - 1];
        nUnusedClusters--;
    }
    clusterWeight[bestCluster] += nodeWeight[j];
    nNodesPerCluster[bestCluster]++;
    if (bestCluster == cluster[j])
        nStableNodes++;
    else
    {
        cluster[j] = bestCluster;
        nStableNodes = 1;
        update = true;
    }
    i = (i < nNodes - 1) ? (i + 1) : 0;
}
while (nStableNodes < nNodes);
newCluster = new int[nNodes];
```

```
    nClusters = 0;
    for (i = 0; i < nNodes; i++)
        if (nNodesPerCluster[i] > 0)
        {
            newCluster[i] = nClusters;
            nClusters++;
        }
    for (i = 0; i < nNodes; i++)
        cluster[i] = newCluster[cluster[i]];
    deleteClusteringStats();
    return update;
}
public boolean runLouvainAlgorithm(double resolution)
{
    return runLouvainAlgorithm(resolution, new Random());
}
public boolean runLouvainAlgorithm(double resolution, Random random)
{
    boolean update, update2;
    Network reducedNetwork;
    if ((cluster == null) || (nNodes == 1))
        return false;
    update = runLocalMovingAlgorithm(resolution, random);
```

```
        if (nClusters < nNodes)
        {
            reducedNetwork = getReducedNetwork();
            reducedNetwork.initSingletonClusters();
            update2 = reducedNetwork.runLouvainAlgorithm(resolution, random);
            if (update2)
            {
                update = true;
                mergeClusters(reducedNetwork.getClusters());
            }
        }
        deleteClusteringStats();
        return update;
    }
    public boolean runLouvainAlgorithmWithMultilevelRefinement(double resolution)
    {
        return runLouvainAlgorithmWithMultilevelRefinement(resolution, new Random());
    }
    public boolean runLouvainAlgorithmWithMultilevelRefinement(double resolution,
Random random)
    {
        boolean update, update2;
        Network reducedNetwork;
```

```
    if ((cluster == null) || (nNodes == 1))
        return false;
    update = runLocalMovingAlgorithm(resolution, random);
    if (nClusters < nNodes)
    {
        reducedNetwork = getReducedNetwork();
        reducedNetwork.initSingletonClusters();
        update2 = reducedNetwork.runLouvainAlgorithm(resolution, random);
        if (update2)
        {
            update = true;
            mergeClusters(reducedNetwork.getClusters());
            runLocalMovingAlgorithm(resolution, random);
        }
    }
    deleteClusteringStats();
    return update;
}
public boolean runSmartLocalMovingAlgorithm(double resolution)
{
    return runSmartLocalMovingAlgorithm(resolution, new Random());
}
public boolean runSmartLocalMovingAlgorithm(double resolution, Random random)
```

```
{
    boolean update;
    int i, j, k;
    int[] reducedNetworkCluster, subnetworkCluster;
    Network reducedNetwork;
    Network[] subnetwork;

    if ((cluster == null) || (nNodes == 1))
        return false;
    update = runLocalMovingAlgorithm(resolution, random);
    if (nClusters < nNodes)
    {
        if (!clusteringStatsAvailable)
            calcClusteringStats();
        subnetwork = getSubnetworks();
        nClusters = 0;
        for (i = 0; i < subnetwork.length; i++)
        {
            subnetwork[i].initSingletonClusters();
            subnetwork[i].runLocalMovingAlgorithm(resolution, random);
            subnetworkCluster = subnetwork[i].getClusters();
            for (j = 0; j < subnetworkCluster.length; j++)
                cluster[nodePerCluster[i][j]] = nClusters + subnetworkCluster[j];
```

```
            nClusters += subnetwork[i].getNClusters();
        }
        calcClusteringStats();
        reducedNetwork = getReducedNetwork();
        reducedNetworkCluster = new int[nClusters];
        i = 0;
        for (j = 0; j < subnetwork.length; j++)
            for (k = 0; k < subnetwork[j].getNClusters(); k++)
            {
                reducedNetworkCluster[i] = j;
                i++;
            }
        reducedNetwork.setClusters(reducedNetworkCluster);
        update |= reducedNetwork.runSmartLocalMovingAlgorithm(resolution, random);
        mergeClusters(reducedNetwork.getClusters());
    }
    deleteClusteringStats();
    return update;
}
private Network()
{
}
private void writeObject(ObjectOutputStream out) throws IOException
```

```
{
    deleteClusteringStats();
    out.defaultWriteObject();
}
private boolean removeCluster2(int cluster)
{
    double maxQualityFunction, qualityFunction;
    double[] reducedNetworkEdgeWeight;
    int bestCluster, i, j;
    reducedNetworkEdgeWeight = new double[nClusters];
    for (i = 0; i < nNodes; i++)
        if (this.cluster[i] == cluster)
            for (j = firstNeighborIndex[i]; j < firstNeighborIndex[i + 1]; j++)
                reducedNetworkEdgeWeight[this.cluster[neighbor[j]]] += edgeWeight[j];
    bestCluster = -1;
    maxQualityFunction = 0;
    for (i = 0; i < nClusters; i++)
        if ((i != cluster) && (clusterWeight[i] > 0))
        {
            qualityFunction = reducedNetworkEdgeWeight[i] / clusterWeight[i];
            if (qualityFunction > maxQualityFunction)
            {
                bestCluster = i;
```

```
                maxQualityFunction = qualityFunction;
            }
        }
    if (bestCluster == -1)
        return false;
    for (i = 0; i < nNodes; i++)
        if (this.cluster[i] == cluster)
            this.cluster[i] = bestCluster;
    clusterWeight[bestCluster] += clusterWeight[cluster];
    clusterWeight[cluster] = 0;
    if (cluster == nClusters - 1)
    {
        i = 0;
        for (j = 0; j < nNodes; j++)
            if (this.cluster[j] > i)
                i = this.cluster[j];
        nClusters = i + 1;
    }
    return true;
}
private void orderClusters(boolean orderByWeight)
{
    class ClusterSize implements Comparable<ClusterSize>
```

```
{
    public int cluster;
    public double size;
    public ClusterSize(int cluster, double size)
    {
        this.cluster = cluster;
        this.size = size;
    }
    public int compareTo(ClusterSize cluster)
    {
        return (cluster.size > size) ? 1 : ((cluster.size < size) ? -1 : 0);
    }
}
ClusterSize[] clusterSize;
int i;
int[] newCluster;
if (cluster == null)
    return;
if (!clusteringStatsAvailable)
    calcClusteringStats();
clusterSize = new ClusterSize[nClusters];
for (i = 0; i < nClusters; i++)
```

```
            clusterSize[i] = new ClusterSize(i, orderByWeight ? clusterWeight[i] :
nNodesPerCluster[i]);
        Arrays.sort(clusterSize);
        newCluster = new int[nClusters];
        i = 0;
        do
        {
            newCluster[clusterSize[i].cluster] = i;
            i++;
        }
        while ((i < nClusters) && (clusterSize[i].size > 0));
        nClusters = i;
        for (i = 0; i < nNodes; i++)
            cluster[i] = newCluster[cluster[i]];
        deleteClusteringStats();
    }
    private Network getSubnetwork(int cluster, int[] subnetworkNode, int[]
subnetworkNeighbor, double[] subnetworkEdgeWeight)
    {       int i, j, k, subnetworkNEdges, subnetworkNNodes;
        Network subnetwork;
        subnetwork = new Network();
        subnetworkNNodes = nodePerCluster[cluster].length;
        subnetwork.nNodes = subnetworkNNodes;
```

```
if (subnetworkNNodes == 1)
{
   subnetwork.firstNeighborIndex = new int[2];
   subnetwork.neighbor = new int[0];
   subnetwork.edgeWeight = new double[0];
   subnetwork.nodeWeight = new double[] {nodeWeight[nodePerCluster[cluster][0]]};
}
else
{
   for (i = 0; i < nodePerCluster[cluster].length; i++)
      subnetworkNode[nodePerCluster[cluster][i]] = i;
   subnetwork.firstNeighborIndex = new int[subnetworkNNodes + 1];
   subnetwork.nodeWeight = new double[subnetworkNNodes];
   subnetworkNEdges = 0;
   for (i = 0; i < subnetworkNNodes; i++)
   {
      j = nodePerCluster[cluster][i];
      for (k = firstNeighborIndex[j]; k < firstNeighborIndex[j + 1]; k++)
         if (this.cluster[neighbor[k]] == cluster)
         {
            subnetworkNeighbor[subnetworkNEdges] = subnetworkNode[neighbor[k]];
            subnetworkEdgeWeight[subnetworkNEdges] = edgeWeight[k];
            subnetworkNEdges++;
```

```
                }
            subnetwork.firstNeighborIndex[i + 1] = subnetworkNEdges;
            subnetwork.nodeWeight[i] = nodeWeight[j];
        }
        subnetwork.neighbor = new int[subnetworkNEdges];
        subnetwork.edgeWeight = new double[subnetworkNEdges];
        System.arraycopy(subnetworkNeighbor, 0, subnetwork.neighbor, 0, subnetworkNEdges);
        System.arraycopy(subnetworkEdgeWeight, 0, subnetwork.edgeWeight, 0, subnetworkNEdges);
      }

      subnetwork.totalEdgeWeightSelfLinks = 0;
      return subnetwork;
   }
   private void calcClusteringStats()
   {
      int i, j;
      clusterWeight = new double[nClusters];
      nNodesPerCluster = new int[nClusters];
      nodePerCluster = new int[nClusters][];
      for (i = 0; i < nNodes; i++)
      {
```

```
            clusterWeight[cluster[i]] += nodeWeight[i];
            nNodesPerCluster[cluster[i]]++;
        }
        for (i = 0; i < nClusters; i++)
        {
            nodePerCluster[i] = new int[nNodesPerCluster[i]];
            nNodesPerCluster[i] = 0;
        }
        for (i = 0; i < nNodes; i++)
        {
            j = cluster[i];
            nodePerCluster[j][nNodesPerCluster[j]] = i;
            nNodesPerCluster[j]++;
        }
        clusteringStatsAvailable = true;
    }
    private void deleteClusteringStats()
    {
        clusterWeight = null;
        nNodesPerCluster = null;
        nodePerCluster = null;
        clusteringStatsAvailable = false;
    }}
```

3.3) Output:

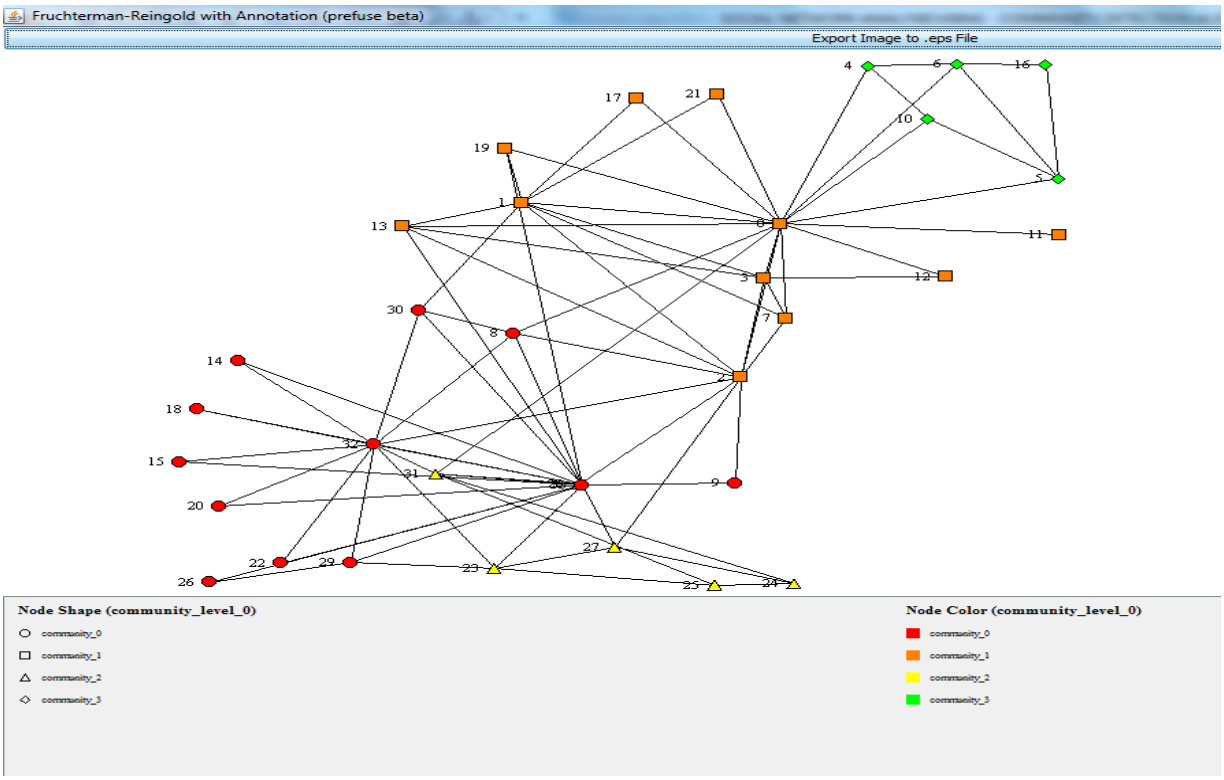


Fig 3:Applying Louvain algorithm to Karate_club network

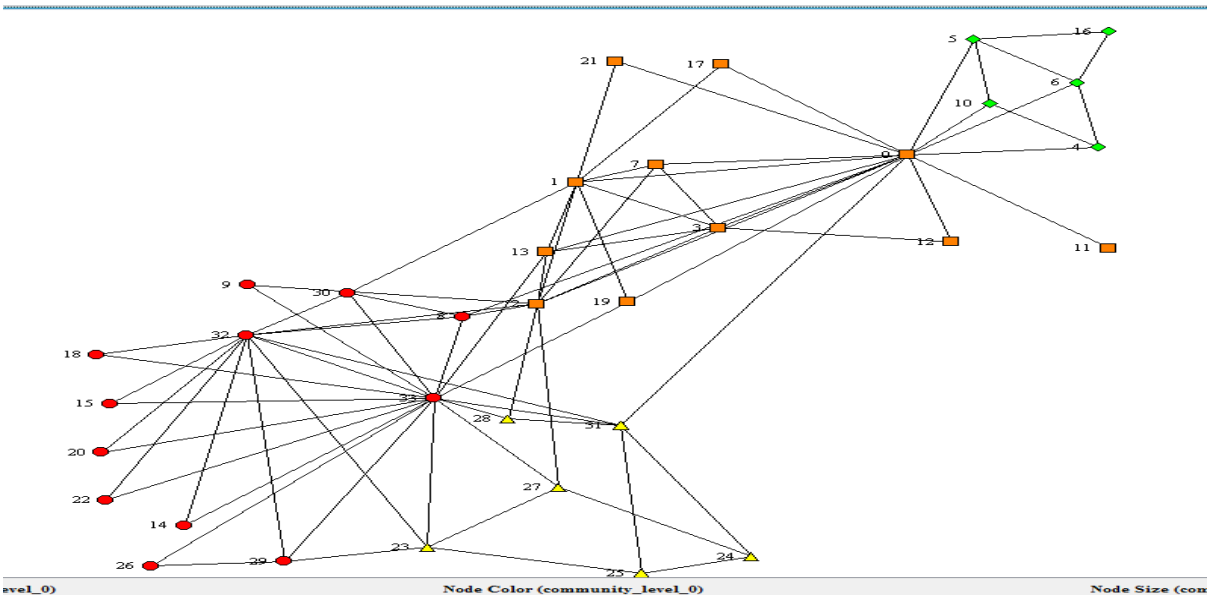
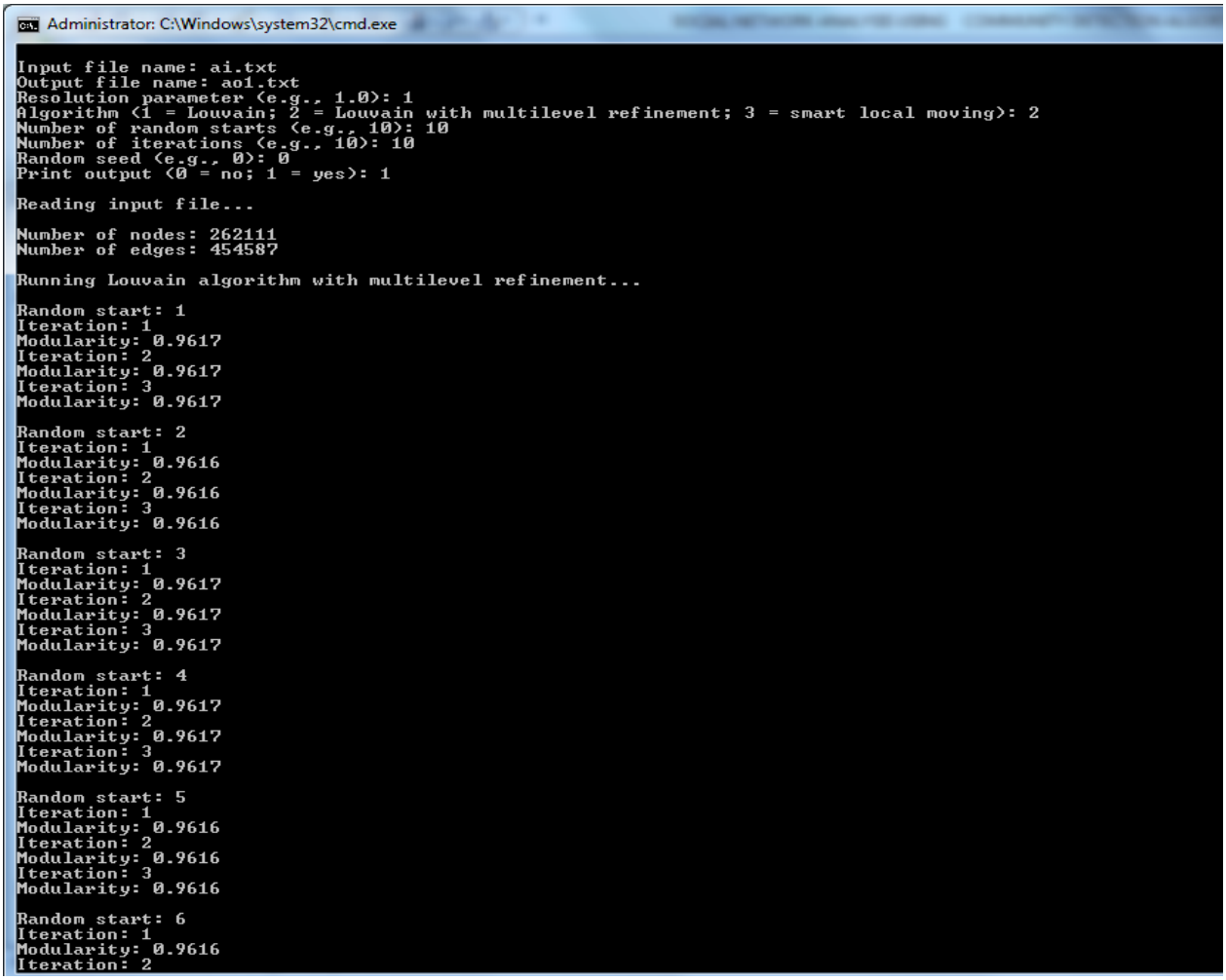
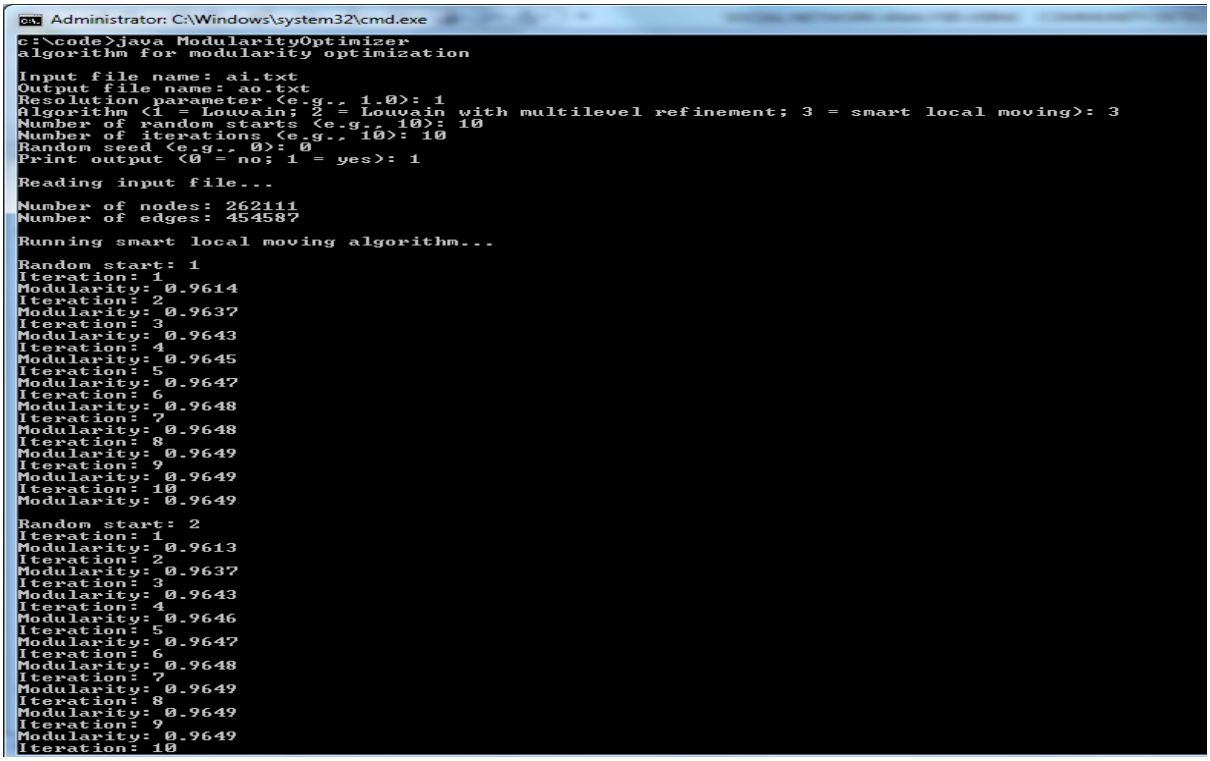


Fig 4 :Applying Louvain with multilevel refinement algorithm on karate_club_network

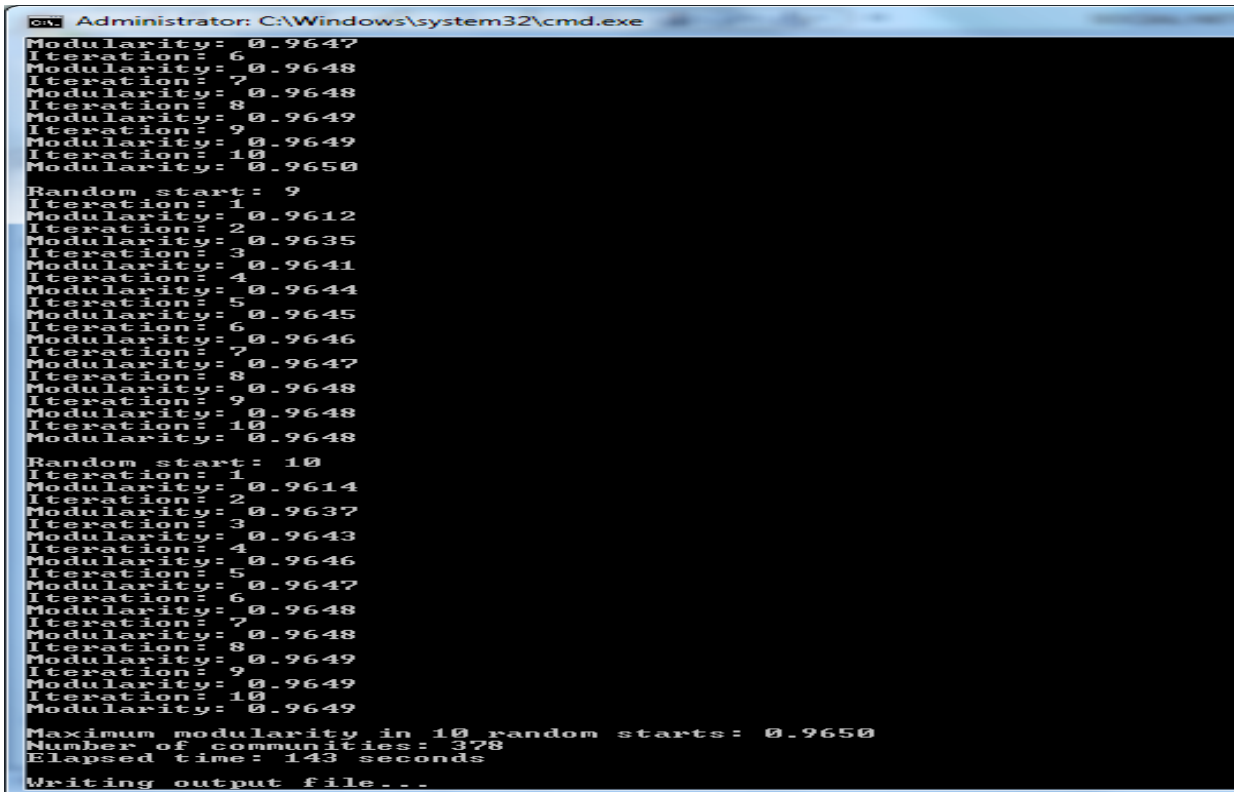


(a)

(b)..fig 5. applying louvain algorithm with multilevel refinement for amazon purchasing netw (a) iteration from random start 1to 6 (b) random start 7 to 10 and maximum modularity =.9617



(a)



(b)

Fig6. Applying SLM on amazon purchasing network data for 10 iteration and 10 random start (a) random start 1 & 2 (b) random start 9&10 with maximum modularity =0.9650

CHAPTER 4 : CONCLUSION

4.1 Conclusion

From all the three algorithms i.e Louvain Algorithm ,Louvain algorithm with multilevel refinement and Smart Local Moving Algorithm it can be concluded that if we want to maximize the modularity of the communities then we should use SLM algorithm for community detection. With 10 iteration per algorithm run ,the SLM algorithm consistently outperforms the original Louvain algorithm and Multilevel algorithm.

With only one iteration per algorithm run, the SLM algorithm slightly outperforms the original Louvain algorithm but the difference is almost negligible. The performance of Louvain algorithm with multilevel refinement is hardly affected by the number of iteration per algorithm run.

In my opinion SLM algorithm is able to identify better community structures for large size network , in terms of modularity than the two algorithm .To identify high-quality community structures ,it is essential to use the iterative variant of the SLM algorithm .In the analysis of large networks , we find that a single run of the SLM algorithm almost always give higher modularity value than 10 runs of the original Louvain or extension of same.

CHAPTER 5 :REFERENCES:

Research Papers:

1. Ludo Waltman and Ness Jan Van Eck “Large Scale moving algorithm for large scale modularity based community detection “ European Physical Journal B 86, 471 , 2013
2. Newman, M.E.J. “Fast algorithm for detecting community structure in networks. ”Physical Review E, 69(6), 066133, 2004.
3. Girvan, M. and Newman, M. E. J. 2002. Community structure in social and biological networks. Proc. National Acad. Sci. 99, 12, 7821--7826
4. Newman, M.E.J . & Girvan “Finding and Evaluating community structure in networks ” Physical Review E,69(2),026113,2004
5. Vincent D. Blondel1, Lambiotte and Etienne Lefebvre1 “Fast unfolding of communities in very large network” Physicscal Review B,0803.0476.
6. S.Fortunato , “Community detection in graphs ” Physics Reports,vol.486
7. S.Fortunato and M.Barthelemy “Resolution limit in community detection” ,PNAS vol.104
8. F.Radicchi , C.Castellano , F.Cecconi , V.Loreto and D.Parisi , :Defining and identifying communities in networks”, PNAS vol.101
9. Rotta and Noack, A. “Multilevel local search algorithms for modularity clustering” , Journal of Experimental Algorithms ,16(2), article 2.3
10. Traag, V.A., Van Dooren, P. & Nesterov, Y. “Narrow scope for resolution-limit-free community detection” Physical Review E, 84(1), 016114, (2011).

Website

1. https://sci2.cns.iu.edu
2. http://snap.stanford.edu/
3. http://nodexl.codeplex.com/