Investigating the RATs: Russian Accounts on Twitter During the 2016 United States Presidential Election

Marisa Papagelis

Wellesley College

Special thank you to Professor Metaxas & Professor Herbst

Introduction:

Leading up to the 2016 United States presidential election, the Russian government infiltrated the United States with thousands of fake social media accounts with the intent of spreading misinformation and harming Senator Hillary Clinton's presidential campaign. False information from Russian government-controlled media reached millions of American social media users between 2013 and 2017.

TwitterTrails, a Web-based investigative tool created as a research project at the Socal Informatics Lab at Wellesley College, analyzes the origin and propagation of rumors from accounts on twitter. In this project, I investigate data on Russian Twitter accounts to create, analyze, and depict a visualization of account users and stories. The data used in this project is stored in file containing the screen name, user identification number, tweet count and story count or each Russian account. Lastly, the data for each account includes the identification numbers of all of the stories that the account interreacted with.

Both the data and the visualization will be used in the final part of the project to draw conclusions pertaining to the use RATs leading up to and during the 2016 Presidential Election.

Methods:

The methods used in this investigation include three classes (AdjListsGraph, RATs, and Investigate) to read in and analyze the Russian account data as well, as a visualization of the data using yEd, a desktop application used to generate diagrams. A description of each method and the graph visualization can be found below.

A. AdjListsGraph Class

This class implements the Graph interface. It contains a vector of generic kind of objects to store the vertices of a graph and a vector of LinkedLists of generic kind of objects to store arcs within the graph. Each LinkedList holds the set of adjacent vertices to a given vertex. The class also consists a constructor with no inputs, to create an empty graph, along with some additional getter methods to gather data about the graph. These methods include getNumVertices to determine the number of vertices in a graph, getNumArces to determine the number of arcs in a graph, isArc to determine if arc (direct connection exists between two vertices, isEdge to determine if an edge exists between two vertices, and isUndirected to determine if the graph is undirected. There is also a getter method called getAllVertices that returns a Vector containing all of the vertices in the graph.

The AdjListsGraph class contains methods such as addVertex, removeVertex, addArc, and removeArc to add/remove vertices, arcs, and edges. There is also a getSuccessors method to return all the vertices in the graph that are adjacent to the given vertex and a getPredecessors to return all vertices in the graph that precede a given vertex using a LinkedList and a given vertex. Additionally, AdjListsGraph contains a toString method to return a string representation of the graph and a saveToTGF method which takes in the name of the output TGF file as a parameter and writes the data to a file using TGF representation with vertices and arcs.

AdjListsGraph completes itself with methods to perform a depth first search and a breadth first search on the data in a graph. The depthFirstSearch method takes in a starting vertex as a parameter and returns a LinkedList containing all of the vertices visited during a depth-first search traversal. The depthFirstSearch method starts out by marking all vertices “not visited” using an array of booleans and then uses a stack, starting with the starting vertex to traverse the graph and mark vertices as visited as it goes. Using a loop, the starting vertex is pushed onto the stack and marked as “visited.” The starting vertex, as well as each additional vertex, contains a LinkedList of its corresponding arcs to be explored. After this, still within the loop, the search visits a new vector at the end of a corresponding arc of the initial vector. This new vector is explored and then placed onto the stack to marked as visited. This traversal continues until all of the corresponding arcs on the vertex are pushed onto the stack, marked as visited, and added to the linked list. When the traversal reaches a vertex that has already been marked visited, the vertex is popped from the stack, and the vertex below it in the stack is explored until it is completely visited. The loop continues until the stack is empty, then the LinkedList containing all of the vertices visited in order is returned.

The breadthFirstSearch method also takes in a starting vertex as a parameter and returns a LinkedList containing all of the vertices visited during a depth-first search traversal. The algorithm is similar to that of the depthFirstSearch method, however it uses a queue, rather than a stack, to keep track of the traversal path. The breadthFirstSearch method also starts by marking all of the vertices in the graph as “not visited.” Using a loop, the starting vertex is enqueued onto the queue and marked as “visited.” While in the loop, as each vertex is visited, it is marked as visited then enqueued onto the queue. The next vertex for the traversal to visit is determined by the vertex in the front of the queue, which is then dequeued and added to the result LinkedList. The loop continues until the queue is empty, then the LinkedList containing all of the vertices visited in order is returned.

B. Rats Class

This class takes in the name of a CSV file as a parameter. It creates an empty graph, called RATgraph to add vertices and edges into in order to create a graph of the data from the CSV file. It uses a helper method called readRats to scan the csv file and pull the data out assigning it to the proper place in the graph using addVertex and addEdge methods from the AdjListsGraph class. This method creates and uses a hashtable, storyTable, which is set up with the story identification number as the key and the number of users using the story as the value. The purpose of the hashtable is to organize the stories and link them to their proper screen name. The graph that is produced is a bipartite graph meaning it is made of Strings with the user’s screen name and the story identification numbers as vertices.

The Rats class includes getter methods to determine the total number of stories, the total number of tweets, the total number of users, and the total number of vertices, along with additional getter methods to return the RATgraph and the most active RAT.

Another function of the Rats class is to determine the most and least popular stories. The class does so by first using a method to called popularity to determine the least and most popular stories. This method does not return anything and only sets the instance variables to the corresponding stories. The class provides getMostPopStory and getLeastPopStory methods to retrieve the most and least popular story identification numbers. In addition to finding the story identification numbers, the class provides a getStoryTitle method that takes in the story's identification number as a parameter and returns the story's title by attaching the number to the end of a given URL and finding said story in the TwitterTrails database. Both the most and least popular story titles are found using this method.

There are two methods within this class called findLCC and findSCC that determine the size of the largest connected component and the smallest connected component respectively. These methods do so by iterating through a vector of the vertices and using a depth first search and a breadth first search respectively to return the size of the largest/smallest component. There is also an isConnected method included in this section which returns a boolean. isConnected returns true if the graph is completely connected. It can figure this out by seeing if the size of the largest component is equal to the total number of vertices (total users and total stories combined).

Lastly, this class includes a main method for testing much like that in the investigate driver class below. This class is only used for testing the immediate class as the investigate class is used to drive the entire investigation.

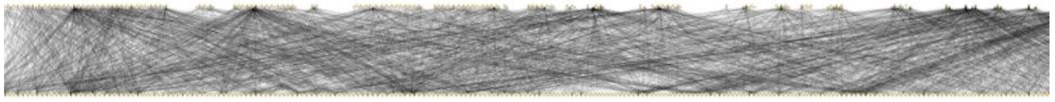
C. Investigate Class

This is the driver class used to create a collection of rats from the csv file and to analyze the data for the investigation. Using only a main method, the Investigate class first creates a collection of rats from the Russian account data file, uses the getGraph method from the rats class to retrieve the graph and save it to a .tgf file, then runs methods from the Rats class to analyze user data and connections from the Russian accounts. These methods analyze the graph using the getter methods getAllTweets, getAllStories, getAllUsers, and getNumVertices to retrieve basic information such as the total number of tweets, the total number of stories, the total number of users, and the total number of vertices. Then, it uses methods from the Rats class, such as getMostPopStory, getLeastPopStory, getStoryTitle, getMostActiveRat, findLCC, findSCC, and isConnected, to determine the identification number and title of the most/least popular story, the screen name of the most active rat, the size of the largest/smallest connected component, and whether or not the graph is completely connected respectively. A general analysis of the data and answers to the questions driving this investigation are printed in the terminal.

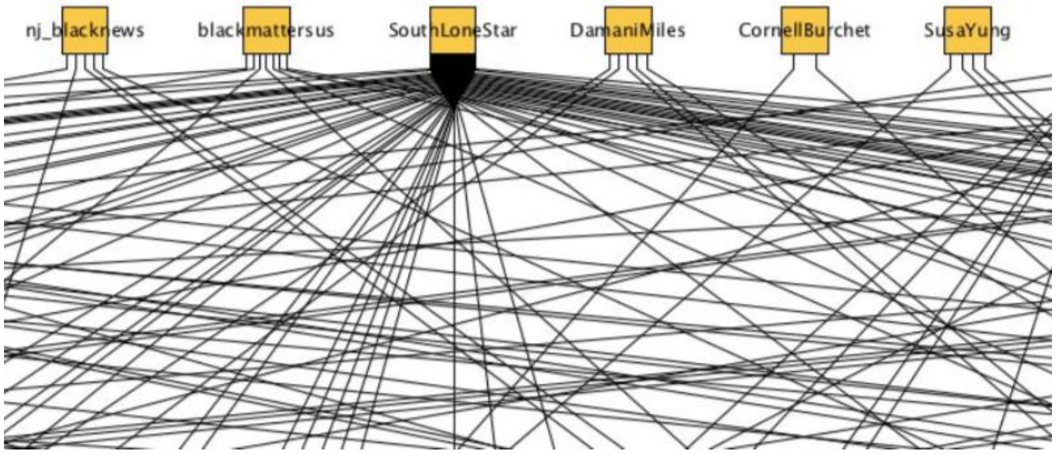
D. yEd Graph Visualization

Using yEd and RATgraph.tgf, I created a visualization of the network created from user screen names and story identification numbers in the Russian account data file. This visualization serves as a confirmation of the results outputted by the code. Because the network is so large, smaller portions of the top and the bottom of the graph are pictured on the next page in addition to the complete graph.

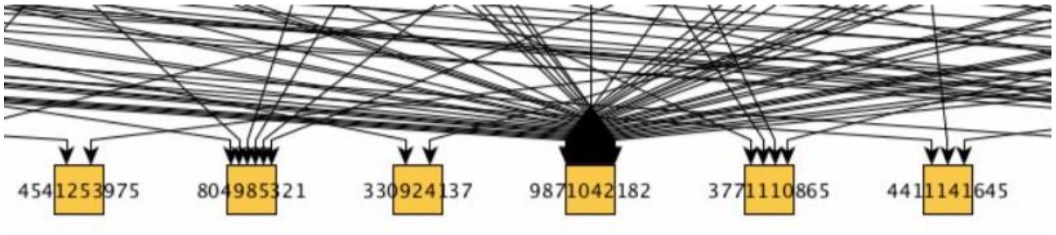
Complete graph:



Top portion of graph (RATs):



Bottom Portion of graph (stories):



Conclusion:

A. General Information

Total Users: 287

Total Stories: 609

Total Tweets: 6065

B. Investigative Questions

1. *Who was the most active RAT?*

   The most active RAT had the screen name of Jenn_Abrams. Jenn_Abrams was decided most active because they had the greatest number of stories (144) relating to the Russian government.

2. *Which was the* **mos**t *popular story among the RATs?*

   The most popular story corresponded to an identification number of 190816579. The story linked to this number was titled "Mistrial in the first trial of one of the officers accused of killing Freddie Gray". Popularity was determined by which story had been most frequently retweeted.

3. *What is the size of the* ***largest*** *connected component?*

   The size of the largest connected component was 896.

C. Additional Questions

1. *Is this graph connected or disconnected?*

   Since the value of the largest connected component was equal to the total number of vertices (total users and total stories combined), all of the users and stories must be connected to one another, resulting in a completely connected network/graph.

2. *Which was the* ***least*** *popular story among the RATs?*

The least popular story corresponded to an identification number of 961283756. The story linked to this number was titled “Leonard Nimoy died 1”. Least popularity was determined by which story had been least frequently retweeted.

3. *What is the size of the* ***smallest*** *connected component?*

    The size of the smallest connected component was also 896 because the graph was completely connected.

Code:

AdjListsGraph Class

```
/**
 * AdjListsGraph<T> implements the Graph<T> interface. It
 * also includes methods to perform a depth first search
 * and a depth first search traversal as well as a method
 * to safe the graph information to a tgf file.
 * @author mpapagel
 * @version 05/15/20
 */
import java.util.*;
import java.io.*;
import javafoundations.*;
public class AdjListsGraph<T> implements Graph<T>
{
   // instance variables
   private Vector<LinkedList<T>> arcs;
   private Vector<T> vertices;

   /**
    * Constructor for AdjListsGraph class
    */
   public AdjListsGraph(){
      arcs = new Vector<LinkedList<T>>();
      vertices = new Vector<T>();
   }

   /**
    * Returns a boolean indicating whether this graph is empty or not.
    * A graph is empty when it contains no vertice,and of course, no edges.
    * @return true if this graph is empty, false otherwise.
    */
   public boolean isEmpty(){
      return vertices.size() == 0;
   }

   /**
    * Getter method for number of vertices in the graph.
    * @return the number of vertices in this graph
    */
   public int getNumVertices(){
      return vertices.size();
   }
```

```
/**
 * Getter method for all the vertices in the graph.
 * @return vertices Vector<T> of all the vertices
 */
public Vector<T> getAllVertices(){
   return vertices;
}

/**
 * Returns the number of arcs in this graph.
 * An arc between vertices A and B exists, if a direct connection
 * from A to B exists.
 * @return the number of arcs in this graph
 *  */
public int getNumArcs(){
   int numArcs = 0;
   for( int  i= 0; i<arcs.size(); i++){
      numArcs+= arcs.get(i).size();
   }
   return numArcs;
}

/**
 * Returns true if an arc (direct connection) exists
 * from the first vertex to the second, false otherwise
 * @return true if an arc exists between the first given vertex (vertex1),
 * and the second one (vertex2),false otherwise
 *
 *  */
public boolean isArc (T vertex1, T vertex2){
   if (! vertices.contains(vertex1)){
      return false;
   }
   else {
      int index = vertices.indexOf(vertex1);
      return arcs.get(index).contains(vertex2);
   }
}

/**
 * Returns true if an edge exists between two given vertices, i.e,
 * an arch exists from the first vertex to the second one, and an arc from
 * the second to the first vertex, false otherwise.
 * @param vertex1 first given vertex
 * @param vertex2 second given vertex
 * @return true if an edge exists between vertex1 and vertex2,
 * false otherwise
 * */
public boolean isEdge (T vertex1, T vertex2){
   return isArc(vertex1, vertex2) && isArc(vertex2, vertex1);
}

/**
 * Returns true if the graph is undirected, that is, for every
 * pair of nodes i,j for which there is an arc, the opposite arc
 * is also present in the graph, false otherwise.
 * @return true if the graph is undirected, false otherwise
 * */
public boolean isUndirected(){
   for(int i=0; i<arcs.size(); i++){
      for(int j = 0; j<arcs.get(i).size(); j++)
         if (! isEdge((arcs.get(i).get(j)),(vertices.get(i)))){
            return false;
         }
   }
   return true;
}

/**
 * Adds the given vertex to this graph
 * If the given vertex already exists, the graph does not change
 * @param vertex the vertex to be added to this graph
 * */
public void addVertex (T vertex){
   if (!vertices.contains(vertex)){
      vertices.add(vertex);
      arcs.add(new LinkedList<T>());
   }
}

/**
 * Removes the given vertex from this graph.
 * If the given vertex does not exist, the graph does not change.
 * @param vertex the vertex to be removed from this graph
 *  */
public void removeVertex (T vertex){
   if (vertices.contains(vertex)){
      int vertexIndex = vertices.indexOf(vertex);
      vertices.remove(vertex);
      arcs.remove(vertexIndex);
      for(int i = 0; i<arcs.size(); i++){
         arcs.get(i).remove(vertex);
```

```
        }
    }

}

/**
 * Inserts an arc between two given vertices of this graph.
 * if at least one of the vertices does not exist, the graph
 * is not changed.
 * @param vertex1 the origin of the arc to be added to this graph
 * @param vertex2 the destination of the arc to be added to this graph
 *  */
public void addArc (T vertex1, T vertex2){
    if (vertices.contains(vertex1) && vertices.contains(vertex2)){
        int vertexIndex =vertices.indexOf(vertex1);
        arcs.get(vertexIndex).add(vertex2);
    }
}

/**
 * Removes the arc between two given vertices of this graph.
 * If one of the two vertices does not exist in the graph,
 * the graph does not change.
 * @param vertex1 the origin of the arc to be removed from this graph
 * @param vertex2 the destination of the arc to be removed from this graph
 * */
public void removeArc (T vertex1, T vertex2){
    if (isArc(vertex1, vertex2)){
        int vertexIndex = vertices.indexOf(vertex1);
        arcs.get(vertexIndex).remove(vertex2);
    }
}

/**
 * Inserts the edge between the two given vertices of this graph,
 * if both vertices exist, else the graph is not changed.
 * @param vertex1 the origin of the edge to be added to this graph
 * @param vertex2 the destination of the edge to be added to this graph
 *  */
public void addEdge (T vertex1, T vertex2){
    if(! isEdge (vertex1, vertex2)){
        addArc (vertex1,vertex2);
        addArc (vertex2, vertex1);
    }
}
```

```
/**
 * Removes the edge between the two given vertices of this graph,
 * if both vertices exist, else the graph is not changed.
 * @param vertex1 the origin of the edge to be removed from this graph
 * @param vertex2 the destination of the edge to be removed from this graph
 *
 */
public void removeEdge (T vertex1, T vertex2){
   if (isEdge (vertex1, vertex2)){
      removeArc(vertex1,vertex2);
      removeArc(vertex2,vertex1);
   }
}

/**
 * Return all the vertices, in this graph, adjacent to the given vertex.
 * @param vertex a vertex in the graph whose successors will be returned.
 * @return LinkedList containing all the vertices x in the graph,
 * for which an arc exists from the given vertex to x (vertex -> x).
 * */
public LinkedList<T> getSuccessors(T vertex){
   int vertexIndex = vertices.indexOf(vertex);
   return arcs.get(vertexIndex);
}

/**
 * Return all the vertices x, in this graph, that precede a given
 * vertex.
 * @param vertex a vertex in the graph whose predecessors will be returned.
 * @return LinkedList containing all the vertices x in the graph,
 * for which an arc exists from x to the given vertex (x -> vertex).
 * */
public LinkedList<T> getPredecessors(T vertex){
   LinkedList<T> predecessors = new LinkedList<T>();
   if (vertices.contains(vertex)){

      //for (int i = 0; i<vertices.indexOf(vertex); i++){
      int i = 0;
      while (vertices.get(i) != vertex );
      {
         predecessors.add(vertices.get(i));
      }
   }
   return predecessors;
}
```

```
/**
 * Returns a string representation of this graph.
 * @return a string represenation of this graph, containing its vertices
 * and its arcs/edges
 *  */
public String toString(){
   String result = "Verticies";
   result += vertices.toString();
   result += "Edges";
   for (int i = 0; i < arcs.size(); i++) {
      result  += "from" + vertices.elementAt(i) + ": "
      + arcs.elementAt(i) + "\n";
   }
   return result;
}

/**
 * Writes this graph into a file in the TGF format.
 * @param tgf_file_name the name of the file where this graph will be written
 * in the TGF format.
 * */
public void saveToTGF(String tgf_file_name){
   try{
      PrintWriter printer = new PrintWriter(new File(tgf_file_name));
      for (int i = 0; i<vertices.size(); i++){
         printer.println((i+1)+" "+vertices.get(i));
      }
      printer.println("#");
      for (int i = 0; i<arcs.size(); i++){
         for (int j = 0; j<arcs.get(i).size(); j++){
            printer.println((i+1)+" "+arcs.get(i).get(j));
         }
      }
   }
   catch(IOException e){
      System.out.println(e);
   }
}

/**
 * Performs a depth first search of the entire graph using a stack
 * starting at the given point with no specific end point.
 * @param T starting vertex for DFS traversal
 * @return LinkedList<T> containing the verticies that are the
 * outcome of the depth first search
 */
```

```
public LinkedList<T> depthFirstSearch(T vertex) {
    ArrayStack<T> stk = new ArrayStack<T>();
    LinkedList<T> result = new LinkedList<T>();
    boolean [] marked = new boolean[vertices.size()];
    for (int v = 0; v < vertices.size(); v++) {
        marked[v] = false; //mark each vertex as unvisited
    }
    T currentNode;
    int currentIndex;
    //push/add starting vertex into stack, then mark it as unvisited
    stk.push(vertex);
    result.add(vertex);
    marked[vertices.indexOf(vertex)] = true;
    while(!stk.isEmpty()) {
        currentNode = stk.peek();
        currentIndex = vertices.indexOf(currentNode);
        LinkedList<T> currentArcsList = arcs.get(currentIndex);
        int indexOfList = 0;
        while (indexOfList == currentArcsList.size()) {
          if (indexOfList == currentArcsList.size()){
              stk.pop();
            }
          else if (marked[vertices.indexOf(currentArcsList.get(indexOfList))] == false) {
             //push vertex into stack and add to result list if it hasn't been visited
             stk.push(currentArcsList.get(indexOfList));
             result.add(currentArcsList.get(indexOfList));
             marked[vertices.indexOf(currentArcsList.get(indexOfList))] = true;
             indexOfList = currentArcsList.size() + 1;
            }
          indexOfList++;
        }
    }
    return result;
}

/**
 * Performs a breadth first search of the entire graph
 * using a queue.
 * @param vertex starting vertex for BFS traversal
 * @return LinkedList<T> containing the verticies that are
 * the outcome of the breadth first search
 */
public LinkedList<T> breadthFirstSearch(T vertex){
    LinkedQueue<T> q = new LinkedQueue<T>();
    LinkedList<T> iterator = new LinkedList<T>();
    int count = 0;
    boolean[] marked = new boolean[vertices.size()];
    for (int v = 0; v < vertices.size(); v++){ // mark each vertex as unvisited
        marked[v] = false;
    }
    q.enqueue(vertex);
    marked[(vertices.indexOf(vertex))] = true;
    T current;
    int currentIndex;
    LinkedList<T> currentList;
    while (!q.isEmpty()){
        current = q.dequeue();
        currentIndex = vertices.indexOf(current);
        iterator.add(current);
        count++;
        currentList = arcs.get(currentIndex);
        for(int i = 0; i < currentList.size(); i++){
            T currentNodeInList = currentList.get(i);
            // enqueue vertex if it hasn't been visited and if arcs exist at from vertex (not null)
            if (!marked[vertices.indexOf(currentNodeInList)] && !(currentNodeInList ==null)){
                q.enqueue(currentNodeInList);
                marked[vertices.indexOf(currentNodeInList)] = true;
            }
        }
    }
    return iterator;
}

/**
 * Main method used for testing implementation on sample graphs.
 */
public static void main(String [] args) {
    AdjListsGraph<String> a = new AdjListsGraph<String>();
    a.addVertex("a");
    a.addVertex("b");
    a.addVertex("c");
    a.addVertex("d");
    a.addVertex("e");
    a.addArc("a", "b");
    a.addArc("b", "c");
    a.addArc("c", "a");
    a.addEdge("a", "c");
    a.removeVertex("e");
    System.out.println("Expected isArc() true: " + a.isArc("a", "b"));
    System.out.println("Expected isEdge() false: " + a.isEdge("a", "b"));
    System.out.println(a.depthFirstSearch("a"));
    System.out.println((a.breadthFirstSearch("a")));
    System.out.println(a.toString());
    a.saveToTGF("a.txt");

    AdjListsGraph<String> tree = new AdjListsGraph<String>();
    tree.addVertex("a");
    tree.addVertex("b");
    tree.addVertex("c");
    tree.addVertex("d");
    tree.addVertex("e");
    tree.addVertex("f");
    tree.addVertex("g");
    tree.addVertex("h");
    tree.addVertex("i");
    tree.addVertex("j");
    tree.addEdge("a","b");
    tree.addEdge("a","c");
    tree.addEdge("b","d");
    tree.addEdge("b","e");
    tree.addEdge("c","f");
    tree.addEdge("c","g");
    tree.addEdge("d","h");
    tree.addEdge("d","i");
    tree.addEdge("e","j");
    System.out.println("Tree BFS: a,b,c,d,e,f,g,h,i,j");
    System.out.println((tree.breadthFirstSearch("a")));
    System.out.println("Tree DFS: a,b,c,d,e,f,g,h,i,j");
    System.out.println((tree.depthFirstSearch("a")));
    System.out.println(tree.toString());
    tree.saveToTGF("Tree.tgf");

    AdjListsGraph<String> cycle = new AdjListsGraph<String>();
    cycle.addVertex("1");
    cycle.addVertex("2");
    cycle.addVertex("3");
    cycle.addVertex("4");
    cycle.addVertex("5");
    cycle.addEdge("1","2");
    cycle.addEdge("2","3");
    cycle.addEdge("3","4");
    cycle.addEdge("4","5");
    cycle.addEdge("5","1");
    System.out.println("Cycle BFS: a,b,c,d,e,f,g,h,i,j");
    System.out.println((cycle.breadthFirstSearch("1")));
    System.out.println("Cycle DFS: a,b,c,d,e,f,g,h,i,j");
    System.out.println((cycle.depthFirstSearch("1")));
```

```
        System.out.println(cycle.toString());
        cycle.saveToTGF("Cycle.tgf");

        AdjListsGraph<String> disconnected = new AdjListsGraph<String>();
        disconnected.addVertex("1");
        disconnected.addVertex("2");
        disconnected.addVertex("3");
        disconnected.addVertex("4");
        disconnected.addVertex("5");
        disconnected.addVertex("2");
        disconnected.addEdge("1","2");
        disconnected.addArc("2","3");
        disconnected.addEdge("3","4");
        System.out.println(disconnected.toString());
        System.out.println("Disconnected BFS: a,b,c,d,e,f,g,h,i,j");
        System.out.println((disconnected.breadthFirstSearch("1")));
        System.out.println("Disconnected DFS: a,b,c,d,e,f,g,h,i,j");
        System.out.println((disconnected.depthFirstSearch("1")));
        disconnected.saveToTGF("Disconnected.tgf");

    }
}
```

Rats Class

```
/**
 * Class reads in data from a csv file and creates an empty graph
 * to add data from the file into to produce a network, then saves it
 * as a .tgf file. This class also determines most and least
 * popular stories and their names as well as the longest and
 * shortest connected component within the graph.
 *
 * @author mpapagel
 * @version 05/15/20
 */
import java.io.*;
import java.util.*;
import java.net.URL;
public class Rats
{
    // instance variables
    private AdjListsGraph<String> RATgraph;
    private int allstories;
    private int allusers;
    private int alltweets;
    private Hashtable<String, Integer> storyTable;
    private String mostPopularStory;
    private String leastPopularStory;
    private String mostActiveRAT;
    private String urlFindStoryTitle =
"http://twittertrails.wellesley.edu/~trails/stories/title.php?id=";

    /**
     * Constructor for Rat class creates a Rats object
     * @param csvFile name of file to be read in
     */
    public Rats(String csvFile)
    {
        this.RATgraph = new AdjListsGraph<String>();
        this.storyTable = new Hashtable<String, Integer>();
        this.readRats(csvFile);
    }

    /**
     * Getter method returns total number of stories
     * @return allstories total number of stories
     */
    public int getAllStories() {
        return this.allstories;
```

```
}

/**
 * Getter method returns total number of tweets
 * @return alltweets total number of tweets
 */
public int getAllTweets() {
   return this.alltweets;
}

/**
 * Getter method returns total number of users
 * @return allusers total number of users
 */
public int getAllUsers() {
   return this.allusers;
}

/**
 * Method reads in the data from the csv file and creates
 * a graph to represent the networks created  by the Russian
 * accounts
 * @param csvFile name of file to be read in
 */
private void readRats(String csvFile) {
   try{
      Scanner scan = new Scanner(new File(csvFile));
      String header = scan.nextLine(); //skip header line
      while (scan.hasNextLine()) {
         String line = scan.nextLine();
         String[] allInfo = line.split("\t"); //separate categories of data in array
         //separate values in array into appropriate variables
         String screenName = allInfo[0];
         RATgraph.addVertex(screenName); //add screen names as vertices in graph
         String userid = allInfo[1];
         String tweetCount = allInfo[2];
         String storyCount = allInfo[3];
         //accumulate total number of tweets and users
         alltweets += Integer.parseInt(tweetCount);
         allusers++;
         //put stories into an array separating by commas
         String[] storiesArray = allInfo[4].split(",");
         //set variable for number of users who used a story
         int usedStory = 0;
         for (int i = 0; i < storiesArray.length; i++) {
            if (storyTable.containsKey(storiesArray[i])) {
                usedStory = storyTable.get(storiesArray[i]);
                //increment frequency of story in hashtable
                storyTable.put(storiesArray[i], usedStory++);
            } else {
                storyTable.put(storiesArray[i], 1); //if a story has a frequency of 1
                allstories++;
                //add stories to graph as vertices
                RATgraph.addVertex(storiesArray[i]);
            }
            //add edges between each screen name and its associated story to graph
            RATgraph.addEdge(screenName, storiesArray[i]);
        }
    }
  } catch(FileNotFoundException e) {
      System.out.println("File not found.");
  }
}

 /**
 * Getter method returns the RATgraph.
 * @return RATgraph graph of users and stories
 */
public AdjListsGraph<String> getGraph(){
  return RATgraph;
}

/**
 * Method finds the most and least popular stories,
 * by iterating through stories in the hashtable, and
 * sets the most/least popular instance variables to these
 * numbers without returning anything.
 */
public void popularity() {
  int largestSize = 0;
  int currentSize;
  int smallestSize = Integer.MAX_VALUE;
  for(String storyid: storyTable.keySet()) {
      currentSize = (RATgraph.getSuccessors(storyid)).size();
      if (currentSize > largestSize) {
          this.mostPopularStory = storyid;
          largestSize = currentSize;
      }
      if (currentSize < smallestSize) {
          this.leastPopularStory = storyid;
          smallestSize = currentSize;
      }
    }
}

/**
 * Getter method returns the number of the most popular story.
 * @return mostPopularStory most popular story
 */
public String getMostPopStory() {
    return this.mostPopularStory;
}

/**
 * Getter method returns the number of the least popular story.
 * @return leastPopularStory least popular story
 */
public String getLeastPopStory() {
    return this.leastPopularStory;
}

 /**
 * Method that takes in the identification number of a story
 * as a parameter and and returns the story's title using a URL
 * search to find it in the TwitterTrails database
 * @param storyId identification number of the story
 * @return storyTitle title of the story
 */
public String getStoryTitle(String storyId) {
    String storyTitle = "";
    try {
        URL url = new URL(urlFindStoryTitle + storyId);
        Scanner scan = new Scanner(url.openStream());
        scan.nextLine();
        storyTitle = scan.nextLine();
    } catch(IOException e){
        System.out.println(e);
    }
    return storyTitle;
}

/**
 * Getter method that returns the screen name of the user who participated
 * in the most stories
 * @return mostActiveRat the screen name of the most active RAT
 */
public String getMostActiveRAT() {
    return this.mostActiveRAT;
```

```
}

/**
 * Method returns the size of the largest connected component
 * in the graph
 * @return largestSize size of the LCC
 */
public int findLCC() {
    Vector<String> vertices = RATgraph.getAllVertices();
    int largestSize = 0;
    int currentSize;
    LinkedList<String> dfsList = new LinkedList<String>();
    for (int i = 0; i < vertices.size(); i++) {
        dfsList = RATgraph.depthFirstSearch((vertices.elementAt(i)));
        currentSize = dfsList.size();
        if (currentSize > largestSize) {
            largestSize = currentSize;
        }
    }
    return largestSize;
}

/**
 * Method returns the size of the smallest connected component
 * in the graph
 * @return smallestSize size of the SCC
 */
public int findSCC() {
    Vector<String> vertices = RATgraph.getAllVertices();
    int smallestSize = Integer.MAX_VALUE;
    int currentSize;
    for (int i = 0; i < vertices.size(); i++) {
       LinkedList<String> bfsList = RATgraph.breadthFirstSearch((vertices.elementAt(i)));
        currentSize = bfsList.size();
        if (currentSize < smallestSize) {
            smallestSize = currentSize;
        }
    }
    return smallestSize;
}

/**
 * Method returns true if the graph is completely connected.
 * @return true if the graph is completely connected
 */
public boolean isConnected() {
        return this.findLCC() == RATgraph.getNumVertices();
    }

    /**
     * main method for testing
     */
    public static void main(String[] args){
        //create new collection of RATs using given csv file
        Rats rats = new Rats("All_Russian-Accounts-in-TT-stories.csv");
        //save collection of RATs to a .tgf
        rats.getGraph().saveToTGF("RATgraph.tgf");
        //print general information regarding the Russian data
        System.out.println("Total Users: " + rats.getAllUsers());
        System.out.println("Total Stories: " + rats.getAllStories());
        System.out.println("Total Tweets: " + rats.getAllTweets());
        //determine most active RAT
        System.out.println("Most Active RAT: "+ rats.getMostActiveRAT());
        //find most/least popular stories
        rats.popularity();
        String mostPopular = rats.getMostPopStory();
        String leastPopular = rats.getLeastPopStory();
        System.out.println("Most Popular Story: " + rats.getMostPopStory() + " " +
rats.getStoryTitle(mostPopular));
        System.out.println("Least Popular Story: " + rats.getLeastPopStory() + " " +
rats.getStoryTitle(leastPopular));
        //find LCC and SCC
        System.out.println("LCC size: " + rats.findLCC());
        System.out.println("SCC size: " + rats.findSCC());
        //find total vertices and see if graph is completely connected
        System.out.println("total vertices: " + rats.getGraph().getNumVertices());
        System.out.println("Is this graph completely connected? " + rats.isConnected());

    }
}
```

Investigate Class

```
/**
 * Driver class creates a collection of rats from the given
 * csv file and runs methods from Rats class to investigate
 * data from the Russian accounts csv file and depicted graph.
 *
 * @author mpapagel
 * @version 05/15/20
 */
public class Investigate
{
   public static void main(String[] args){
       //create new collection of RATs using given csv file
       Rats rats = new Rats("All_Russian-Accounts-in-TT-stories.csv");
       //save collection of RATs to a .tgf
       rats.getGraph().saveToTGF("RATgraph.tgf");
       //print general information regarding the Russian data
       System.out.println("Total Users: " + rats.getAllUsers());
       System.out.println("Total Stories: " + rats.getAllStories());
       System.out.println("Total Tweets: " + rats.getAllTweets());
       //determine most active RAT
       System.out.println("Most Active RAT: "+ rats.getMostActiveRAT());
       //find most/least popular stories
       rats.popularity();
       String mostPopular = rats.getMostPopStory();
       String leastPopular = rats.getLeastPopStory();
       System.out.println("Most Popular Story: " + rats.getMostPopStory() + " " +
rats.getStoryTitle(mostPopular));
       System.out.println("Least Popular Story: " + rats.getLeastPopStory() + " " +
rats.getStoryTitle(leastPopular));
       //find LCC and SCC
       System.out.println("LCC size: " + rats.findLCC());
       System.out.println("SCC size: " + rats.findSCC());
       //find total vertices and see if graph is completely connected
       System.out.println("total vertices: " + rats.getGraph().getNumVertices());
       System.out.println("Is this graph completely connected? " + rats.isConnected());
   }
}
```