CS 230
Final Project
December 9th, 2019

Russian Accounts on Twitter: A Study Using TwitterTrails Data

Hannah Kim & Katherine Guo

Introduction

Social media has been of crucial importance in diverse aspects of our daily lives including political campaigns. Due to its nature, contents may be spread, or “re-posted”, with no significant third party filtering, fact-checking, or editorial judgment.[1] It is a platform where individuals receive attention and popularity more easily compared to traditional newspapers or journals, a characteristic that may turn out to be positive or negative in different situations.

In October 2016, the Obama administration officially accused Russia of attempting to interfere in the 2016 elections.[2] In addition, the Russian government took advantage of various social media, including Facebook and Twitter, to interfere with the United States presidential elections to harm Hilary Clinton’s campaign and boost Donald Trump’s candidacy.[3]

Twitter has reported a number of automated Russian-based accounts that generate contents potentially related to Russian propaganda activities. In Twitter’s supplemental analysis, a total of 50,258 automated accounts were identified as Russian-linked and to have tweeted election-related content during the election period, which “[poses] a challenge to democratic societies everywhere”.[4]

In particular, we are seeking to investigate the activities conducted by some Twitter accounts that were identified as Russian-operated and attempted to interfere with the election. They are referred to as “Russian Accounts on Twitter (RATS)”. Through interacting and helping propagandas that were either subtly or overtly pro-Trump and against Clinton, these RAT accounts attempt to instigate and stir up anti-Clinton sentiments. This often involved stoking both sides of a political controversy at the same time, as well as capitalizing on current news, events, and hashtags for maximum efficacy.[5]

[1] Hunt Allcott and Matthew Gentzkow, “Social Media and Fake News in the 2016 Election,” *Journal of Economic Perspectives* 31, no. 2 (May 2017): 211–36, https://doi.org/10.1257/jep.31.2.211.

[2] Ellen Nakashima, “U.S. Government Officially Accuses Russia of Hacking Campaign to Interfere with Elections,” *Washington Post*, October 7, 2016, sec. National Security, https://www.washingtonpost.com/world/national-security/us-government-officially-accuses-russia-of-hacking-campaign-to-influence-elections/2016/10/07/4e0b9654-8cbf-11e6-875e-2c1bfe943b66_story.html.

[3] Scott Shane, “The Fake Americans Russia Created to Influence the Election,” *The New York Times*, September 7, 2017, sec. U.S., https://www.nytimes.com/2017/09/07/us/politics/russia-facebook-twitter-election.html.

[4] “Update on Twitter’s Review of the 2016 US Election,” accessed November 24, 2019, https://blog.twitter.com/en_us/topics/company/2018/2016-election-update.html.

[5] “Twitter Released 9 Million Russian Troll Tweets. Here’s What We Know. - Vox,” accessed December 4, 2019, https://www.vox.com/2018/10/19/17990946/twitter-russian-trolls-bots-election-tampering.

Methods

We examined a file ("All_Russian-Accounts-in-TT-stories.csv.tsv") containing information about the screen names, user ID's, tweet count, story count, and the identification numbers of stories for each RAT. Java was the coding language of use.

The major data structure that we used was AdjacencyListGraph, which implemented the Graph interface and used various other data structures such as LinkedQueue and LinkedStack. Also, in the application class used to investigate the file, we additionally used Vectors to maintain collections of stories and the Hashtable to keep a record of users as keys and their stories as values.

We wrote methods for Depth-First Search (DFS) and Breadth-First Search (BFS) using an iterative implementation. Both returns an iterator that consists of vertices in the order in which the graph is searched through. We could tell whether the graph was connected using DFS; if the returned iterator contained all the vertices, the graph would be a connected graph. Also, by using BFS, we could figure out how many layers there are in the network of RAT users and stories. We created an AdjacencyListsGraph object, in which each vertex is either a user or a story and the edges are links between a user and a story, and saved it to a Trivial Graph Format (TGF) in the application class. The user ID was saved with the letter "U" in the front, followed by the actual user ID only consisting of numbers.

The resulting bipartite graph was saved to a file named "RATGraph.tgf". There are 896 vertices and 7160 edges in total. The graph can be viewed either as a text file using a text editor or as a graph using the graph editor software yEd. In the latter case, the graphic visualizes RAT users on one side and all the stories on the other, with edges occurring only between these two groups but not within.

Some research questions we seek to explore using our program are:

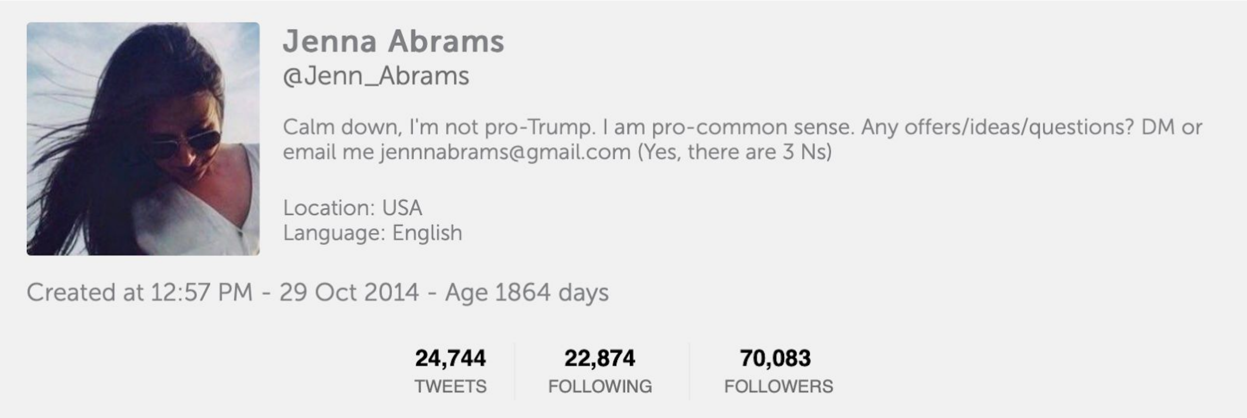
1. Who was the most active RAT?
2. Which story was the most popular among RATs?
3. What is the diameter of the largest connected component (LCC) in RATgraph?
4. What is the depth of the largest connected component (LCC)?

Conclusions

Research Question #1: Who was the most active RAT?

We found the most active RAT by searching through the Vector of RATs in the RATGraph class and comparing the number of stories of each RAT. The most active RAT was @Jenn_Abrams (user_ID: 2882331822), who posted 144 stories in total.

The account (profile snapshot shown below) is currently suspended,[6] but below is what the account used to look like in the past:



Jenna Abrams
@Jenn_Abrams

Calm down, I'm not pro-Trump. I am pro-common sense. Any offers/ideas/questions? DM or email me jennabrams@gmail.com (Yes, there are 3 Ns)

Location: USA
Language: English

Created at 12:57 PM - 29 Oct 2014 - Age 1864 days

24,744 TWEETS | 22,874 FOLLOWING | 70,083 FOLLOWERS

Figure 1. Snapshot of the most active account's profile, before it was suspended

Research Question #2: Which was the most popular story among RATs?

For the most popular story among RATs, our program yields the story 190816579, a story named "Mistrial in the first trial of one of the officers accused of killing Freddie Gray". The story appeared in the database for 53 times, which means 53 RATS reacted to this particular tweet.

We investigated the story on TwitterTrails. Freddie Gray is a 25-year-old black man who sustained a fatal spinal injury while in police custody. His death case led to lawsuits, protests, and major online discussions.[7]

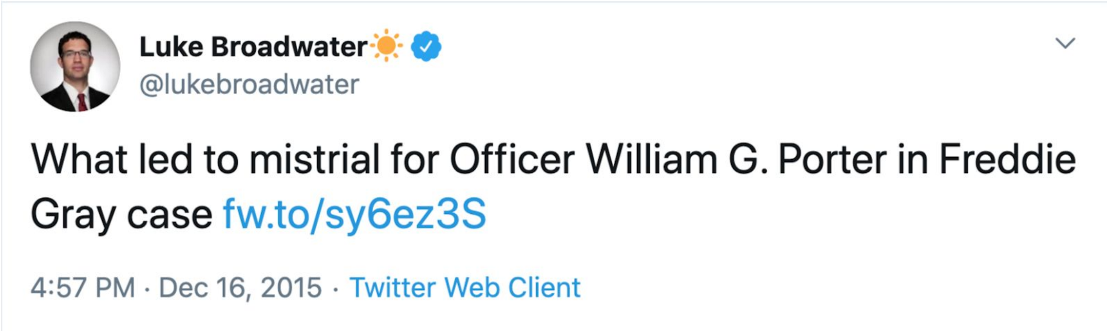
TwitterTrails explores the story based on a tweet about the mistrial in the first trial of Officer William G. Porter, who was accused of killing Freddie Gray.[8] It was posted on December 16th,

[6] "Profile / Twitter," Twitter, accessed December 4, 2019, https://twitter.com/jenn_abrams.

[7] Rebecca R. Ruiz, "Baltimore Officers Will Face No Federal Charges in Death of Freddie Gray," *The New York Times*, September 12, 2017, sec. U.S., https://www.nytimes.com/2017/09/12/us/freddie-gray-baltimore-police-federal-charges.html.

[8] "Mistrial in the First Trial of One of the Officers Accused of Killing Freddie Gray - TwitterTrails," accessed December 4, 2019, http://twittertrails.wellesley.edu/~trails/stories/investigate.php?id=190816579.

2015 by Luke Broadwater(@lukebroadwater), a reporter based in Baltimore, Maryland, where the Freddie Gray case took place.



Luke Broadwater☀
@lukebroadwater

What led to mistrial for Officer William G. Porter in Freddie Gray case fw.to/sy6ez3S

4:57 PM · Dec 16, 2015 · Twitter Web Client

Figure 2. The tweet retweeted the most times[9]

Its co-retweeted network is shown in Figure 3. There are a total of 49 communities of similar users in the co-retweeted Network. The largest community has 1,131 users in it, and the smallest has 2. Nodes in the co-retweeted graph are colored based on their community.

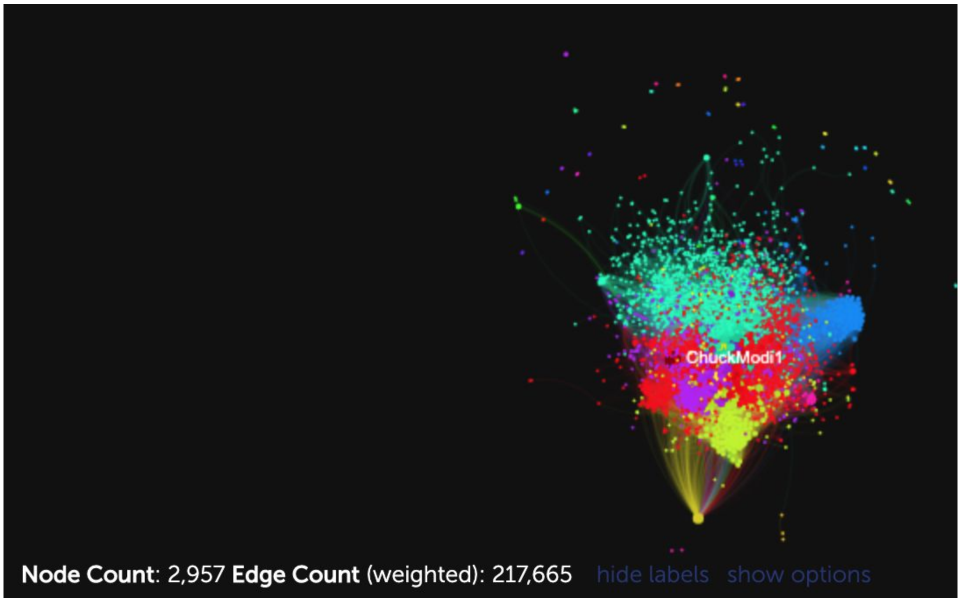


Figure 3. The co-retweeted network for the most popular story

Each community is also represented by a word cloud in Figure 4. The more often the members of a particular community use a word in their profile, the larger that word appears in the word cloud. The largest group is the cyan group, which appears to be conservatives, while the second one that follows — the red group — may be liberals. The purple group seems to be locals since they used words like “Baltimore” and “Washington” to describe themselves. These

[9] “ Luke Broadwater☀ on Twitter: ‘What Led to Mistrial for Officer William G. Porter in Freddie Gray Case Https://T.Co/CpWlf4pe9s’ / Twitter,” Twitter, accessed December 4, 2019, https://twitter.com/lukebroadwater/status/677246260995366912.

different groups imply that the co-retweeters involved in the same story do not necessarily share the same identity and thus the same stance. On the contrary, they might even have polarized views, such as the red and cyan groups.

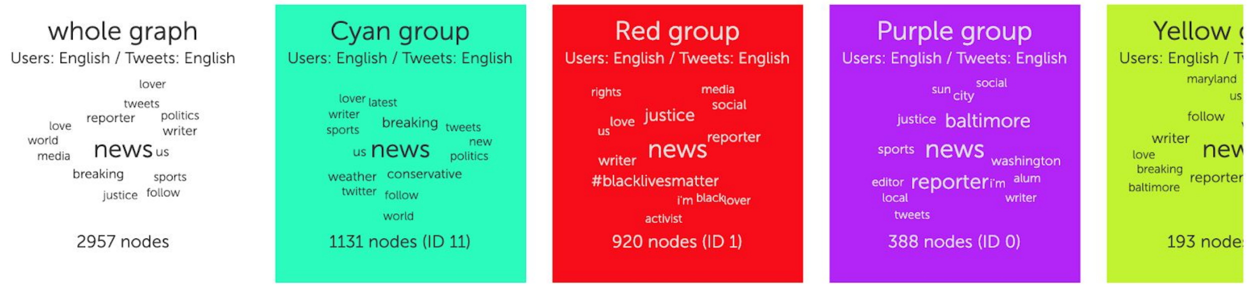


Figure 4. Some representative word clouds in the co-retweeted network

<u>Research Question #3: What is the diameter of the largest connected component in the RATgraph?</u>

We ran Depth-First Search (DFS) on RATgraph.tgf starting arbitrarily on the first index. Then, we compared the number of elements in the ArrayIterator returned by the DFS with the number of vertices in RATgraph.tgf. The resulting boolean is true, which means that the diameter of the largest connected component is equal to the total number of vertices (including all users and stories) in the graph (896). This implies that the entire RATgraph is a single connected component, and thus is the largest connected component.

The entire RAT network is connected: there is a path between any two nodes, which leads to our finding that there is a path between any two users or stories.

<u>Research Question #4: What is the depth of the largest connected component?</u>

Drawing on the fact that the graph is connected in its entirety, we used the breadth-first-search approach to find out the maximum number of steps needed to get from any one of the vertices to another. In other words, we counted how many layers we pass through starting from a vertex and getting through to the end after traversing all levels of vertices in between.

We would have needed to loop through all the vertices and run the traversal from every vertex had we not known that the graph was connected, but since we did, we just started the traversal arbitrarily from the first vertex. Another critical assumption that we knew to be true was that the first letter of all vertices representing users was “U”.

Using two booleans to keep track of the chunks of users and stories, we learned that there are seven layers and thus six steps in the graph. This is also consistent with the idea of the Six Degrees of Kevin Bacon that the farthest distance between any one vertex to another is six.

Since it takes six steps at maximum to start at any one vertex and get to another, we come to the conclusion that the network is quite closely connected and that there is no one central node; every component of the graph is similar in terms of centrality within the graph.

Collaboration

Throughout the project we worked together on coding, conducting research, and finalizing the report. We completed the coding section together, operating on the same computer. Hannah's focus was on Investigate.java and Katherine's focus was on AdjListsGraph.java, but we mostly edited together and cross-checked each other's work.

When writing the analysis, Katherine focused on Research Questions #2 and #3, and Hannah focused on Research Questions #1 and #4. We also both engaged in supplementing details to each other's sections.

Overall, we closely collaborated and shared all the responsibilities.

Code

```
/**
 * ArrayIterator.java
 * We added 2 methods to the
javafoundations version. Everything
else remains unchanged.
 * @author yguo2 hkim22
 * @version Dec 2019
 */

   /**
    * Return the count of the
ArrayIterator
    *
    * @return  the number of elements
in the iterator
    */
   public int size(){
       return count;
   }

   /**
    * Return element at a given index
    *
    * @param index the index of the
element to be found
    * @return the element at the given
index
    */
   public T elementAt(int index) {
       return items[index];
   }


/**
 * AdjListsGraph.java
 * The AdjListsGraph class implements
the Graph interface.
 * It represents an AdjListsGraph
object, which is a collection of
unordered lists used to represent a
finite graph.
 *
 * @author yguo2, hkim22
 * @version 11/20/2019
 */

package javafoundations;
import java.io.*;
import java.lang.Exception;
import java.util.*;

public class AdjListsGraph<T>
implements Graph<T>
{
    private Vector<T> vertices;
    private Vector<LinkedList<T>> arcs;

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

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

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

    /**
     * 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 num = 0;
        for (int i = 0; i <
arcs.size(); i ++) {
            num += arcs.get(i).size();
        }
        return num;
    }

    /**
     * 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)
&& vertices.contains(vertex2)) {
            return
(arcs.get(vertices.indexOf(vertex1))).c
ontains(vertex2);
        }
        return false;
    }

    /**
     * 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.
     * @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(vertices.get(i),
arcs.get(i).get(j))){
                    return false;
                }
            }
        }
        return true;
    }

    /**
     * Adds the given vertex to this
graph
     * If the given vertex already
exists, the graph does not change
     * @param 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 the vertex to be removed
from this graph
     *  */
    public void removeVertex (T
vertex){
        if (vertices.contains(vertex)){
            int index =
vertices.indexOf(vertex); //index of
vertex to be removed in vertices
            arcs.remove(index);
//remove predecessors of vertex
            for (int i = 0;
i<arcs.size(); i++){
                if
(arcs.get(i).contains(vertex)){ //if
other vertices have an arc to vertex
                    int j =
arcs.get(i).indexOf(vertex);

arcs.remove(arcs.get(i).get(j));
                }

vertices.remove(vertex); //removes
vertex frm vertices
            }
        }
    }

    /**
     * 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 the origin of the arc to
be added to this graph
     * @param 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 index1 =
vertices.indexOf(vertex1);
            //int index2 =
vertices.indexOf(vertex2);

arcs.get(index1).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 the origin of the arc to
be removed from this graph
     * @param the destination of the
arc to be removed from this graph
     * */
    public void removeArc (T vertex1, T
vertex2) {
        if (vertices.contains(vertex1)
&& vertices.contains(vertex2) &&
isArc(vertex1, vertex2)) {
            int index1 =
vertices.indexOf(vertex1);
            int index2 =
vertices.indexOf(vertex2);

arcs.get(index1).remove(index2);
        }
    }

    /**
     * Inserts the edge between the two
given vertices of this graph,
     * if both vertices exist, else the
graph is not changed.
     * @param the origin of the edge to
be added to this graph
     * @param the destination of the
edge to be added to this graph
     **/
    public void addEdge (T vertex1, T
vertex2) {
        addArc(vertex1, vertex2); //add
the edge from vertex 1 to vertex 2
        addArc(vertex2, vertex1); //add
the edge from vertex 2 to vertex 1
    }

    /**
     * Removes the edge between the two
given vertices of this graph, ifboth
vertices exist, else the graph is not
changed.
     * @param the origin of the edge to
be removed from this graph
     * @param the destination of the
edge to be removed from this graph
     */
    public void removeEdge (T vertex1,
T vertex2) {
        removeArc(vertex1, vertex2);
//remove the edge from vertex 1 to
vertex 2
        removeArc(vertex2, vertex1);
//remove the edge from vertex 2 to
vertex 1
    }

    /**
     * Return all the vertices, in this
graph, adjacent to the given vertex.
     * @param 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) {
        if (vertices.contains(vertex)){
            return
arcs.get(vertices.indexOf(vertex));
        }
        return null;
    }

    /**
     * Return all the vertices x, in
this graph, that precede a given
     * vertex.
     * @param 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> temp = new
LinkedList<T>();
        if (vertices.contains(vertex)){
            for (int i = 0; i <
vertices.size(); i++){
                if
(isArc(vertices.get(i), vertex)){

temp.add(vertices.get(i));
                }
            }
            return temp;
        }
        return null;
    }

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

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

writer.println((i+1) + " " + 
(vertices.indexOf(arcs.get(i).get(j))+1
));
                }
            }
            writer.close();
        } catch (IOException ex) {
            System.out.println(ex);
        }
    }

    /**
     * Runs a breadth-first traversal 
(BFS)
     * @param v index of the starting 
vertex
     * @return the iterator containing 
the order traversed using BFS
     */
    public ArrayIterator<T> BFS(int v)
    {
        int currentVertex;
        // Create a queue
        LinkedQueue<Integer> 
traversalQueue = new 
LinkedQueue<Integer>();
        ArrayIterator<T> iter = new 
ArrayIterator<T>();

        // If the starting vertex is 
invalid, returns null
        if (!(v<vertices.size() || 
v<0)){
            return null;
        }

        // Mark all the vertices as not 
visited(false by default)
        boolean visited[] = new 
boolean[vertices.size()];

        traversalQueue.enqueue(v); // 
add the starting vertex to the queue
        // Mark the starting vertex as 
visited
        visited[v] = true;

        // Procedure
        while 
(!traversalQueue.isEmpty())
        {
            currentVertex = 
traversalQueue.dequeue();

iter.add(vertices.elementAt(currentVert
ex));
            for (int vertexIndex = 0; 
vertexIndex < vertices.size(); 
vertexIndex++)
                if 
(isEdge(vertices.elementAt(currentVerte
x),vertices.elementAt(vertexIndex)) &&
                !visited[vertexIndex])
                {

traversalQueue.enqueue(vertexIndex);

visited[vertexIndex] = true;
                }
        }
        return iter;
    }

    /**
     * Runs a depth-first traversal 
(DFS)
     * @param v index of the starting 
vertex
     * @return the iterator containing 
the order traversed using DFS
     */
    public ArrayIterator<T> DFS(int v)
    {
        int currentVertex;
        LinkedStack<Integer>
traversalStack = new
LinkedStack<Integer>();
        ArrayIterator<T> iter = new
ArrayIterator<T>();
        boolean[] visited = new
boolean[vertices.size()];
        boolean found;
        // If the starting vertex is
invalid, returns null
        if (!(v<vertices.size() ||
v<0)){
            return null;
        }
        traversalStack.push(v);
        iter.add
(vertices.elementAt(v));
        visited[v] = true;

        while
(!traversalStack.isEmpty())
        {
            currentVertex =
traversalStack.peek();
            found = false;
            for (int i = 0; i <
vertices.size() && !found; i++)
                if
(isEdge(vertices.elementAt(currentVerte
x),vertices.elementAt(i)) && !visited
[i]) {

traversalStack.push(i);

iter.add(vertices.elementAt(i));
                    visited[i] = true;
                    found = true;
                }
            if (!found
&& !traversalStack.isEmpty())
                traversalStack.pop();
        }
        return iter;
    }

    /**
     * Main driver of the test
     */
    public static void main (String[]
args){
        // Testing on a cycle (C5)
        AdjListsGraph<String> g1 = new
AdjListsGraph();

        g1.addVertex("A");
        g1.addVertex("B");
        g1.addVertex("C");
        g1.addVertex("D");
        g1.addVertex("E");
        g1.addEdge("A","B");
        g1.addEdge("B","C");
```

```
        g1.addEdge("C","D");
        g1.addEdge("D","E");
        g1.addEdge("E","A");

        // Make a tgf file for C5
        g1.saveToTGF("Cycle.tgf");

        // System.out.println(g);
        //
System.out.println(g.getSuccessors("A")
);
        //
System.out.println(g.getSuccessors("C")
);
        //
System.out.println(g.getPredecessors("B
"));

        //System.out.println(g.BFS(2));
        //System.out.println(g.DFS(2));

        // g.saveToTGF("test.tgf");
        // g.saveToTGF("test2.tgf");

        // Testing on a tree
        AdjListsGraph<String> g2 = new
AdjListsGraph();
        g2.addVertex("A");
        g2.addVertex("B");
        g2.addVertex("C");
        g2.addVertex("D");
        g2.addVertex("E");
        g2.addVertex("F");
        g2.addVertex("G");
        g2.addEdge("A","B");
        g2.addEdge("A","C");
        g2.addEdge("B","D");
        g2.addEdge("B","E");
        g2.addEdge("C","F");
        g2.addEdge("C","G");

        g2.saveToTGF("Tree.tgf");

        // Testing on a path P4
        AdjListsGraph<String> g3 = new
AdjListsGraph();

        g3.addVertex("A");
        g3.addVertex("B");
        g3.addVertex("C");
        g3.addVertex("D");
        g3.addEdge("A","B");
        g3.addEdge("B","C");
        g3.addEdge("C","D");

        g3.saveToTGF("Path.tgf");

        // Testing on a disconnected
graph
        AdjListsGraph<String> g4 = new
AdjListsGraph();
```

```
        g4.addVertex("A");
        g4.addVertex("B");
        g4.addVertex("C");
        g4.addVertex("D");
        g4.addEdge("A","B");
        g4.addEdge("B","C");
        g4.addEdge("C","A");


g4.saveToTGF("Disconnected.tgf");

        // Testing on a bipartite graph
K2,3
        AdjListsGraph<String> g5 = new
AdjListsGraph();

        g5.addVertex("A");
        g5.addVertex("B");
        g5.addVertex("C");
        g5.addVertex("D");
        g5.addVertex("E");
        g5.addEdge("A","C");
        g5.addEdge("A","D");
        g5.addEdge("A","E");
        g5.addEdge("B","D");
        g5.addEdge("B","C");
        g5.addEdge("B","E");

        g5.saveToTGF("Bipartite.tgf");

        // Testing on a single vertex
        AdjListsGraph<String> g6 = new
AdjListsGraph();
        g6.addVertex("A");

        g6.saveToTGF("Single.tgf");

        // Testing on a complete graph
K
        AdjListsGraph<String> g7 = new
AdjListsGraph();

        g7.addVertex("A");
        g7.addVertex("B");
        g7.addVertex("C");
        g7.addVertex("D");
        g7.addVertex("E");
        g7.addEdge("A","B");
        g7.addEdge("A","C");
        g7.addEdge("A","D");
        g7.addEdge("A","E");
        g7.addEdge("B","D");
        g7.addEdge("B","C");
        g7.addEdge("B","E");
        g7.addEdge("C","D");
        g7.addEdge("C","E");
        g7.addEdge("D","E");

        g7.saveToTGF("Complete.tgf");
```

```
    }
}

/**
 * Investigate.java
 * This class investigates the RATs and
uses AdjListsGraph.java.
 *
 * @author yguo2, hkim22
 * @version 11/20/2019
 */

import javafoundations.*;
import java.util.*;
import java.io.*;


public class Investigate
{
    // instance variables
    // Compose a graph
    private AdjListsGraph<String>
RATAdjGraph;

    // the key is userID (a String) and
the value is a LinkedList of stories
    private Hashtable<String,
LinkedList<String>> RAT;
    private Vector<String>
storyCollection;
    /**
     * Constructor for objects of class
Investigate
     */
    public Investigate()
    {
        // initialise instance
variables
        RATAdjGraph = new
AdjListsGraph<String>();
        RAT = new Hashtable<String,
LinkedList<String>>();
        storyCollection = new
Vector<String>();
    }

    /**
     * Read from a file that contains
information of the RATS
     *
     * @param file_name  the file that
contains RAT information
     */
    private void readFile(String
file_name){
        try {
            Scanner fileScan = new
Scanner (new File(file_name));
            String firstLine =
fileScan.nextLine(); // the first line
is the header
```

```
            while (fileScan.hasNext())
{
                String line =
fileScan.nextLine();
                String[] info =
line.split("\t");

                String user_id =
info[1];
                String stories =
info[4];


RATAdjGraph.addVertex(user_id);
                RAT.put(user_id,
divideStories(stories));

//System.out.println(user_id);  //
Testing
            }
            fileScan.close();
        } catch (IOException ex) {
            System.out.println(ex);
        }
    }

    /**
     * Helper method
     * Divide the stories from a
LinkedList add them to the graph as
vertices
     * @param  stories   a string of
stories
     * @return  a linked list of all
the stories
     */
    private LinkedList<String>
divideStories(String stories){
        String[] story =
stories.split(",");
        LinkedList<String> result = new
LinkedList<String>();
        for (int i = 0; i<story.length;
i++){
            result.add(story[i]);

RATAdjGraph.addVertex(story[i]);

storyCollection.add(story[i]); // add
each individual story to the collection
of stories
        }
        return result;
    }

    /**
     * Add edges to the graph
     */
    private void makeEdges(){
        for (String user:RAT.keySet())
{
            for (String
story:RAT.get(user)){

RATAdjGraph.addEdge(user, story);
            }
        }
    }

    /**
     * Save graph to TGF file
     * @param tgf_file_name The name of
the output file
     */
    private void makeTGF(String
tgf_file_name){

RATAdjGraph.saveToTGF(tgf_file_name);
    }

    /**
     * Find and return the most active
RAT object
     * Answers Research Question #1
     * @return a string of the userID
of the most active RAT
     */
    private String findActive() {
        int max = 0;
        String result = "";
        for (String user:RAT.keySet())
{
            if
(RAT.get(user).size()>max) {
                max =
RAT.get(user).size();
                result = user;
            }
        }
        return result;
    }

    /**
     * Find and return the most popular
story
     *
     * Answers Research Question #2
     * @return a string of the
identification number of the most
popular story
     */
    private String findPopular() {
        int max = 0;
        String result = "";
        for (String
story:storyCollection) {
            int num =
RATAdjGraph.getSuccessors(story).size()
;
            if (num>max){
                max = num;
                result = story;
            }
        }
        return result;
    }

    /**
     * Runs DFS on the
RATSgraph(AdjListGraph) and test if the
graph is connected.
     * Answers Research Question #3
     *
     * @return true if the diameter of
the largest connected components is
equal to the number of total vertices
     *         false otherwise
     */
    private boolean runDFS(){
        // Pick a random vertex (here
we pick 0) as the starting vertex, and
run DFS on it
        return
(RATAdjGraph.DFS(0).size() ==
RATAdjGraph.getNumVertices());
    }

    /**
     * Use the BFS method in
AdjListsGraph.java to find out how many
layers the LCC have
     *
     * Answers Research Question #4
     * @return  number of layers in the
LCC
     */
    private int runBFS(){
        ArrayIterator<String> iter =
RATAdjGraph.BFS(0);
        int result = 0;
        boolean isU = false;
        boolean isStory = false;
        for (int i = 0; i<iter.size();
i++) {
            String e =
iter.elementAt(i);
            if
(e.substring(0,1).equals("U") && !isU)
{
                result += 1;
                isU = true;
                isStory = false;
            } // the beginning of the
series of users
            else if
(!e.substring(0,1).equals("U")
&& !isStory) {
                result += 1;
                isStory = true;
                isU = false;
            } // the beginning of the
series of stories
        }
```

```
        return result;
    }

   /**
     * Main driver of the test
     */
    public static void main (String[]
args){
        Investigate ratTest = new
Investigate();

        ratTest.readFile("All_Russian-
Accounts-in-TT-stories.csv.tsv");
        ratTest.makeEdges();

ratTest.makeTGF("RATgraph.tgf");


//ratTest.makeTGF("RATdirected.tgf");

System.out.println(ratTest.findActive()
);

System.out.println(ratTest.findPopular(
));

        System.out.println("The
RATgraph is connected: " +
ratTest.runDFS());
        System.out.println("Number of
layers in LCC: " + ratTest.runBFS());
    }
}
```

References

Allcott, Hunt, and Matthew Gentzkow. 2017. “Social Media and Fake News in the 2016 Election.” *Journal of Economic Perspectives* 31 (2): 211–36. https://doi.org/10.1257/jep.31.2.211.

Nakashima, Ellen. 2016. “U.S. Government Officially Accuses Russia of Hacking Campaign to Interfere with Elections.” *Washington Post*, October 7, 2016, sec. National Security. https://www.washingtonpost.com/world/national-security/us-government-officially-accuses-russia-of-hacking-campaign-to-influence-elections/2016/10/07/4e0b9654-8cbf-11e6-875e-2c1bfe943b66_story.html.

Shane, Scott. 2017. “The Fake Americans Russia Created to Influence the Election.” *The New York Times*, September 7, 2017, sec. U.S. https://www.nytimes.com/2017/09/07/us/politics/russia-facebook-twitter-election.html.

“Twitter Released 9 Million Russian Troll Tweets. Here’s What We Know. - Vox.” n.d. Accessed December 4, 2019. https://www.vox.com/2018/10/19/17990946/twitter-russian-trolls-bots-election-tampering.

Twitter. “(20) Luke Broadwater☀ on Twitter: ‘What Led to Mistrial for Officer William G. Porter in Freddie Gray Case Https://T.Co/CpWlf4pe9s’ / Twitter.” Accessed December 4, 2019. https://twitter.com/lukebroadwater/status/677246260995366912.

“Mistrial in the First Trial of One of the Officers Accused of Killing Freddie Gray - TwitterTrails.” Accessed December 4, 2019. http://twittertrails.wellesley.edu/~trails/stories/investigate.php?id=190816579.

Twitter. “Profile / Twitter.” Accessed December 4, 2019. https://twitter.com/jenn_abrams.

Ruiz, Rebecca R. “Baltimore Officers Will Face No Federal Charges in Death of Freddie Gray.” *The New York Times*, September 12, 2017, sec. U.S. https://www.nytimes.com/2017/09/12/us/freddie-gray-baltimore-police-federal-charges.html.

“Update on Twitter’s Review of the 2016 US Election.” Accessed November 24, 2019. https://blog.twitter.com/en_us/topics/company/2018/2016-election-update.html.